Testing Documentation

Team 81

Andrew Durnford, Eric Jiang, Calvin Luo, Edison Qu, Surajj Vinodh Kumar

2.1 Main Page

Task / Part	Contributors	Description / Notes
Vital statistics Rules Parental controls Score	Calvin and Edison	Worked on the back end logic and systems for the pet, especially the vital stats like the health and hunger
Instructions Main menu Sprites	Surajj	Created a tutorial screen as well as a main menu. Also got sprites for the pets
Save/Load system Shop and Inventory	Eric	Created the save/load CSV system, with simple get/set methods. Also created the player's inventory and the shop
Pet commands	Andrew	Created the commands that the player can give to the pet (sleep, feed, play, etc)
Main page Test plan Summary	Eric, Surajj	Created this page as well as the testing plan

2.2 Introduction

2.2.1 Overview

This document outlines the testing phase of our virtual pet game, which was designed and implemented in Java. The game allows players to care for a digital pet by managing its hunger, happiness, health, and sleep. The testing process was crucial to ensure the stability, functionality, and usability of the application before final deployment.

The project has successfully completed the implementation phase, with the core game mechanics and graphical user interface (GUI) built using Java and Java Swing. To ensure the quality and reliability of the codebase, we employed rigorous testing methodologies, primarily utilizing JUnit for unit testing. Additionally, Maven was integrated into the development pipeline to streamline dependency management and test execution.

Our testing strategy consisted of the following methodologies:

- **Unit Testing:** JUnit was used to test individual components, including pet behavior logic, state transitions, and interaction handling.
- **Integration Testing:** Ensured that different modules (such as the pet's state management and GUI interactions) worked seamlessly together.
- **ValidationTesting:** Assessed core features such as feeding, playing, resting, and the effects of neglecting the pet over time.
- **System Testing:** Assessed the program when run as a whole

2.2.2 Objectives

The main objectives of the testing phase were:

- **Ensure code correctness** by validating pet behavior logic and game mechanics.
- **Verify integration** between game logic and the graphical user interface.
- **Test user interactions** to confirm smooth gameplay and UI responsiveness.
- **Identify and fix bugs** to improve overall game stability.
- **Ensure performance efficiency** so that the game runs smoothly under different conditions.

- **Validate functional requirements** by confirming that all core game features work as intended.
- **Prevent regression issues** by ensuring new changes do not introduce unexpected bugs.

2.2.3 References

- CS2212B Group Project Specification, Winter 2025.
- Requirements Documentation
- Java Swing Documentation: https://docs.oracle.com/javase/tutorial/uiswing/
- GitLab Documentation: https://docs.gitlab.com/
- JUnit Documentation: https://junit.org/junit5/
- Maven Documentation: https://maven.apache.org/

2.3 Test plan

Unit Testing

Unit testing focuses on individual components of the software. Each test case will be aimed at verifying the correctness of methods, especially business logic in classes like DataManager, GameProgress, and others. We've implemented comprehensive JUnit 5 tests for all major classes, particularly focusing on data management classes that handle core application data.

Test Case Name:	*DataManager Test*
Test Case Description:	*Verify the DataManager initialized correctly with default settings.*
Test Steps:	• *Initiate the DataManager class* • *Verify the default state of the data.* • *Check if data is reset correctly.*
Pre-Requisites:	*DataManager class exists, and test framework JUnit is set up.*
Expected Results:	*DataManager should initialize with default values, and reset functionality should work.*
Test Category:	*Unit Test*
Requirement:	*Data management*
Automation:	*Automated JUnit*
Date Run:	*03/28/2025 5:00PM*

Pass/Fail:	*Pass*
Test Results:	*Initialized correctly with the expected data*
Remarks:	None

Test Case Name:	*GameProgress Test*
Test Case Description:	*This test verifies that the GameProgress class correctly retrieves and updates player progress data from the CSV file.*
Test Steps:	● *Create a GameProgress instance with a test CSV file.* ● *Add a new player entry with sample data.* ● *Retrieve and validate the stored level, experience, quests completed, and earned badges.* ● *Update player level, experience, quests, and badges.* ● *Retrieve the updated values and check if they match the expected changes.*
Pre-Requisites:	● *A test CSV file with predefined player data.* ● *A valid GameProgress instance.*
Expected Results:	● *Initially retrieved player data should match test data.* ● *Updated values should be correctly reflected in subsequent retrievals.*
Test Category:	*Unit Test*
Requirement:	*Ensuring GameProgress correctly reads, writes, and updates player data.*

Automation:	*Automated JUnit*
Date Run:	*03/29/2025 3:00PM*
Pass/Fail:	*Pass*
Test Results:	*Initialized correctly with the expected data*
Remarks:	None

Test Case Name:	*Inventory Test*
Test Case Description:	*This test verifies that the Inventory class correctly retrieves, adds, updates, and removes inventory items stored in the CSV file.*
Test Steps:	● *Create an Inventory instance with a test CSV file.* ● *Add a new item to the inventory.* ● *Retrieve and validate the item details (ID, name, quantity, type).* ● *Update item quantity and type.* ● *Remove an item from the inventory.* ● *Retrieve the updated inventory list and ensure changes are correctly reflected.*
Pre-Requisites:	● *A test CSV file with predefined inventory data.* ● *A valid Inventory instance.*
Expected Results:	● *Initially retrieved inventory data should match test data.* ● *Newly added items should be correctly stored and retrievable.* ● *Updated values should be correctly reflected in subsequent retrievals.* ● *Deleted items should no longer be present in the inventory.*

Test Category:	*Unit Test*
Requirement:	*Ensuring Inventory correctly reads, writes, updates, and deletes inventory items.*
Automation:	*Automated JUnit*
Date Run:	*03/30/2025 4:00PM*
Pass/Fail:	*Pass*
Test Results:	*Initialized correctly with the expected data*
Remarks:	None

Test Case Name:	*Parental Controls Test*
Test Case Description:	*This test verifies that the ParentalControls class correctly retrieves, updates, and enforces parental control settings stored in the CSV file.*
Test Steps:	● *Create a ParentalControls instance with a test CSV file.* ● *Retrieve parental control settings for a specific player.* ● *Update restrictions such as playtime limits.* ● *Verify updated values are correctly reflected.*
Pre-Requisites:	● *A test CSV file with predefined parental control settings.* ● *A valid ParentalControls instance.* ● *Mock players with different control settings for verification.*
Expected Results:	● *Initially retrieved control settings should match test data.*

	● *Updated values should be correctly stored and retrievable.* ● *Restrictions should be enforced correctly, preventing unauthorized actions.*
Test Category:	*Unit Test*
Requirement:	*Ensuring ParentalControls correctly manages and enforces parental control settings.*
Automation:	*Automated JUnit*
Date Run:	*03/29/2025 4:00PM*
Pass/Fail:	*Pass*
Test Results:	*Initialized correctly with the expected data*
Remarks:	None

Test Case Name:	*Players Test*
Test Case Description:	*This test suite validates the functionality of the Players class, ensuring accurate retrieval and updating of player information stored in a CSV file.*
Test Steps:	● *Set up a temporary CSV file with test player data.* ● *Initialize the Players object with the test CSV file.* ● *Execute individual test cases for various player attributes (ID, username, email, join date, last login, currency, and score).* ● *Validate that data retrieval methods return expected values.* ● *Validate that data update methods correctly modify and store the values.*

Pre-Requisites:	● *Java Development Environment.* ● *JUnit 5 for running unit tests.* ● *Test CSV file with sample player data.*
Expected Results:	● *Each test should successfully retrieve the correct player data and accurately update values. No incorrect or missing data should be present.*
Test Category:	*Unit Test*
Requirement:	*The system should be able to fetch and modify player details stored in a CSV file*
Automation:	*Automated JUnit*
Date Run:	*03/28/2025 12:00PM*
Pass/Fail:	*Pass*
Test Results:	*Initialized correctly with the expected data*
Remarks:	None

Test Case Name:	*Store Test*
Test Case Description:	*This test suite verifies the functionality of the Store class, ensuring that prices for player-owned items can be retrieved and updated correctly.*
Test Steps:	● *Create a temporary CSV file with sample store inventory data.* ● *Initialize a Store object with the test CSV file.* ● *Test the getPrice method for retrieving item prices.* ● *Test the setPrice method for updating item prices.*

	• *Validate if the correct exceptions are thrown for invalid player or item IDs.*
Pre-Requisites:	• *Java Development Environment.* • *JUnit 5 for running unit tests.* • *Test CSV file with sample player data.*
Expected Results:	• *The getPrice method should return the correct price for an existing player-item pair.* • *The setPrice method should successfully update the price and reflect changes.* • *Retrieving a non-existent player or item should return an error.*
Test Category:	*Unit Test*
Requirement:	*TThe system should correctly retrieve and update store item prices for players using the CSV file.*
Automation:	*Automated JUnit*
Date Run:	*03/29/2025 11:00AM*
Pass/Fail:	*Pass*
Test Results:	*Initialized correctly with the expected data*
Remarks:	None

Test Case Name:	*Transactions Test*
Test Case Description:	*This test case verifies that transaction details can be correctly retrieved and updated in the CSV file. It also checks for proper error handling when invalid data is provided.*

Test Steps:	● *Load a test CSV file with sample transactions.* ● *Retrieve transaction details using getTransactionId(), getPlayerId(), getTransactionType(), getAmount(), and getTimestamp().* ● *Modify values using setTransactionId(), setPlayerId(), setTransactionType(), setAmount(), and setTimestamp().* ● *Retrieve the modified values to ensure they were updated correctly.* ● *Test edge cases (e.g., retrieving a non-existent transaction, setting a negative amount).*
Pre-Requisites:	● *A valid CSV file containing sample transaction data must exist and be accessible to the test.*
Expected Results:	● *Transactions should be retrieved correctly.* ● *Updated transactions should persist and be correctly retrieved.* ● *Invalid operations (e.g., retrieving a non-existent transaction or setting a negative amount) should throw appropriate exceptions.*
Test Category:	*Unit Test*
Requirement:	*The system should support transaction retrieval and updates while handling invalid inputs correctly.*
Automation:	*Automated JUnit*
Date Run:	*03/28/2025 11:00AM*
Pass/Fail:	*Pass*
Test Results:	*Initialized correctly with the expected data*
Remarks:	None

Test Case Name:	*Virtual Pets Test*
Test Case Description:	*Validate the functionality of the Virtual Pets system, ensuring correct pet creation, feeding, and interaction features work as expected.*
Test Steps:	● *Create a new virtual pet with a unique name and type.* ● *Feed the pet and check if hunger level decreases.* ● *Play with the pet and observe happiness level changes.* ● *Let the pet idle for some time and verify that hunger and happiness change accordingly.* ● *Attempt to overfeed or overplay with the pet and check for system constraints.* ● *Close and reopen the application to confirm data persistence.*
Pre-Requisites:	● *Test is set up.* ● *Database or storage is functional for data persistence.*
Expected Results:	● *The pet should be successfully created.* ● *Feeding should reduce hunger level appropriately.* ● *Playing should increase happiness level.* ● *Overfeeding/overplaying should trigger proper warnings.* ● *Pet status should update correctly over time.* ● *Data should persist across sessions.*
Test Category:	*Unit Test*
Requirement:	*Ensure proper handling of virtual pet interactions, including feeding, playing, and idling mechanics.*
Automation:	*Automated JUnit*
Date Run:	*03/28/2025 6:00PM*

Pass/Fail:	*Pass*
Test Results:	*Initialized correctly with the expected data*
Remarks:	None

Test Case Name:	*Vital Stats Test*
Test Case Description:	*Validate the functionality of the VirtualStats module to ensure correct computation and display of user statistics.*
Test Steps:	● *Initialize the VirtualStats class with test data.* ● *Call the method responsible for computing statistics.* ● *Verify that the returned values match expected results.* ● *Test with boundary values (e.g., zero, max limits).* ● *Test with invalid inputs (e.g., negative values, null).*
Pre-Requisites:	● *A test user account should be available.* ● *Test dataset with predefined values should be accessible.*
Expected Results:	● *The VirtualStats methods return correct calculated statistics.* ● *No unexpected exceptions are thrown for valid inputs.* ● *Proper error handling is in place for invalid inputs.*
Test Category:	*Unit Test*
Requirement:	*Ensure VirtualStats computes and returns accurate statistics based on input data.*

Automation:	*Automated JUnit*
Date Run:	*03/29/2025 7:00PM*
Pass/Fail:	*Pass*
Test Results:	*Initialized correctly with the expected data*
Remarks:	None

Integration testing

The focus of integration testing is on the coming together and connecting of the above units, as reflected in the software's overall design and architecture.

Test Case Name:	*Main menu buttons*
Test Case Description:	*Verify whether the main menu buttons correctly opens the correct screen, and vice versa*

Test Steps:	• *Check all buttons*
	  ○ *New game*
	  ○ *Load game*
	  ○ *Info*
	  ○ *Parental controls*
	  ○ *Exit*
	• *Verify whether the back buttons from these screens goes back to the main menu*

Pre-Requisites:	*Main menu is functional, and prototypes of the screens exist. The back buttons of the screens must be implemented*

Expected Results:	*The buttons lead to the correlating screens*
Test Category:	*Integration test*
Requirement:	*Multiple Screens: The game must have at least five distinct screens/pages including: a main menu screen (requirement 3.1.2), a gameplay screen, an instructional/tutorial screen (requirement 3.1.3), a load game / new game screen (requirement 3.1.4), and a parental controls screen (requirement 3.1.11). Your team is free to add additional screens as required.*
Automation:	*Manual*
Date Run:	*03/28/2025 3:00PM*
Pass/Fail:	*Pass*
Test Results:	*Menu and back buttons lead to the correct screen*
Remarks:	None

Test Case Name:	*Store & Inventory*
Test Case Description:	*Verify whether the store correctly updates the inventory, and correctly integrates with the player's currency*
Test Steps:	• *Buy items* • *Verify whether the items are added to the inventory* • *Buy items with insufficient currency*

	● *Verify whether items are not added to the inventory*
Pre-Requisites:	*Pet menu buttons are functional, store and inventory are complete. Player CSV handler is finished.*
Expected Results:	*Bought items are correctly added to the inventory*
Test Category:	*Integration test*
Requirement:	*3.1.13 One Extra Functional Requirement* *Add more items, currency, and an item store.*
Automation:	*Manual*
Date Run:	*03/28/2025 3:00PM*
Pass/Fail:	*Pass*
Test Results:	*Items correctly go to the inventory and updates correctly*
Remarks:	None

Test Case Name:	*Inventory & pets*
Test Case Description:	*Verify whether the inventory interacts with the pet properly*

Test Steps:	● *Feed, gift, and heal the pet* ● *Verify whether the corresponding items update in the inventory* ● *Repeat but with insufficient items* ● *Verify whether the pet stats do not update upon unsuccessful usage*
Pre-Requisites:	*Inventory and pet interaction screen are finished. Player csv is finished*
Expected Results:	*Items are properly used, or not used if no items are available for the category*
Test Category:	*Integration test*
Requirement:	*3.1.7 Commands The player interacts with the pet by issuing commands. The commands available are dependent on the pet's state (see requirement 3.1.6). b) Feed: The pet is fed food, and their fullness value increases (they are less hungry). The player should be able to select from different food types each with their own properties (e.g., some might add more to the fullness value than others). The food available should be based on the player's inventory (see requirement 3.1.8).* *c) Give Gift: The pet is given a gift by the player and their happiness value increases. The player should be able to select from different gift types each with their own properties (e.g., some might add more happiness than others). The gifts available should be based on the player's inventory (see requirement 3.1.8).* *d) Take to the Vet: The player takes the pet to the vet and the pet's health is increased by a set amount. Once this command is used, it should be unavailable for a set time until a cool down is over.*
Automation:	*Manual*

Date Run:	*03/28/2025 3:00PM*
Pass/Fail:	*Pass*
Test Results:	*Item and pet stats updated properly upon usage (or lack thereof)*
Remarks:	Requirement d) was expanded upon, giving healing items as well as the option for the vet

For integration testing, we followed a **Bottom-Up Integration** approach. We first implemented and tested in-game features such as pet commands, inventory management, the shop system, and interactions with the pet. After confirming their functionality, we worked backward to integrate higher-level components such as the pet selection screen and main menu. To facilitate this process, we used simple driver classes that supplied hand-crafted pet and player data, ensuring that lower-level modules functioned correctly before integrating them into the broader system.

Validation testing

In this testing, requirements established as part of requirements modelling are validated against the software has been constructed.

Test Case Name:	*User Interface*
Test Case Description:	*An easy to navigate Graphical User Interface (GUI), largely mouse based, should inform players about the pet's state, and respond to the user's inputs.*
Test Steps:	● *All screens must be consistent* ● *Must have mouse-based interactions* ● *Should provide necessary info about the pet* ● *At least 5 distinct screens must be present* ● *User input must have feedback* ● *Check all buttons* ○ *New game* ○ *Load game*

	○ *Info* ○ *Parental controls* ○ *Exit* ● *A graphic representing the game is required* ● *The names must be listed* ● *The team number must be listed* ● *The term & course (2212) must be listed* ● *Info screen must have the required info*
Pre-Requisites:	*All GUIs must be implemented*
Expected Results:	*The buttons lead to the correlating screens*
Test Category:	*Validation test*
Requirement:	*3.1.1 User Interface Your application needs to provide a Graphical User Interface (GUI) that players of the intended age group can navigate easily. This interface should be at least partly mouse based, inform players about the pet's state, and respond to the user's inputs. The following UI requirements must be included: Multiple Screens: The game must have at least five distinct screens/pages including: a main menu screen (requirement 3.1.2), a gameplay screen, an instructional/tutorial screen (requirement 3.1.3), a load game / new game screen (requirement 3.1.4), and a parental controls screen (requirement 3.1.11). Your team is free to add additional screens as required. Mouse-Based Interaction: The interface must support mouse-based interactions, enabling users to navigate through the game's screens, make selections, and control game elements primarily with a mouse (e.g., click a button that feeds the pet, etc.). Keyboard Shortcuts and Commands: The interface should include keyboard shortcuts and commands for common actions to facilitate ease of use, especially for power users or when a mouse is not the most efficient input method or for accessibility reasons. Where relevant, include keyboard controls for gameplay actions (e.g., pressing 'P' to pause, 'F' to feed, 'G' to give a gift, 'Esc' to return to the main menu, etc.). Feedback Systems: The UI must provide visual or auditory feedback in response to user actions, such as clicking buttons or entering commands, to confirm that the desired action has been taken.*

	3.1.2 Main Menu Your game must have a main menu screen that displays the title of your game and provides options for the user to select from. These options must include 1) start a new game, 2) load a previously saved game, 3) tutorial or instructions, 4) parental controls, 6) exit. Your team may add additional options. Each option should take the player to a new screen in your application. The main menu should also display: 1) some graphic or visual that is representative of your game, 2) list the names of the developer's who created it (this would be the members of your team), 3) your team's number, 4) term it was created in (e.g. "Fall 2024), and 5) mention that this was created as part of CS2212 at Western University. *3.1.3 Instructions and/or Tutorials You must include at least one screen with detailed instructions on how to play your game and use this application. This can be text based, include pictures, or be an interactive tutorial. In all cases it must be comprehensive enough to inform players with no previous experience with your software, how to play your game. It should document all major features from the players perspective (i.e. it does need to include low level technical details). When creating your instructions or tutorial, you should keep in mind the target demographic for your game and write these instructions appropriately for their age level.*
Automation:	*Manual*
Date Run:	*03/28/2025 3:00PM*
Pass/Fail:	*Pass*
Test Results:	*Menus are consistent, main menu buttons work, main menu has the image and the info at the bottom, and the info screen has the basic info suited for the target demographic*

Remarks:	None

Test Case Name:	*New pets, save/load*
Test Case Description:	*The pet selection as well as the save/load system should work*
Test Steps:	● *Make a new pet* ○ *At least 3 options should be available* ● *Modify the pet (feed, let the stats decay, etc)* ● *Close the game & exit to the main menu (two separate tests)* ● *Load and check whether the stats match* ● *The user should be able to load any of their pets*
Pre-Requisites:	*All menus are implemented*
Expected Results:	*The pet stats should persist between sessions, and the user should be able to create a new pet*
Test Category:	*Validation test*
Requirement:	*3.1.4 New Game & Pet Selection When a new game is selected, the player is presented with a choice of picking from at least 3 different virtual pet types. An image representing each pet type should be displayed on the screen and well as some basic information about each pet type. The user should be able to select one of the pets and give it a name. Once named, a new save file (see requirement 3.1.5) is created, and the user is brought to the main game screen. Each pet should have slightly different characteristics that impact gameplay. For example, one pet might become hungry faster than the others, another might require less gifts to stay happy, and so on. The characteristics of each pet are up to your team, but they should be noticeably*

	different in terms of game paly and described to the player when selecting a pet. As mentioned, each pet should be represented by an image. The image should match up with the sprite2 used to represent the pet in the game. Your team is not required to create their own images, and it is acceptable to use premade sprites found online so long as you credit where they were obtained from. *3.1.5 Save/Load Game State It is important to provide players with the ability to save their progress and the current state of their pet and inventory (requirement 3.1.8). To achieve this, the game must feature a Save Game State mechanism that specifically captures the player's progression and the current state of the pet. This should include any vital statistics about the pet, the type of pet being used, the pet's name, their current health, hunger, the current score, and the player's inventory, etc. The primary goal is to enable players to resume their progression from their last play session. To facilitate this, you can either automatically save progress after set check points or provide players with a manual save option that is easily accessible somewhere in the GUI. When restarting the game, players should have the capability to load their previously saved state. This loading process should load the pet back into the same state it was in when the game was last saved. A clear confirmation message should be displayed each time the game state is saved, reassuring players that their progress has been successfully recorded. Your Save/Load system must be designed in such as way that a player can have multiple pets that they can switch between by saving and then loading a new pet. You may either have the player select a file that contains the save state to load or have a set number of save slots for the user to choose from (at least 3).*
Automation:	*Manual*
Date Run:	*03/28/2025 3:00PM*
Pass/Fail:	*Pass*

Test Results:	*The use can create a dog, cat, or dragon. Multiple pets can be created, saved, and loaded, and the data persists*
Remarks:	None

Test Case Name:	*Pet gameplay*
Test Case Description:	*Playing with the pet - the main gameplay*
Test Steps:	● *All vital stats should be implemented* ○ *They should decay over time EXCEPT health* ● *Various conditions should apply when vital stats reach a critical threshold* ● *The player should be able to issue various commands:* ○ *Go to bed* ○ *Feed* ○ *Gift* ○ *Take to vet* ○ *Play* ○ *Exercise* ● *The appropriate sprite reflecting the status should be displayed*
Pre-Requisites:	*Pet selection, load, and interaction window should be implemented*
Expected Results:	*The player should be able to play with the pet as expected*
Test Category:	*Validation test*
Requirement:	*All of 3.1.6 Vital Statistics & Rules, 3.1.7 Commands,*

	and 3.1.10 Pet Sprite (too long to paste here)
Automation:	*Manual*
Date Run:	*03/28/2025 3:00PM*
Pass/Fail:	*Pass*
Test Results:	*The vital stats work and decay as well as the various states. The commands work, and the pet sprite reflects the state.*
Remarks:	None

Test Case Name:	*Inventory, score, bonus feature (store)*
Test Case Description:	*The item system and player's score should be implemented*
Test Steps:	● *Test the inventory* ● *Test the store* ● *Test whether buying from the store updates the inventory* ● *Test whether using items updates the score*
Pre-Requisites:	*The inventory, store, and pet interaction window should be finished*
Expected Results:	*The player should be able to acquire and use items*
Test Category:	*Validation test*

Requirement:	*3.1.8 Player Inventory The game should keep track of and display the player's current inventory of items. Items fall into two categories: a) Food Items: This category of items increases the pet's fullness value (making them less hungry). Each food item should have different properties and increases the fullness value by a different amount. b) Gift Items: This category of items increases the pet's happiness level. Each gift item should have a different property and increase the happiness level by a different amount. The game should have a minimum of three items from each category. The inventory should display a list of the items currently in the player's inventory including a count of how many they have of each item type. This can be a text-based listing or displayed pictorially (e.g. with icons). The inventory can be its own screen in the application or part of the gameplay screen. Using the Give Gift and Feed commands (from requirement 3.1.7) should lower the amount of that item in the inventory. The game should include a mechanic that allows the player to obtain items. This can be as simple as giving the player items periodically after a set amount of time, rewarding them with items at set score levels (requirement 3.1.9), or as complex as requiring them to complete a minigame first. The details of the mechanic are up to your team, but there must be a way of obtaining items. The players inventory should be saved as per requirement 3.1.5. 3.1.9 Keeping Score The game should keep track of the player's score. The score should start at zero and increase when the player does a positive action such as feeding the pet, giving it a gift, or playing with it. Some actions may also reduce the score, such as subtracting points if the pet had to be taken to the vet. The exact details of what adds and subtracts score, or how much is up to your team but should be clearly documented. The current score should be displayed on the game play screen and saved in the save file as per requirement 3.1.5. 3.1.13 One Extra Functional Requirement* ● *Add more items, currency, and an item store.*
Automation:	*Manual*
Date Run:	*03/28/2025 3:00PM*

Pass/Fail:	*Pass*
Test Results:	*The item system works as expected. Score properly updates both when doing positive actions (feeding, gifting) and subtracts when doing negative actions (healing, vet)*
Remarks:	None

Test Case Name:	*Parental controls*
Test Case Description:	*The parental controls should be implemented*
Test Steps:	● *Access the parental controls from the main menu* ● *Password lock should work and the parent should be able to change it* ● *Statistics should be shown* ● *Parents should be able to play as a player* ● *Reset statistics* ● *Other features:* ○ *Limit play time* ○ *Total screen time as well as average screen time* ○ *Revive pet*
Pre-Requisites:	*All features need to be complete - statistics depend on other systems*
Expected Results:	*The parent should be able to view and reset statistics*
Test Category:	*Validation test*

Requirement:	*3.1.11 Parental Controls The game should provide a separate area accessible via the main menu (requirement 3.1.2 Main Menu) that provides controls and statistics for the parent of the player. This screen should be password protected to prevent the player from accessing it. You may have a hardcoded password or pin; user accounts are not required. Parent users are also allowed to play the game as a player would and can perform all tasks a player can.*
Automation:	*Manual*
Date Run:	*03/28/2025 3:00PM*
Pass/Fail:	*Pass*
Test Results:	*The parent can view the statistics, reset them, and set a new password as well as play as a regular player. They can also revive the pet*
Remarks:	None

Test Case Name:	*Housekeeping, non-functional requirements*
Test Case Description:	*Various other miscellaneous features*

Test Steps:	● *Exit the program*
	● *Check whether the data saves properly*
	● *Try invalid input*
	● *Try to get to the main menu at all stages*
	● *Non-functional requirements:*
	  ○ *Must work in Java 23*
	  ○ *Object-oriented approach*

	○ *Check offensive content* ○ *GUI* ○ *No internet connection required, store local data* ○ *Only freely available dependencies and libraries* ○ *Check to see if it's all in GitLab* ○ *Check if comments are commented with Javadoc* ○ *Check if JUnit 5 tests are used* ○ *Check to see if it modifies files it does not create* ○ *Error messages must be shown* ○ *The entire project just be under 500 megabytes* ○ *Check if it runs smoothie* ○
Pre-Requisites:	*All features implemented*
Expected Results:	*The player should be able to acquire and use items*
Test Category:	*Validation test*
Requirement:	*3.1.12 Housekeeping & Error Handling Your application also needs to do basic housekeeping things that are part of any decently put together application. For example, the user must be able to exit your application cleanly and data must be saved correctly on exit such that it is available the next time the application is started (e.g. save states should be persistent, parental statistics should not be lost, and so on). If you allow the user to minimize or maximize the window, UI elements should scale correctly. It is ok if you disable this and do not allow the user to minimize or maximize the window. You may also force the game to be full screen or a set window size. Any errors that could be raised by your application should be handled and clear messages in simple English should be shown to the user, that may help them understand and correct the error. Ideally, your application should avoid crashing without explanation. Any user input that could potentially be incorrect or cause faults in your software, must be validated and checked for errors. There must always be a way to navigate between the screens of*

	the application. The user should never get stuck on a screen with no way back to the main menu. You may add other housekeeping functions as you see fit to help support your application. How you choose to make these available to the user is up to you. *All of 3.2 Non-Functional Requirements*
Automation:	*Manual*
Date Run:	*03/28/2025 3:00PM*
Pass/Fail:	*Pass*
Test Results:	*All housekeeping rules were followed as well as non-functional requirements*
Remarks:	None

System testing

Here, the software and other system elements are tested as a whole.

Test Case Name:	*Stress Testing*
Test Case Description:	*Trying to overload the system*
Test Steps:	• *Click all buttons extremely rapidly, try to quickly access menus*

Pre-Requisites:	*Entire project must be implemented*
Expected Results:	*The system acts the same as if actions were slow*
Test Category:	*System test*
Requirement:	*N/A*
Automation:	*Manual*
Date Run:	*03/28/2025 4:00PM*
Pass/Fail:	*Pass*
Test Results:	*Buttons as well as the game act as expected when rapidly clicking as well as moving through screens*
Remarks:	None

Test Case Name:	*Performance Testing*
Test Case Description:	*Testing the performance*
Test Steps:	● *Run the game on an old computer*
Pre-Requisites:	*Entire project must be implemented*

Expected Results:	*The game must act reasonably fast on an old laptop*
Test Category:	*System test*
Requirement:	*N/A*
Automation:	*Manual*
Date Run:	*03/28/2025 4:00PM*
Pass/Fail:	*Pass*
Test Results:	*The game works smoothly on an old device*
Remarks:	None

Test Case Name:	*Recovery Testing*
Test Case Description:	*Recovering from a failure*
Test Steps:	● *Delete various data files*
Pre-Requisites:	*Entire project must be implemented*
Expected Results:	*The system generates new files upon deletion as if the game were booted for the first time*

Test Category:	*System test*
Requirement:	*N/A*
Automation:	*Manual*
Date Run:	*03/28/2025 4:00PM*
Pass/Fail:	*Pass*
Test Results:	*The game correctly generates new files*
Remarks:	None

2.4 Summary

This document details the successful implementation and testing phase of the virtual pet game, developed using Java and Java Swing. The testing process focused on ensuring code correctness, integration, UI functionality, and overall game stability. Unit and integration tests were conducted using JUnit and Maven, confirming the reliability of game mechanics. The GUI underwent manual testing to verify responsiveness and usability. Results indicate that the game meets functional requirements and runs efficiently. With no major defects found, the game is now ready for deployment, with potential future refinements based on feedback.