SE 3XA3: Test Plan
Namcap

Team 2, VPB Game Studio
Prajvin Jalan (jalanp)
Vatsal Shukla (shuklv2)
Baltej Toor (toorbs)

December 8, 2016

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	**Version**	**Notes**
2016-10-30	1.0	Addition of content to sections 1.1, 1.2 and 1.4
2016-10-30	1.1	Addition of content to sections 2.1 and 2.2
2016-10-30	1.2	Addition of content to sections 2.3 and 2.4
2016-10-30	1.3	Addition of content to section 3.2
2016-10-30	1.4	Updates to content of subsections of 3.2 and 7.1
2016-10-30	1.5	Completion of content for 3.2 and update to 7.1
2016-10-31	1.6	Completion of content for 3.1
2016-10-31	1.6	Completion of Section 7.2
2016-10-31	1.7	Completion of content for 4
2016-10-31	1.8	Completion of section 1.3; revision of sections 1, 2, and 4
2016-10-31	1.9	Addition of section 6.1 and 6.2
2016-10-31	2.0	Completion of section 5
2016-10-31	2.1	Updates to Sections 3.1 and 6.1
2016-10-31	2.2	Completion of section 6.1
2016-10-31	2.3	Revision of System and Unit tests
2016-10-31	2.4	Addition to content to 6.1 and 3.2
2016-11-29	2.5	Modifications for Ghost to Enemy refactoring
2016-12-07	2.6	Modifications to Unit Test Cases
2016-12-08	2.7	Final revision based on feedback - removal of already covered test case and additional mentioning of code coverage metric to be used

This document is the test plan for the Pacman redevelopment project, Namcap. The test plan outlines the testing methodologies and techniques to be used when testing the functionalities and characteristics of the system and its component parts.

1 General Information

1.1 Purpose

The purpose of testing is to ensure that the developed implementation functions correctly and to address any areas where the system is vulnerable. Through the formal specification of the testing methods and verification techniques, testing the implementation becomes more reliable.

1.2 Scope

As Namcap is a redevelopment of a classic arcade game, the test plan will aim to formalize the various functionality testing techniques as well as the usability tests utilized in order to ensure that the implementation meets the given requirement. This document will explicitly detail the different methods and testing tools to be utilized for this project.

1.3 Acronyms, Abbreviations, and Symbols

Table 2: **Table of Abbreviations**

Abbreviation	**Definition**
PoC	Proof of Concept
GUI	Graphical User Interface
AI	Artificial Intelligence
JAR	Java Archive (file)
OS	Operating System
JVM	Java Virtual Machine

Table 3: **Table of Definitions**

Term	Definition
Structural Testing	(White box testing) derived from the program's internal structure.
Functional Testing	(Black box testing) derived from a description of the program's function.
Dynamic Testing	Testing that requires program execution.
Static Testing	Testing that does not involve program execution.
Manual Testing	Testing that is conducted by people, by-hand testing.
Automated Testing	Testing that occurs automatically, usually set-up with testing frameworks.

1.4 Overview of Document

This document will describe the testing methodologies to be utilized to verify Namcap as an implementation and development. The test plan will outline all testing tools, schedules, automated and manual tests, tests to address the requirements for the application, unit tests, and any additional testing performed on the PoC and existing implementation.

2 Plan

2.1 Software Description

Namcap is a redevelopment of the classic 2D arcade game Pacman. The gameplay involves the player sprite moving through a 2D level attempting to acquire (collide with) as many dots as possible to increase the score. Enemy sprites (ghosts in the original) will move throughout the level and upon collision with the player will cause the player to lose a life and/or end the game. The player can however consume (collide with) a power up to send the enemies back to the center of the level (when collision occurs). The implementation covers these aspects of the core gameplay (scoring, collision, movement, and enemy mechanics).

2.2 Test Team

The test team for Namcap is comprised of Prajvin Jalan, Vatsal Shukla, and Baltej Toor.

2.3 Automated Testing Approach

For the purposes of automated testing, the test team will use both JUnit and the built-in Robot class library. JUnit is a unit testing framework that will run automated tests for most logical components and any GUI components where applicable. The Robot class library will be used to automate testing that simulates user input. Based on the simulated user input, logical and GUI components will be tested to ensure that the appropriate collision and scoring responses occur.

2.4 Testing Tools

The automated unit testing tool JUnit and the Robot class library are the testing tools that the team will use to verify the implementation. Future development will make use of the JaCoCo Java code coverage library to measure the degree to which the source code is executed when running both JUnit and Robot (manual/automated) test cases. JaCoCo uses class file instrumentation to record execution coverage data.

2.5 Testing Schedule

Follow this link to the [Namcap Gantt Project] (must have project file structure).

3 System Test Description

3.1 Tests for Functional Requirements

3.1.1 Game Functionality Testing

A Robot (automated) unit testing class will be implemented and used to test the mechanics of the game.

1. GFT1

   Type: Functional, Dynamic, Automated

   Initial State: Application is displaying the main menu page

   Input: Cursor clicked on Start Game button

   Output: New game is started and window is changed to reflect a new game state

   How test will be performed: The Robot class will place the cursor within the coordinates of the Start Game button and the Robot will perform a left-click.

2. GFT11

   Type: Functional, Dynamic, Automated

   Initial State: Within game state

   Input: Escape button pressed

   Output: Application must pause and ask user if they want to quit.

   How test will be performed: The Robot class will press the escape key. If the game pauses, the test will pass.

Player Movement/Collision Testing

1. GFT2

   Type: Functional, Dynamic, Automated

   Initial State: Within the game state

   Input: Arrow keys

   Output: Player moves in the respective direction (if path is clear)

   How test will be performed: The Robot class will virtually press the left/right/up/down arrow. The Robot class will output text indicating current player direction.

2. GFT3

   Type: Functional, Dynamic, Automated

   Initial State: Player comes in contact with wall

   Input: No input

   Output: Player stops moving when coming in contact with the wall

   How test will be performed: The Robot class will output a line of text to the console indicating that the player has stopped due to collision with a wall.

3. GFT4

   Type: Functional, Dynamic, Automated

   Initial State: Player comes in contact with enemy

   Input: No input

   Output: If player has more than 1 life, decrement lives. If player has one life, end game.

   How test will be performed: The Robot class will navigate the player towards an enemy until they collide.

4. GFT6

   Type: Functional, Dynamic, Automated

   Initial State: Player comes in contact with dots

   Input: Arrow keys

   Output: Dot disappears after collection

   How test will be performed: The Robot class will navigate the player through the map and collect the dots. A function will be implemented to check if the dots disappear upon collection. A text message will be printed to indicate dot consumption.

5. GFT7

Type: Functional, Dynamic, Automated

Initial State: Player collects the big dot

Input: Arrow keys

Output: Big dot disappears after collection

How test will be performed: The Robot class will navigate the player to the nearest big dot and collect it. A function will be implemented to check if the big dot disappears upon collection. A message will be printed to indicate big dot consumption.

6. GFT8

Type: Functional, Dynamic, Automated

Initial State: Player collects the big dot

Input: Arrow keys

Output: Player is able to collide with enemies

How test will be performed: The Robot class will navigate the player to the nearest enemy, after collecting the big dot. A function will be implemented to check if the collision does not decrement the player's lives.

Enemy Movement/Collision Testing

1. GFT5

Type: Functional, Dynamic, Automated

Initial State: Within the game state

Input: No input

Output: Enemies move on a valid path

How test will be performed: A JUnit test will be implemented to fail whenever the enemy moves through an obstacle. The coordinate of the breached wall will be printed to indicate where the error occurred.

2. GFT9

   Type: Functional, Dynamic, Automated

   Initial State: Player collects the big dot

   Input: No input

   Output: Enemies change colour

   How test will be performed: A JUnit test will be implemented to check if the asset used to view the enemies has changed. A message will be printed to indicate that the colour change has occurred.

3. GFT14

   Type: Functional, Dynamic, Automated

   Initial State: Player collides with enemy after collection of big dot

   Input: Arrow keys

   Output: Enemy is removed from game and respawned back to their original cell

   How test will be performed: A JUnit test will be implemented to check if the enemy's coordinates were reset to their original spot.

Scoring Testing

1. GFT10

   Type: Functional, Dynamic, Automated

   Initial State: Player collects all dots

   Input: Arrow keys

   Output: Game over screen is activated

   How test will be performed: A JUnit test will be implemented to check which JFrame is currently active. A message will be printed to indicate that the game responded with game over response.

2. GFT12

   Type: Functional, Dynamic, Automated

   Initial State: Player collects dot

   Input: Arrow keys

   Output: The points are increased

   How test will be performed: A JUnit test will be implemented to check if the player's points were increased. Message to be printed to the screen to indicate correct increment.

3. GFT13

   Type: Functional, Dynamic, Automated

   Initial State: Player collects big dot

   Input: Arrow keys

   Output: The points are increased at twice the rate

   How test will be performed: A JUnit test will be implemented to check if the player's points were increased by twice the standard rate. Message to be printed to screen to indicate correct increment.

3.2 Tests for Non-functional Requirements

This section of System Testing will consist of manual tests since it would be infeasible to implement automated tests for a majority of the subsections of Non-Functional Requirements. Manual tests will be performed by users, therefore many of the tests will correspond with the Usability Survey such that testers can ensure that these requirements are met.

3.2.1 Look and Feel Requirements

Application Layout

1. AL1

   Type: Functional, Dynamic, Manual

Initial State: Application is displaying the instructions page

Input: No user input

Output: Namcap instructions page

How test will be performed: User will look at the Namcap instructions page and ensure that the layout and colour scheme of the display is similar (but not identical) to that of the original arcade Pacman.

2. AL2

Type: Functional, Dynamic, Manual

Initial State: Application is displaying the game board with game entities

Input: No user input

Output: Namcap game board with game entities

How test will be performed: User will look at the Namcap game board with its game entities and ensure that the layout and colour scheme of the display is similar (but not identical) to that of the original arcade Pacman.

3.2.2 Usability and Humanity Requirements

Understandable Objectives

1. UO1

Type: Functional, Dynamic, Manual

Initial State: Application is displaying the instructions page

Input: No user input

Output: Namcap instructions page in English

How test will be performed: User will look at the Namcap instructions page and ensure that the instructions are in English.

2. UO2

Type: Functional, Dynamic, Manual

Initial State: Application is displaying the instructions page

Input: No user input

Output: Namcap instructions page clearly defining objectives

How test will be performed: User will look at the Namcap instructions page and ensure that the objective of the game is clear before playing the game.

Understandable Gameplay

1. UG1

Type: Functional, Dynamic, Manual

Initial State: Application is displaying the game board with game entities

Input: Custom user input for gameplay

Output: Game display updates accounting for movement, collision, scoring, and enemy mechanics

How test will be performed: User will play Namcap (one playthrough - 3 lives) and be able to successfully maneuver the Player through dots and enemies, reinforcing their understanding of the game's controls and objective.

3.2.3 Performance Requirements

Response Time

1. RT1

Type: Functional, Dynamic, Manual

Initial State: Application is displaying the game board with game entities

Input: Custom user input for gameplay

Output: Game display updates player movement based on keyboard input with a delay less than Θ

How test will be performed: User will play Namcap and note down any instances where their player does not respond to their keyboard inputs within a delay of Θ.

Unexpected Failures

1. UF1

   Type: Functional, Dynamic, Manual

   Initial State: Application is displaying the game board with game entities

   Input: Custom user input for gameplay testing

   Output: Game display updates for user input and application does not crash

   How test will be performed: User will play Namcap and note down any instances where during their testing, the application unexpectedly crashes (the number of crashes will be within Ω of total user tests).

2. UF2

   Type: Functional, Dynamic, Manual

   Initial State: Application is displaying the game board with game entities

   Input: Highscore text file for high score in game display [file exists, file does not exist, valid score in file, invalid input in file]

   Output: Game display updates score or an exception occurs

   How test will be performed: User will play Namcap (second playthrough) and a high score will be read from a text file if possible, and exceptions will be tested for based on the various inputs

3.2.4 Maintainability and Support Requirements

Operating System Support

The primary operating systems that Namcap is being tested on currently are Windows and Mac OS since these are the most readily available for testers.

1. OSS1

   Type: Functional, Dynamic, Manual

   Initial State: A Namcap JAR file is available on a machine running Windows

   Input: User runs the JAR file

   Output: Namcap starts up and game can be played

   How test will be performed: User will run the JAR file on their computer (any Windows version with a JVM) and should be able to successfully play the game (one playthrough).

2. OSS2

   Type: Functional, Dynamic, Manual

   Initial State: A Namcap JAR file is available on a machine running Mac OS

   Input: User runs the JAR file

   Output: Namcap starts up and game can be played

   How test will be performed: User will run the JAR file on their computer (any Mac OS version with a JVM) and should be able to successfully play the game (one playthrough).

3.2.5 Security Requirements

Open-Source Code

1. OSC1

   Type: Structural, Static, Manual

   Initial State: Open-source repository is publicly accessible

   Input: User accesses the open-source repository (visits the repository link)

   Output: User is able to view the application source code

   How test will be performed: User will visit the repository link and be able to view the application source code.

3.2.6 Cultural Requirements

Offensive Symbols or Text

1. OST1

   Type: Functional, Dynamic, Manual

   Initial State: Application is displaying the instructions page

   Input: No user input

   Output: Namcap instructions page

   How test will be performed: User will look at the Namcap instructions page and ensure that it contains no offensive symbols or text.

2. OST2

   Type: Functional, Dynamic, Manual

   Initial State: Application is displaying the game board with game entities

   Input: No user input

   Output: Namcap game board with game entities

   How test will be performed: User will look at the Namcap game board with its game entities and ensure that it contains no offensive symbols or text.

3.2.7 Legal Requirements

Legal Violation

1. LV1

   Type: Functional, Dynamic, Manual

   Initial State: Application is displaying the instructions page

   Input: No user input

   Output: Namcap instructions page

How test will be performed: User will look at the Namcap instructions page and ensure that literary work and listed game entities have been altered such that the redevelopment is not too similar to the original Pacman.

2. LV2

   Type: Functional, Dynamic, Manual

   Initial State: Application is displaying the game board with game entities

   Input: No user input

   Output: Namcap game board with game entities

   How test will be performed: User will look at the Namcap game board with its game entities and ensure that literary work and game entities have been altered such that the redevelopment is not too similar to the original Pacman.

3.2.8 Health and Safety Requirements

Break Prompt

1. BP1

   Type: Functional, Dynamic, Manual

   Initial State: Application is displaying the instructions page

   Input: No user input, application runs for Φ

   Output: After Φ, application pauses and prompts the user to take a break

   How test will be performed: User will let the application run for Φ and ensure that when the time passes the application pauses appropriately and prompts them to take a break.

4 Tests for Proof of Concept

Before additional features can be added to the game, a basic proof of concept will be carried out to show that the project is feasible.

4.1 Significant Risk

In order for the game to run, it must be successfully compiled. Therefore, the game is intended to work on all operating systems as long as they have the latest version on Java installed. However, this can be a risk if the game is unable to run on some operating systems.

4.2 Demonstration Plan

For the proof of concept, a working prototype of the game will be produced that will run by opening a JAR file on any operating system with Java installed. The prototype will consist of the game which implements a player and enemy movement, collision detection, points system, and boundary detection.

The game will consist of a basic Pacman map layout in which a player character and enemy character will exist. The player and enemy character will start at a specific point on the map. Neither the player nor the enemy can pass through the walls on the map. The player will be able to collect the dots to increment the game points. Upon collision with the enemy, the game will end and output the final score.

The player character will be represented by a pacman .png file that will be located in the assets package within the project. The enemy character will be represented by a .png file that will also be located in the assets package. The player will be controlled by the user by using the arrow keys on the keyboard.

The enemy will have a simple AI programmed for the proof of concept. The AI will allow the enemy to move in the direction they are currently moving unless it hits a barrier. Then, the AI will randomly generate a direction for the enemy to move in. The enemy will move in the new generated direction if the path is clear, else, another direction is generated.

Proof of Concept Testing Refer to the Tests for Functional Requirements section for the following PoC tests:

1. PoCT1

   Type: Functional, Dynamic, Manual

   Refer to: GFT1

Description: The app can be opened

2. PoCT2

   Type: Functional, Dynamic, Manual
   Refer to: GFT2
   Description: The player can be moved

3. PoCT3

   Type: Functional, Dynamic, Manual
   Refer to: GFT3
   Description: Player cannot move through walls

4. PoCT4

   Type: Functional, Dynamic, Manual
   Refer to: GFT4
   Description: The game will exit when player collides with enemy

5. PoCT5

   Type: Functional, Dynamic, Manual
   Refer to: GFT5
   Description: Enemies move on a valid path

6. PoCT6

   Type: Functional, Dynamic, Manual
   Refer to: GFT6, GFT12
   Description: The player can collect dots and increase points

7. PoCT7

   Type: Functional, Dynamic, Manual

   Refer to: GFT10

   Description: Game over when all dots are collected

5 Comparison to Existing Implementation

To compare our implementation with the existing implementation, we will manually perform our system tests on the existing project and compare the outputs. If the outputs are similar/same, we have implemented the project correctly.

Refer to the System Test Description section for the following system test IDs. These tests will be performed on the existing implementation in parallel with the developed implementation:

- GFT2
- GFT3
- GFT4
- GFT6
- GFT7
- GFT8
- GFT9
- GFT10
- GFT12
- GFT13
- GFT14
- RT1
- UF1

6 Unit Testing Plan

6.1 Unit testing of internal functions

A JUnit testing class will be implemented and used to test specific aspects of the application. These tests will be more specific than the functional tests and will address methods within the source code.

1. UT1

   Type: Unit, Static, Automated

   Initial State: Application is displaying the main menu page

   Input: Start Game button action is performed

   Output: New game is started and window is changed to reflect a new game state (Pass or fail for JUnit method)

   How test will be performed: A JUnit method will test the method that opens a new frame containing the game and confirm that the new window has the appropriate game interface [refer to GFT1 test]

2. UT2

   Type: Unit, Static, Automated

   Initial State: Player is in starting position at the start of the game

   Input: Current X position of the player

   Output: Pass or fail for JUnit method

   How test will be performed: A JUnit method will test the player's getCurrentX method to ensure their current X position equals their initial start position of the game.

3. UT3

   Type: Unit, Static, Automated

   Initial State: Player is in starting position at the start of the game

   Input: Current Y position of the player

Output: Pass or fail for JUnit method

How test will be performed: A JUnit method will test the player's getCurrentY method to ensure their current Y position equals their initial start position of the game.

4. UT4

   Type: Unit, Static, Automated

   Initial State: Player is in starting position at the start of the game

   Input: Current direction of the player

   Output: Pass or fail for JUnit method

   How test will be performed: A JUnit method will test the player's getCurrentDirection method to ensure their current direction equals the direction of their initial start position of the game.

5. UT5

   Type: Unit, Static, Automated

   Initial State: Player is in starting position at the start of the game

   Input: PlayerX, PlayerY, EnemyX, EnemyY

   Output: End of game (Pass or fail for JUnit method)

   How test will be performed: A JUnit method will set the player's position to be within collision range of an enemy, and the checkCollision method will be tested to ensure it properly checks player-to-enemy collision and ends the game [refer to GFT4 test]

6. UT6

   Type: Unit, Static, Automated

   Initial State: Player is in starting position at the start of the game, all dots are on map

   Input: PlayerX, PlayerY (different parts of the map grid)

   Output: Increase in score and dot disappearance (Pass or fail for JUnit method)

How test will be performed: A JUnit method will test the player's checkDot method to ensure that when they are on a dot that their score is increased and the dot has disappeared (board's getDot method) [refer to GFT6 test]

7. ~~UT7~~

   ~~Type: Unit, Static, Automated~~

   ~~Initial State: Player is in starting position at the start of the game, all dots on map are clear except one~~

   ~~Input: PlayerX, PlayerY~~

   ~~Output: End of game (Pass or fail for JUnit method)~~

   ~~How test will be performed: A JUnit method will test the player's checkDot method to ensure that when they collect the final dot that the endGame method is called and the player wins the game [refer to GFT10 test]~~

8. UT7

   Type: Unit, Static, Automated

   Initial State: Player is in starting position at the start of the game

   Input: X and Y positions for barrier and non-barrier positions

   Output: True or false for barriers (Pass or fail for JUnit method)

   How test will be performed: A JUnit method will test the player's checkBarrier method to ensure that when a position is checked, the appropriate response is received for if barriers exist or not [refer to GFT3 test]

9. UT8

   Type: Unit, Static, Automated

   Initial State: Board is created with only barrier and dot entities

   Input: X and Y positions for barrier locations

Output: Expected true for all barrier locations (Pass or fail for JUnit method)

How test will be performed: A JUnit method will test the board's getBarrier method to ensure that barriers exist in all the appropriate locations.

10. UT9

    Type: Unit, Static, Automated

    Initial State: Board is created with only barrier and dot entities

    Input: X and Y positions for locations without barriers

    Output: True or false for new barrier locations (Pass or fail for JUnit method)

    How test will be performed: A JUnit method will test the board's updateBarrier method to ensure that barriers can be successfully added to specific locations on the map (getBarrier to check).

11. UT10

    Type: Unit, Static, Automated

    Initial State: Board is created with only barrier and dot entities

    Input: X and Y positions for dot locations

    Output: Expected true for all dot locations (Pass or fail for JUnit method)

    How test will be performed: A JUnit method will test the board's getDot method to ensure that dots exist in all the appropriate locations.

12. UT11

    Type: Unit, Static, Automated

    Initial State: Board is created with only barrier and dot entities

    Input: X and Y positions for locations without dots

    Output: Integer value for dot locations (type of dot that exists here) (Pass or fail for JUnit method)

How test will be performed: A JUnit method will test the board's updateDot method to ensure that dots can be successfully added to specific locations on the map (getDot to check).

13. UT12

    Type: Unit, Static, Automated

    Initial State: Board is created with game entities, player and score objects are created

    Input: Score values (negative, positive, values that increase the score past the maximum)

    Output: Player score, and end game situation if applicable (Pass or fail for JUnit method)

    How test will be performed: A JUnit method will test the Score class's addScore method to ensure that score can be appropriately added; the Player's checkDot method will be tested to see whether the game ends depending on the score [refer to GFT12 test]

6.2 Unit testing of output files

1. UTF1

    Type: Unit, Dynamic, Automated

    Initial State: Application is in gameplay state

    Input: High score within game and High Score stored in text file

    Output: Pass or fail for JUnit method

    How test will be performed: A JUnit class will test the method that reads a high score from the text file and check whether or not the in game high score matches that of the text file [refer to UF2 test]

7 Appendix

7.1 Symbolic Parameters

Table 4: **Symbolic Constants**

Constant	Value	Description
Θ	0.5 seconds	Response time between user actions and in-game operations
Ω	1 %	Percent of application tests that will cause unexpected failures in the application
Φ	2 hours	Duration that application runs for before prompting user to take a break

7.2 Usability Survey Questions

The following usability survey will be filled out by users to evaluate playability of the game and test for some non-functional requirements where manual testing is required.

Figure 1: User Survey

Usability Survey Questions

Please complete the following survey as you play through the game. The first section covers usability of the application and your thoughts on the game in terms of general entertainment. The second section helps us test our game to ensure we can provide a stable application for the public.

Playability

Entertainment: 0 1 2 3 4 5 6 7 8 9 10
[where 10 is most entertaining]

Interface: 0 1 2 3 4 5 6 7 8 9 10
[where 10 is most user-friendly]

Game Difficulty: 0 1 2 3 4 5 6 7 8 9 10
[where 10 is most difficult]

User Testing

Does the instructions page look identical to that of the original Pacman?	Yes	No		
Does the game interface look identical to that of the original Pacman?	Yes	No		
Does the instructions page clearly define the objective of Namcap?	Yes	No		
Are you able to fully grasp the overall gameplay after one playthrough (3 lives)?	Yes	No		
Do you notice any offensive symbols or text in any part of the game?	Yes	No		
How many times did the game unexpectedly crash?	0	1	2	3+
Were there instances where your key presses had a response delay of more than 1 second?	0	1	2	3+