Quality Assurance

Contents

1 Test Specification **2**

1.1 Scope 2

1.2 Test Results 3

1.3 Test Plan 3

1.3.1 Test Phases 3

1.4 Unit Tests 4

1.4.1 Brain 4

1.4.2 Game 4

1.4.3 Ant 7

1.4.4 World 9

2 Acceptance Criteria **10**

2.1 Introduction 10

2.2 Test Environment 11

2.3 Acceptance Criteria Specification 11

2.4 Release Tests 11

1 Test Specification

1.1 Scope

The tests will cover all possible control flow in the program. This will be examined by JaCoCo, a code coverage library for Java. JaCoCo examines the bytecode of the program, and produces percentages of the statements covered by existing tests for each method, each class, and overall.

Tests shall aim to keep code coverage above 80% at all times. As discussed in the project plan, this will be managed by splitting the team into two groups for the development stage - one group to write code, and the other to write tests for it.

In addition, code coverage should aim to be 'above' 100% in areas: although not required, the members of the testing group will aim to assist the developers using test-driven development - i.e. by writing tests for a particular program submodule before the coders have built it.

Tests shall be primarily aimed at targeting the explicit functional requirements of the project. As such, each functional requirement shall generate multiple tests, aiming to strictly enforce that functional requirement.

Tests shall also enforce program behaviours described by the class diagrams and sequence diagrams of the modelling phase.

The continuous integration software used in the remote Git repository, Travis CI, runs code coverage and automatics tests upon pushes to the repository (regression testing). The status of these operations, displayed in `README.md` in the root project directory, shall be monitored to ensure test coverage is up-to-date and that code is in line with the requirements.

When a test fails, the programmers shall be alerted to the location of the error, pertinent details, and the related functional requirement. This will allow the developers to adjust the program to suit the user's requirements. Where the source of an error cannot be found, breakpoints shall be used for debugging.

Tests shall have extensive documentation. Unit test classes shall contain class-level documentation including a link to the class being tested. For example:

```
This class is testing: {@link antgame.core.Ant}
```

In addition, each test method shall contain links to each method that is being tested. It shall also describe the functional requirement that the test is written to validate. For example:

`This method tests that an ant is not able to move onto an already-occupied cell: {@link antgame.core.Ant#move(int)}`

1.2 Test Results

Final test results of the submission are included in a `.zip` folder alongside this document. The test results are presented in HTML format.

1.3 Test Plan

1.3.1 Test Phases

Phase 1 - Unit Testing This phase is to be worked on concurrently while the programming sub-team produces code. As described above, code coverage should be maintained at above 80% at all times.

JUnit 4.2 shall be used as the testing suite on all the tester's platforms.

Phase 2 - Release Testing This phase takes place after development is complete. As per the project plan, this should take approximately one week to complete thoroughly.

Release testing shall be undertaken by the entire team.

It shall involve running the entire system, interacting through the GUI, and examining the system's response to both typical and edge cases (e.g. attempting to load an empty ant brain, or adding no teams to a tournament).

Overhead Software/Libraries

- JUnit 4.2 - external library, managed with Gradle
- JaCoCo - external library, managed with Gradle

1.4 Unit Tests

1.4.1 Brain

	Instruction Tests
Prerequisites	-
Description	We shall be testing the functionality of the `Sense`, `Mark`, `Unmark`, `Drop`, `Pickup`, `Move`, `Flip` and `Turn` instructions followed by the `Condition` class representing a type of condition used by an `Ant` during a sense. The Condition will be tested by ensuring all conditions match their type and `getMarker()` on a non marker condition followed by a marker condition. Sense will be tested by ensuring whether the direction and condition of the sense are set correctly. The `Mark` and `Unmark` instruction's `getMarker()` will be tested to ensure the marker to place/remove is returned correctly. `Flip` will be tested to ensure the upper bound on the random number generator is set correctly. `Drop`, `Pickup` and `Move` will be tested to ensure they have the correct identifier and `Instruction.Type`, and transition to the next state appropriately on success and failure.
Expected Result	A successful test is determined by whether an instruction has the correct Instruction identifer and that their type matches the tested instruction and whether it transitions to the next state on success and failure appropriately. In terms of the `Condition`, all conditions should match their type and `getMarker()` on a marker condition should return a marker and on a non marker condition should throw an exception. `Sense` should set the direction and condition of the instruction correctly. The `Mark` and `Unmark` instructions `getMarker()` method should return an instruction. The `Flip` instructions's `getRange()` should return the upper bound on the random number generated. Finally `Drop`, `Pickup` and `Move` will be successful if they have the correct identifier and that their instruction type matches.

	Adding Instructions to the Brain
Prerequisites	-
Description	Different instructions shall be added to the Brain and `getInstructionGraph()` will be tested to ensure that the latest added `Instruction` is always returned.
Expected Result	A successful test is determined by whether the last added instruction is returned by a call to `getInstructionGraph()` in the `Brain` class.

1.4.2 Game

	Parse an Ant Brain
Prerequisites	An ant-brain file.
Description	Checks if a user's custom ant brain (expected to have been provided in a file on disk) is valid according to the ant brain specification. If so, constructs an in-memory programmatic representation of the brain.
Expected Result	A programmatic representation of the ant brain. It iterates through each line of the file, tokenising the line and validating each token. Builds a graph of ant-brain instructions which form a finite-state machine with transitions between instructions dependent on the conditions specified in the line.

	Simulate 2-Player Game
Prerequisites	Two different colonies (consisting of a colour, a team name and an ant-brain) and an ant-world.
Description	a simulation of the game based on the ant finite-state machines and their environment.
Expected Result	For each turn (maximum 300,000), the ants on each cell shall be iterated over. Each ant will perform an action based on their ant-brain. The state of a cell (and possibly other ants) will be updated based on the action of the ant.

	Produce Statistics for Game
Prerequisites	The state of an ant-game at a particular turn (accumulative) or the full history of an ant-game at the end of the game.
Description	For each team the following statistics should have been computed: • Amount of food in anthill • Number of foe's ants killed • Number of team's ants left • Number of movements made • Number of markings left
Expected Result	Given the state of the game at a particular turn, all relevant statistics counters shall be updated. At the end of the game, the statistics shall be calculated and output.

	Simulate a Tournament
Prerequisites	A list of teams (specified by a name and an ant-brain), and an arbitrary (but at least 1) number of ant-worlds.

Description	Plays ant games between multiple users, each with their own ant-brain specification.
Expected Result	Should iterate over all unique combinations (team-1, team-2, world) and simulate a game with that 3-tuple, with the first team playing as red and the second team playing as black. Should keep track of all wins, losses and statistics for each team. After all simulations, the overall winner should be returned.

	Uploading Ant Brains and Worlds
Prerequisites	Ant brain and ant world files.
Description	The user can upload brains into an ant world with two ant colonies, one for each player and it should check if an ant-brain and ant world supplied by a player is syntactically well-formed.
Expected Result	The program should be able to load the ant-brain and generate random but well-formed ant worlds.

	Two-Player Game
Prerequisites	-
Description	The program should allow two players to compete in a match.
Expected Result	enables two players to upload their ant-brains and choose an ant-world, and then run the game in the ant world, taking statistics and determining the winner of the game.

	Tournament
Prerequisites	-
Description	The program should allow users to play tournaments.
Expected Result	an arbitrary number of players can upload ant-brains, who are all paired up to play against each other. The overall tournament winner is the ant brain that wins the most individual games. In a tournament, each team will play every other team twice on each of the contest worlds. Once as red and once as black. So for 23 teams, each team will be pitted up against 22 other teams.

	Winning a Game
Prerequisites	-
Description	A test to check for the correct winner of a match.
Expected Result	The match is won by the species with the most food in its anthill at the end of 300,000 rounds.

1.4.3 Ant

	Sensing a Cell
Prerequisites	-
Description	Performs a sense operation on a particular cell of a particular condition (cell and condition specified by the ant-brain instruction).
Expected Result	The ant's current instruction stores information about the direction to sense in and the condition to sense for. The direction is one of here, ahead, left-ahead or right-ahead. The ant queries the world to determine the cell in the relevant direction, and checks the condition on that cell. If the condition holds for that cell, it transitions to the success state. If not, it transitions to the fail-state.

	Marking a Cell
Prerequisites	-
Description	Performs a mark operation on the current cell, placing a chemical marker as specified by the current ant-brain instruction.
Expected Result	Tells the world to mark the ant's current cell with a particular integer. Transitions to the next state.

	Unmarking a Cell
Prerequisites	-
Description	Performs an unmark operation on the current cell, removing a chemical marker as specified by the current ant-brain instruction.
Expected Result	Tells the world to unmark the ant's current cell, removing the marker with a particular identifier. Transitions to the next state.

	Picking up Food
Prerequisites	-
Description	Picks up food from the current cell, transitioning to one of two ant-brain states depending on whether the pick-up operation was successful.
Expected Result	Queries the world to see if the current cell has food. If not, transitions to the fail-state. Otherwise sets the internal state of the ant to has-food = true, as per requirement Ant/Misc.4. Removes a food particle from the current cell. Transitions to the success state.

	Dropping Food

Prerequisites	-
Description	Drops food into the current cell, transitioning to the next state as specified in the ant-brain finite state machine.
Expected Result	Gets the current cell from the world context and adds a food particle to it. Transitions to the next state.

	Turning Direction
Prerequisites	-
Description	Causes the internal direction attribute of the ant (as specified in Ant/Misc.7) to change based on the current turn instruction's specification.
Expected Result	The direction specified by the instruction can be one of left or right. This shall cause the direction of the ant to 'rotate' - i.e. left will cause the ant to rotate one edge anticlockwise in its hexagonal cell, and vice-versa for right. The ant will then transition into the next state.

	Moving
Prerequisites	-
Description	Causes the ant to move forward based on its direction into an adjacent cell (if possible).
Expected Result	Based on the ant's current direction and the ant's current cell, the world can be queried for the cell in front of the ant. If the cell is free (not rocky and does not contain another ant), the ant shall remove itself from the current cell and add itself to the relevant adjacent cell. It shall then update its own resting attribute to indicate to the game controller that it must now rest for 14 moves. Then it will transition to the success state. If not free, the ant shall transition to the fail-state.

	Flip
Prerequisites	-
Description	Causes the ant to transition into one of two states depending on the result of a pseudo-random generated integer.
Expected Result	Using the system's random number generator, the ant shall produce a random integer between 0..(n-1), where n is the integer specified by the flip instruction. If 0, the ant shall transition to state-1. Otherwise, it shall transition to state-2.

	Marking and Unmarking

Prerequisites	-
Description	Ants perform the marking and and unmarking of chemical cells functionality.
Expected Result	Ants communicate or leave trails by means of chemical markers.

	Sensing
Prerequisites	-
Description	Ants perform the sense function successfully.
Expected Result	Each species of ants can sense (with limited capabilities), but not modify, the markers of the other species.

	Deaths
Prerequisites	-
Description	Tests when ants are supposed to be killed.
Expected Result	If an ant is adjacent to 5 (or 6) ants of the other species then it dies. However, if an ant is in a cell in the very corner of the grid or along one of the edges it can only have at most 4 ants adjacent to it - therefore it cannot be killed.

1.4.4 World

	Parse Ant World File
Prerequisites	An ant-world file.
Description	Checks if a user's custom ant world (expected to have been provided in a file on disk) is valid according to the ant world specification. If so, constructs an in-memory programmatic representation of the world.
Expected Result	Iterates through each line of the file, tokenising the line and validating each token. Interpreting each token as a cell and building a hexagonal programmatic structure containing cells with properties specified in the file.

	Spawning Ants
Prerequisites	An ant-world file.
Description	Checks for successful spawning of ants.
Expected Result	The two species of ants are placed in a random world containing two anthills, some food sources, and several obstacles.

	Simulating Behaviour
Prerequisites	An ant-world file, correct ant behaviour.
Description	Checks that the behaviour of ants inside the world is correct.
Expected Result	An ant world should simulate the behaviour of both kinds of ants, using the ant-brains supplied by the players.

	Cell Contents
Prerequisites	An ant-world file.
Description	A cell holds an unlimited amount of food during the course of the simulation.
Expected Result	Food is always non-negative.

	Cell Positioning
Prerequisites	A contest ant-world file.
Description	Checks for validity of the ant world.
Expected Result	There must be at least one empty cell between adjacent non-food items.

2 Acceptance Criteria

2.1 Introduction

This section describes the final validation and verification stage. This acceptance criteria specification primarily focuses on the user and release testing phase of the project, rather than the development testing that is described in the test specification document.

The two main procedures to be considered in ensuring the acceptance of the complete product will be:

- Validation: ensuring that all components of the program satisfy the *user* requirements.
- Verification: ensuring that all components of the program satisfy the *functional* and *non-functional* requirements.

2.2 Test Environment

For this final testing stage, the test environment shall be made as 'realistic' as possible. That is, unless debugging, the tests shall be carried out in a program environment that is similar to the environment in which the client will use the program. These program environments are specified in the non-functional requirements (see supported systems in the non-functional requirements document). This will essentially cover user and release testing, wherein the program shall be used by the team as if they were a user.

In particular, the system should be tested on each of the three main operating systems as identified in the non-functional requirements.

2.3 Acceptance Criteria Specification

The acceptance tests shall involve using the program with the client-provided data files (using various combinations of ant-brains and ant-worlds). The system should be 'stress-tested', meaning:

- Non-conventional behaviour - experimenting with the GUI performing relatively uncommon actions (e.g. running a game without specifying an ant-brain, specifying no teams in a tournament, etc.) to check that the program does not fall into an inconsistent state.
- Large tasks - for example, testing the system with large ant brain and world files.

In addition to the above, the user requirements shall be individually checked against the program by each member of the team. The non-functional requirements should also be checked by each member of the team, from a user's perspective.

2.4 Release Tests

Along with checking all user requirements, the following requirements will be tested:

	Visualising Ant Worlds
Description	Uses a Swing GUI to show users a representation of the ant world in a game at a particular turn.
Expected Result	Game draws cells to screen, with an indication of the type of cell and what it contains specified by a colour.

	Reliability of Parsing User Files

Description	The software shall parse user files of up to 10,000, even if the files are corrupt or not well-formed.
Expected Result	The software does not crash more than 1 time in 1000.

	Reliability of Game
Description	The software should crash no more than 1 time in 1000 when simulating a game or tournament.
Expected Result	The software does not crash and visualisations of games and tournaments should proceed with no visible defects (glitches) at least 99 times out of 100. If the program crashes, a log containing the stack trace of the error and any other pertinent details shall be written to the program's working directory, titled "ant-game-crash-[current unix time].log".

	Software Usability
Description	The software should have a minimal learning curve for new users. That is, the GUI should have intuitive and clear controls that step the user(s) through each stage of the game without being required to follow external instructions (not including understanding the rules of the game and the development of a user's ant-brain file).
Expected Result	The software shall present a help menu, accessible at all times in the program, which upon clicking displays a popup user interface containing the searchable user documentation for the program.

	Software Performance
Description	Parsing user files (ant-brain, ant-world) should take no longer than 5 seconds, The GUI should not hang (that is, be unresponsive to user interaction) for more than one second after any user interaction. The two-player game simulation should take no longer than 30 seconds for 300,000 rounds.
Expected Result	The software performs to the specified performance timings above.

	Operating System Compatibility
Description	The software shall be cross-platform across commonly used personal desktop operating systems (Windows 7+, Mac OS X 10.8.3+, Linux) that support Java 8.0.
Expected Result	The software functionals correctly on all platforms.