Test Plan (TP)

Intelligent Lifestyle

Team Daedalus (`s440gf`)

Kieran James Simpson - (`kieranjs`)
Nathaporn Eiamvittayakorn - (`neiam`)
Quyen Le Quach - (`qlq`)
Tjhandana Masyuri Jaya - (`masyurit`)
Wendy Wai-Tak Tsang - (`wwttsang`)

Revision: 1.0.0
29 October 2004

Maintained by: Tjhandana Masyuri Jaya

Abstract

The Test Plan (TP) formally describes the scope, approach and processes of testing activities Team Daedalus performs. Also, this document contains descriptions of test items, features to be tested, features not to be tested and Team responsibilities in conducting testing activities.

Contents

1 Introduction **9**
1.1 Purpose . . . 9
1.2 Scope . . . 9
1.3 Test Plan Objectives . . . 9
1.4 Intended Audience . . . 9
1.5 Project Overview . . . 10
1.6 Personnel . . . 10
1.6.1 Development Team . . . 10
1.6.2 Supervisor . . . 11
1.6.3 Clients . . . 11
1.7 Definitions and Acronyms . . . 11
1.7.1 Definitions . . . 11
1.7.2 Acronyms . . . 12
1.8 References . . . 13
1.8.1 Standards . . . 13
1.8.2 Project documentation . . . 13
1.8.3 Online References . . . 14
1.8.4 Other References . . . 14

2 Project Description **15**
2.1 Project Scope . . . 15
2.2 Project Process Model . . . 15

3 Testing Objectives **16**
3.1 Test Items . . . 16
3.1.1 Code . . . 16
3.1.2 Classes and Modules to be Tested . . . 16
3.1.3 User Documentation(UD) . . . 17
3.1.4 Features to be tested . . . 17

4 Testing Methodology **18**
4.1 Testing Life Cycle Concepts . . . 18
4.2 V&V Life Cycle Model . . . 20
4.3 Verification and Validation Concepts . . . 20

5 Testing Strategy **21**
5.1 Unit Testing . . . 21
5.2 Integration Testing . . . 21
5.3 Agent Testing . . . 22
5.4 System Testing . . . 22
5.5 Reliability Modelling . . . 22
5.6 Installation Testing . . . 22
5.7 Usability Testing . . . 23

6 Testing Techniques **24**
6.1 Grey Box Testing . . . 24
6.1.1 Rationale for Adopting Grey Box Testing . . . 24
6.1.2 Applying Grey Box Testing . . . 24
6.2 Exception Testing . . . 25
6.2.1 Rationale for Adopting Exception Testing . . . 25
6.2.2 Applying Exception Testing . . . 25
6.3 Database Testing . . . 26
6.3.1 Types of Database . . . 26
6.3.2 Rationale of Database Testing . . . 26
6.3.3 Applying Database Testing . . . 26
6.3.3.1 Things to be tested . . . 27
6.4 Boundary Value Analysis Testing . . . 27
6.4.1 Rationale of Adopting Boundary Value Analysis Testing . . . 27
6.4.2 Applying Boundary Value Analaysis Testing . . . 28
6.5 Coverage Testing . . . 28
6.5.1 Types of Coverage Testing . . . 28
6.5.2 Rationale of Adopting Coverage Testing . . . 29
6.5.3 Applying Coverage Testing . . . 29

7 Unit Testing **30**
7.1 General Strategy . . . 30
7.2 Entry Criteria . . . 30
7.3 Exit Criteria . . . 30
7.4 Pass/Fail Criteria . . . 30
7.5 Test Cases and Reports . . . 31
7.6 Procedures . . . 31
7.6.1 Test Request . . . 31
7.6.2 Conduct Testing . . . 31
7.6.3 Execute Unit Test Cases . . . 32

8 Integration Testing **33**
8.1 General Strategy . . . 33
8.2 Entry Criteria . . . 33
8.3 Exit Criteria . . . 33
8.4 Pass/Fail Criteria . . . 34
8.5 Test Cases and Reports . . . 34
8.5.1 Naming Conventions . . . 34
8.6 Procedures . . . 34

9 System Testing **35**
9.1 General Strategy . . . 35
9.2 Entry Criteria . . . 35
9.3 Exit Criteria . . . 35
9.4 Pass/Fail Criteria . . . 35
9.5 Test Cases and Reports . . . 35
9.6 Procedures . . . 36

10 Agent Testing **37**
10.1 General Strategy . . . 37
10.2 Entry Criteria . . . 37
10.3 Exit Criteria . . . 37
10.4 Pass/Fail Criteria . . . 37
10.5 Test Cases and Reports . . . 37
10.5.1 Naming Conventions . . . 38
10.6 Procedures . . . 38
10.6.1 Conduct Testing . . . 38
10.6.2 Execute Agent Test Cases . . . 38

11 Acceptance Testing **39**
11.1 General Strategy . . . 39
11.2 Entry Criteria . . . 39
11.3 Exit Criteria . . . 39
11.4 Pass/Fail Criteria . . . 39
11.5 Test Cases and Reports . . . 39
11.6 Acceptance Testing Procedures . . . 40
11.6.1 Creating Acceptance Tests . . . 40
11.6.2 Pre-acceptance Testing Activities . . . 40
11.6.3 Performing Acceptance Testing . . . 41
11.6.4 Post-acceptance Testing Activites . . . 41

12 Reliability Modelling **42**
12.1 General Strategy . . . 42
12.2 Entry Criteria . . . 42
12.3 Exit Criteria . . . 42
12.4 Pass/Fail Criteria . . . 43
12.5 Test Cases and Reports . . . 43
12.6 Procedures . . . 43

13 Installation Testing **44**
13.1 General Strategy . . . 44
13.2 Entry Criteria . . . 44
13.3 Exit Criteria . . . 44
13.4 Pass/Fail Criteria . . . 45
13.5 Test Cases and Reports . . . 45
13.6 Procedures . . . 45

14 Usability Testing **46**
14.1 General Strategy . . . 46
14.2 Entry Criteria . . . 46
14.3 Exit Criteria . . . 47
14.4 Pass/Fail Criteria . . . 47
14.5 Test Cases and Reports . . . 47
14.6 Procedures . . . 47
14.6.1 Heuristic Evaluation Procedures . . . 47
14.6.1.1 Human-Device Interaction . . . 48

14.6.1.2 Guiding . . . 49
14.6.2 Cognitive Walkthrough Procedures . . . 49
14.6.2.1 Questionaire . . . 50

15 Testing Management Tools - TMT **51**
15.1 Tester's Responsibilities . . . 51
15.2 Coding Sub-Team Responsibilities . . . 52
15.3 Testing Sub-Team Responsibilities . . . 52

16 Test Case Design **53**
16.1 Unit Testing . . . 53
16.2 Integration Testing . . . 53
16.3 System Testing . . . 53
16.4 Agent Testing . . . 53
16.5 Rationale in Agent Testing . . . 54
16.6 Creating Agent Test Case . . . 54

17 Language Constraints **55**

18 Testing Tools **56**
18.1 Tools . . . 56
18.2 NUnit . . . 56
18.2.1 Rationale for Choosing NUnit . . . 56
18.2.2 Benefits of NUnit . . . 56
18.2.3 How to Use NUnit . . . 57
18.3 JUnit . . . 57
18.3.1 Rationale for choosing JUnit . . . 57
18.3.2 Benefits of JUnit . . . 57
18.3.3 How to Use JUnit . . . 58
18.4 Apache Ant . . . 58
18.4.1 Benefits of Apache Ant . . . 58
18.4.2 How to Use Apache Ant . . . 58

A Templates **59**
A.1 Change Log . . . 59
A.2 Build File . . . 60
A.3 Unit Testing Test Cases for Java . . . 65
A.4 JUnit Assertions . . . 68
A.5 Integration Testing Test Cases . . . 69
A.6 Integration Testing Report . . . 70
A.7 Agent Testing Test Cases . . . 71
A.8 Agent Testing Report . . . 73
A.9 Agent Testing Tester Agent . . . 74
A.10 Agent Testing Test List . . . 77
A.11 Reliability Modelling Report . . . 78
A.12 Usability Testing Templates - PDA Heuristic Evaluation Checklist . . . 80
A.13 Usability Testing Templates - STT Heuristic Evaluation Checklist . . . 81
A.14 Usability Testing Templates - TTS Heuristic Evaluation Checklist . . . 82

A.15 Installation Testing Report 83

List of Figures

1 *Testing Lifecyle Concept* 19
2 *V-Model* 20

List of Tables

1 Development Team . . . 10
2 Acceptance Testing Checklist Specification . . . 40
3 Change Log . . . 59

1 Introduction

This section outlines the purpose, scope and the overview of the document. It also contains the details of the development team as well as the Clients for the Intelligent Lifestyle project. This section also defines the definitions and acronyms that are used throughout this document.

1.1 Purpose

This document describes the testing strategies and approach to testing that Team Deadalus will adopt to verify and validate the quality of the final product prior to its realise. This includes descriptions of test items, features to be tested, features not to be tested and Team responsibilities in conducting testing activities.

1.2 Scope

This document is a guideline for conducting testing activities. A summary of the test items, features to be tested as well as features not to be tested are listed in section 3.

Beside from this document, readers may also find project documentation listed below useful in terms of gaining a better understanding of the project:

- Software Requirements Specification (SRS)
- Software Architectural Design Document (SADD)
- Software Design Document (SDD)
- Software Project Management Plan (SPMP)
- Software Verification and Validation Plan (SVVP)

1.3 Test Plan Objectives

The objectives of the the Test Plan include:

1. Defining the activities required to prepare for and conduct unit, integration, system, acceptance, usability, and performance testing.
2. Communication to all responsible parties the Test strategy employed.
3. Define deliverables and responsible parties.

1.4 Intended Audience

The intended audience for this document is Team Daedalus and its supervisor. This document may also be used for a reference in future projects.

1.5 Project Overview

The Intelligent LifeStyle project aims to create a demonstration of the concepts of intelligent agents in a context adaptive environment. The Home Intrusion scenario will be used as an application for the demonstration.

There are three main parts to the project:

1. ROADMAP methodology - the analysis of requirements and the design will need to follow this methodology for the purpose of validating it.
2. Physical Interactive Demonstration - this will involve the visual simulation and integration of several technologies including PDA's, face detection and tracking and speech recognition. This presentation will go through the scripted Home Intrusion Scenario.
3. Visual Multimedia Presentation - this will show how ROADMAP has been applied to the design of the Home Intrusion scenario and also the visual simulation to demonstrate the working system without having to set up all the equipment.

1.6 Personnel

This section specifies the personnel associated with this project.

1.6.1 Development Team

The development team of this project is Team Daedalus, enrolled in the subject of Advanced Software Engineering Project.

Name	**Login**[1]	**Phone no.**
Carol Poon	cyspoon	0401-959-660
Dominic Mendonca	dxm	0411-093-253
Glenn Fry	gmfr	0418-372-176
Simon Youn	hjsy	0403-438-830
Jian Alan Huang	jhua	0402-001-910
Kieran Simpson	kieranjs	0412-821-128
Masyuri Tjhandana	masyurit	0413-150-311
Mei Ling Leong	mlleong	0413-689-314
Nathaporn Eiamvittayakorn	neiam	0407-565-824
Quyen Quach	qlq	0412-122-031
Shirley Soon	sasoon	0407-552-338
Wendy Tsang	wwttsang	0411-199-822
Ivan Wong	ywong	0411-863-261

Table 1: Development Team

[1] Email addresses of team members can be derived from the user's login name by appending `@students.cs.mu.oz.au`.

1.6.2 Supervisor

Kendall Lister
Department of Computer Science and Software Engineering
University of Melbourne
krl@cs.mu.oz.au

1.6.3 Clients

Leon Sterling
Department of Computer Science and Software Engineering
University of Melbourne
leon@cs.mu.oz.au

Thomas Juan
Department of Computer Science and Software Engineering
University of Melbourne
tlj@cs.mu.oz.au

1.7 Definitions and Acronyms

This section defines all definitions and acronyms used by Team Daedalus in this document.

1.7.1 Definitions

$GROUP
The shared space allocated to Team Deadalus by the Computer Science and Software Engienering department. It is used to store the CVS repository, the website and other resources relevant to Team Deadalus (see SCMP). It resides at /home/se440/s440gf/

$GROUPCVS
This is the CVS repository of Team Daedalus which is located in the group directory. It resides at /home/se440/s440gf/Respository

$GROUPWWW
This is the website for Team Deadalus. It is found at http://www.cs.mu.oz.au/SE-projects/s440gf

Agent
Pieces of code in an system that can detect changes in its environment.

Black Box Testing
Black-box tests are derived from the functional design specifications without considering any details of implementation.

Boundary Value Analysis Testing
Testing technique that focuses on the functions or methods that interact with numerical values. This testing will test those critical values, (non) allowable values, and others.

Branch Coverage Testing

This testing technique identify branches in the code, and ensure that each path is taken to be executed and tested.

Conditional Coverage Testing
Testing technique that test possible combinations in the conditions, prior to branching.

Coverage Testing
Part of White Box Testing technique that ensure the coverage of the source code. Full Coverage simply means that all lines of the source code must be tested, by executing it. Other technique included in the Coverage Testing are Branch Coverage and Conditional Coverage Testing.

Document Manager
A member of Team Deadalus soly responsible for maintaining and managing a particular document. There is exactly one of these per document.

Functional Testing
Testing activities that aims to test the function of the system against its specification.

Grey Box Testing
Testing technique that combines both Black Box and White Box Testing.

NetOffice
The online system used to allocate and manage tasks. It forms the basis of the micro-planning for the project.

Test Suite
A collection of testcases designated for testing specified.

Validation
This is also called testing. It is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

Verification
The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

White Box Testing
White-box tests are derived from the code and require knowledge of the internal program structure.

1.7.2 Acronyms

This section defines the expansions of acrynyms used within this document.

EPS	Encapsulated Postscript
GUI	Graphic User Interface
PDF	Portable Document Format
PM	Project Manager
PP	Project Plan
QA	Quality Assurance
RMP	Risk Management Plan
ROADMAP	
SADD	Software Architecture Design Document
SCMP	Software Configuration Management Plan
SDD	Software Design Document
SDLC	Software Development Life Cycle
SPMP	Software Project Management Plan
SQAP	Software Quality Assurance Plan
SRS	Software Requirements Specifications
SVVP	Software Verification Validation Plan
TM	Traceability Matrix
TP	Test Plan
TUAL	Technologies Useful in Agent Lablets
UD	User Documentation
V&V	Verification and Validation

1.8 References

1.8.1 Standards

- IEEE Standard for Software Test Documentation STd 829-1998

1.8.2 Project documentation

Please refer to the Team website for the following relevant documents:

1. Gantt charts outlining major project milestones and macro-planning
2. NetOffice for task tracking information which contains micro-planning details and audit trails.
3. Team Deadalus Risk Management Plan (RMP)
4. Team Daedalus Software Architecture Design Document (SADD)
5. Team Daedalus Software Configuration Management Plan (SCMP)
6. Team Daedalus Software Detailed Design (SDD)
7. Team Daedalus Software Project Management Document (SPMP)
8. Team Daedalus Software Quality Assurance Plan (SQAP)
9. Team Daedalus Software Requirements Specification (SRS)
10. Team Daedalus Software Verification and Validation Plan (SVVP)
11. Team Daedalus Traceability Matrix (TM)

1.8.3 Online References

- *Ant* http://ant.apache.org
- *Jade* http://jade.cselt.it/
- *Junit* http://junit.org
- *NUnit* http://www.nunit.org/
- *Testing Artefacts* http://www.cuj.com/documents/s=8027/cuj0104stanley/

1.8.4 Other References

- 433-342 Lecture Notes
- Vliet, Hans Van, *Software Engineering: Principles and Practice 2nd Edition*, Wiley and Sons Ltd 2000
- Lewis, Willaim E., *Software Testing and Continuous Quality Improvement*

2 Project Description

This section describes further in the project description, which will give a general insight on how the testing could be conducted throughout our project.

The aim of the final system will be to provide a demonstration of intelligent agents. These agents will be shown acting in scenarios to solve a specific problem. Through these scenarios, their 'intelligence' and interactions can be shown.

The project will provide a demonstration of intelligent applications in a home environment. In this context, the word 'intelligence' means that as the environment is constantly changing, the system will have to detect those changes. The system will act accordingly in a human-like manner on behalf of the users.

The system will be analysed (feasibility analysis), and designed using the ROADMAP methodology as this is a requirement of the Clients. For more information on ROADMAP, refer to the SRS Document section 5.

2.1 Project Scope

It would be beyond the scope of the team to implement a fully functional system of intelligent agents to satisfy the purpose of demonstrating intelligent agents.

The phrase 'some intelligent agents' allows the team to implement the system in such way as to simulate intelligence and this will be demonstrated in the home intruder scenario (Refer to SRS Document section 6.2).

2.2 Project Process Model

The process model Team Daedalus adopts consists of essentially three interations throughout the lifecycle of the project. The three iterations are as follows:

1. **Iteration 1** - Prototyping - the main focus will be on requirements gathering and feasibility analysis. The aim of this iteration is to elicit a firm set of the Clients' requirements.
2. **Iteration 2** - During the second iteration, the overall structure of the system will be designed, with the most certain functionalities implemented and tested.
3. **Interation 3** - During the third iteration, the remining system will be further designed, implemented, tested and finally made available for release.

For further details, refer to the SPMP Document section 5.1 .

3 Testing Objectives

This section describes the test items, features to be tested, features not to be tested and their associated risks. For more information, refer to Team Daedalus's Build Plan and the goals requirements as specified in the SRS.

3.1 Test Items

Team Daedalus performs testing activities for the items described in the following sections. The following documents provide the basis for defining correct operation:

- SRS
- SADD
- SDD

3.1.1 Code

The following describes the objectives of Team Daedalus for code testing:

1. To perform unit testing for at least 80 per cent of the overall classes, prioritised by units dependency and criticality.
2. To perform integration and functional testing in the order of builds and dependencies. Integration testing is performed by using drivers and stubs as necessary to substitute for the non-functional classes that are depended upon by other testable classes.
3. To perform system testing of the overall system to ensure that it mees the quality and goal requirements as stated in the SRS.

3.1.2 Classes and Modules to be Tested

This section outlines the classes to be tested. The following information is derived from Team Daedalus' Build Plan, and for details of the dependencies of modules please refer to section 8 of that document.
The classes and modules to be tested for Build 1 are as follows:

1. DB Access Class;
2. BDIAgent Class;
3. Text to Speech Module;
4. Speech to Text Module;
5. Data Type Classes.

The classes and modules to be tested for Build 2 are as follows:

1. Face Recognition Module;
2. Clarinox Bluetooth POP Module;

3. Recorder;
4. Communication Agent;
5. Scheduler Agent;
6. Context Resolver Agent.

3.1.3 User Documentation(UD)

The following describes the objectives of Team Deadalus for UD testing:

1. To ensure that the instructions in the UD are accurate and achieve the intended audience.
2. To ensure that the system works with specifications as described in the UD.

3.1.4 Features to be tested

Team Daedalus aims to perform functional testing based on the goals as specified in the SRS.

1. Detection (refer to 5.1 in SRS);
2. Identification (refer to 5.2 in SRS);
3. Greeting (refer to 5.3 in SRS);
4. Communications (refer to 5.4 in SRS);
5. Guiding (refer to 5.6 in SRS);
6. Scheduling (refer to 5.7 in SRS) , and
7. Recording (refer to 5.8 in SRS).

4 Testing Methodology

This section describes the methodology, and the testing life cycle that Team Daedalus will adopted throughout the project. Team Daedalus project life cycle will consisted of three iterations, following a spiral model. The testing activities will only take place in the last two iterations, when the system is built. Due to the project time and resource constraint, Team Daedalus will not aims to test the prototypes, which are build in the first iteration. (Refer to Build Plan for more detail on implementation planning.)

4.1 Testing Life Cycle Concepts

Team Daedalus testing activities will started in parallel with the coding activities as by that time, the detail design should be sufficient and stable for the Testing sub-team to plan for the test, evaluate th suitable testing methodologies, processes and coding the test cases. By running both activities in parallel, the plan will maximise the time and resource usage.

Figure 1 below illustrate the activities flow of the testing life cycle in the iteration two and three. However, it do not show how all these activities collating with the spiral development model, which team Daedalus followed. For more detail on the development model, please refer to SPMP section 5.

As two builds are planned for each iteration, there will be two design, implementation-redesign and testing phase for each. Testing will be done in parallel with implementation-redesign activities in order to manage time and resource usage. During the first implementation and re-designing phase of each iteration, the testing activities involve the preparation and planning of the execution of the test including testing processes, methodologies and tools adopt. Test cases are created using the detailed design information once the preparation and planning for testing has finished. As for the second implementation and re-designing phase, only some testing preparation will need to be revisited in order to improve the testing processes. More test case will also be created then.

After each testing phase, Coding sub-team will need to correct bugs found in each testing phase, re-test with the existing test case and request the Testing sub-team for more testing if more test case are required to test the bug corrected code. Both phases will finish simultaneously as passing all testing exist criteria means that there are no need to revisit bugs correction stage in implementation phase.

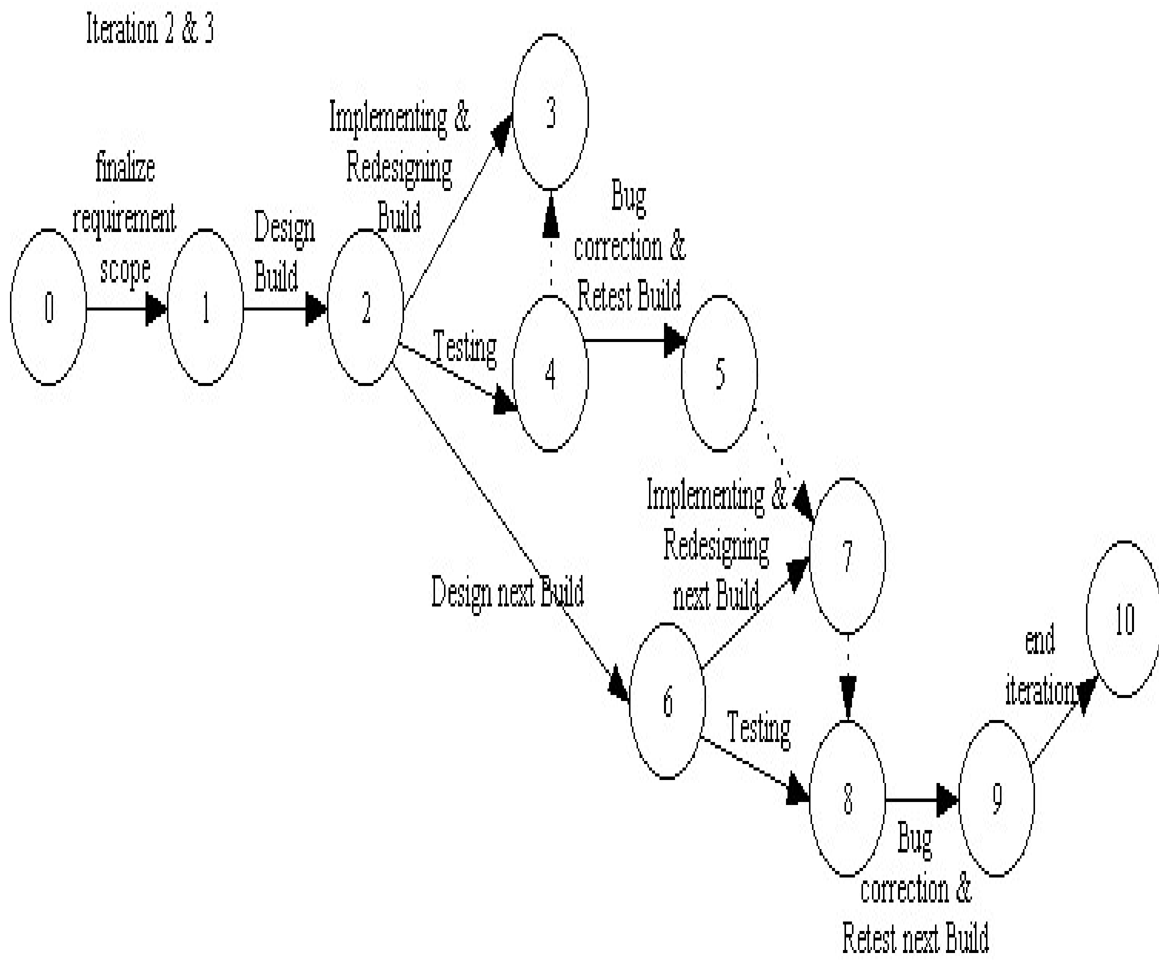

Figure 1: *Testing Lifecyle Concept*

4.2 V&V Life Cycle Model

Throughout the testing activities, Team Daedalus will follow the V-Model in order to test, verify and validate Team Daedalus system and intermidate deliverables. The V-Model is adopted because it outlines the testing to be done to verify and validate previous development activities in relation to testing and V&V activities. The model also allow parallel activities to be carry out and planned, which suite the Team's need and available time and resources. The V-Model are as shown in the diagram below:

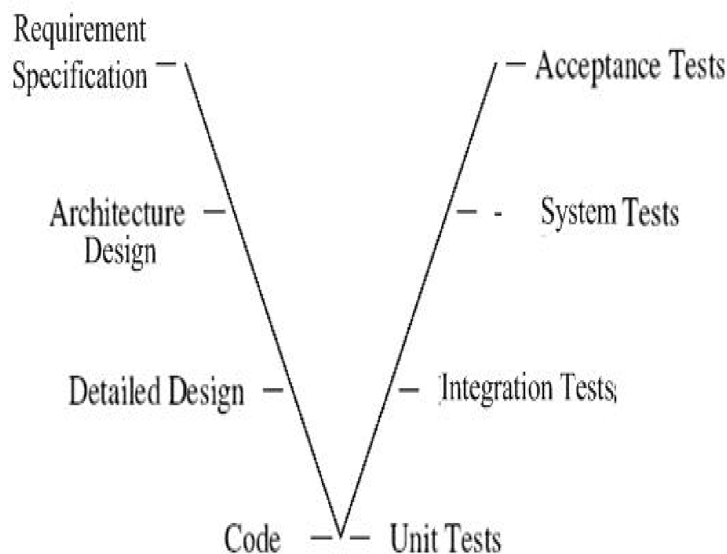

Figure 2: *V-Model*

4.3 Verification and Validation Concepts

The concepts of verification and validation activities are as described below:

1. Verification: The activities and processes that assess software products, deliverable and procedures throughout the software development life-cycle in order to verify that quality is built into the software and that the software satisfies the requirements.

2. Validation: The activities and processes that will provides supporting evidence that the software and deliverables satisfy it's requirements and solves the Client's problem.

For more information on V&V activities objectives, V-Model and planned, please refer to the SVVP section 2 and 4.1.

5 Testing Strategy

This section describes the strategy that will be implemented throughout the testing phase. It further describes on how the testing should be conducted, and how the system will be divided into modules to be unit tested, functions to be integration tested, and the system itself as a whole.

Throughout the project, the following types of testing will be used:

1. Unit Testing
2. Integration Testing
3. Agent Testing
4. System Testing
5. Reliability Modelling
6. Acceptance Testing
7. Installation Testing
8. Usability Testing

5.1 Unit Testing

A module is a particular thread that could run independently, without being dependent on another module. This modules will be tested under Unit Testing. The modules that will be included in the final system must pass a Unit Testing. Therefore, any changes made to these modules, and the changes made will be included in the final system, the particular module need to go through and pass another Unit Testing. By passing Unit Testing, it is expected that the errors will be minimised, before going to the integration stage.

Unit Testing basically will test modules independently. It doesn't necessarily mean one file, since dependencies may exist in order to run the module. In designing the test cases, every method must be tested as part of the testing (Refer to section for further details on test case design), as well as using the boundary value testing (Refer to section for further details on testing techniques). Therefore, Unit Testing will focus on both the Black Box and White Box testing. Furthermore, where the particular module has to establish a connection with the database, Database Testing also needs to be considered (Refer to section for further details on Database Testing).

5.2 Integration Testing

Several particular modules will construct a particular function of the system. For example, from SRS Document section, the goal 'TO GREET' is an example of a function to be implemented in the system. This goal will consist of several modules, and thus, will be tested as a whole (as a function). This collection of the modules will be tested under Integration Testing. Again, the functions that are going to be included in the final system must go through and pass an Integration Testing. Also, as an entry criteria of Integration Testing, each of the modules constructing the particular function needs to pass Unit Testing first. Therefore, any changed made to any of the modules, the particular modules need to go through and pass another Unit Testing, and the function is also required to

go through and pass another Integration Testing. This way, it can be ensured that every methods and functions will be tested and quality assured.

Unlike Unit Testing, Integration Testing will mainly focus on the Black Box Testing only. The reason would be that White Box Testing has been done in the unit stage, which wouldn't require another one. Whereas with the functionalities (Black Box), it needs to be tested again since integration of working modules may cause problems, due to incompatibility of different modules being integrated. In other words, Integration Testing's primary goal is to test the functionalities of particular function that is expected to be implemented in the system. It may consists several modules, which their compatibilities need to be tested.

5.3 Agent Testing

Agent Testing is similar to Integration Testing, except that Agent Testing focuses more on the Agent interactions. The reason we have Agent Testing is because of our system follows ROADMAP methodology, which allows us to design and implement the system in an 'agent-manner'. Therefore, in Agent Testing, rather than providing an input for it, we have to provide a situation where the particular agent is required to act accordingly.

5.4 System Testing

Finally, collections of functions that construct the final system, will be tested under System Testing. Basically, the System Testing should be conducted when no further changes are intended to be made. A success in the System Testing implies that the system is ready to be released, and is ready to be acceptance tested with the Clients. However, any changes that are made to the system after the System Testing, due to testing failures and any other reasons, the modules where changes occur need to go through and pass another Unit Testing. This goes also with the Integration Testing, and thus, it is required to have another System Testing.

5.5 Reliability Modelling

Reliability Modelling is conducted to ensure that the reliability goal specified in the SRS Document is specified, which is so that the system has to survive without any errors for at least 10 minutes. The Reliability Modelling will be done in 2 stages, which are in unit stages and system stages. Reliability Modelling should help the development team to see how the system is progressing through time, and when the quality of the system needs to be improved further or not.

5.6 Installation Testing

Installation Testing will also be conducted, which focuses on the installation of the system into Client's environment. The SRS Document has specified the minimum hardware requirement to run the system. One of the purposes of this testing is to ensure that with the specified minimum requirements, the system could actually be run properly. Another purpose of this testing is to ensure that the Client, or other user that does not know the system well, wouldn't have any problem with installing the system, given a user installation manual in UD Document.

5.7 Usability Testing

Lastly, Usability Testing is conducted to ensure that the system is easy to use, and do not require assistance in running or using the system. Usability Testing can improve the system in terms of its user-friendliness. Basically, the non-funcational testings (Performance, Installation and Usability Testing) are conducted when no further major changes are intended to be made.

6 Testing Techniques

This section describes all testing techniques employed by Team Daedalus to test the intermediate builds and the final system.

6.1 Grey Box Testing

Grey Box Testing is both a functional and structural testing technique that tests both the function of the system against its specification (as documented in the SRS) and the internal logic of the system (as documented in the SDD).

6.1.1 Rationale for Adopting Grey Box Testing

Grey Box Testing technique is adopted as the techniques allows the Testing sub-team the following benefits:

1. Ability to focuses on both testing the functional of the system against the requirement specification and examining logic paths of the system and thus verify what the system supposed to do accoding to the specification and verify the logical flow and structure of the design, code and the level of testing coverage.

2. Flexibility for the testers, allows them to test the system behaviour by considering its expected functions, the code and the internal structure.

3. Allowing the the testers to communicate with the Design and Coding sub-teams to understand the internal structure and design of the system.

4. Coping with the project unstable and experimental of ROADMAP Design methodologies requirement and the inexperience nature of the developers and testers.

5. Minimise the testing effort and the number of test cases to be generated on the errors with low possibility of occurance or impact.

6. Suitable for the project time and resource constraints.

6.1.2 Applying Grey Box Testing

In adopting Grey Box Testing technique, the Testing sub-team members are required to:

1. Gain design knowledge by reading the SADD and SDD prior to their test cases design.

2. Communicate with the Design and Coding sub-team to confirm any information required during the Testing phase.

3. Gain internal log knowledge by using the source code in the process of designing the test cases and selecting test cases to be written and execute.

4. Generate the test cases for each methods and functions from the knowledage of SADD, SDD and internal logic of the system.

5. The number of the test cases generate will be n, where n is the number of the methods or functions being tested. Each test if the methods or functions are operated as expected.

6. Execute the test cases.

6.2 Exception Testing

Exception Testing is the technique that focus on the exception aspect of testing, which include error messages, exception handling processes and the conditions that trigger them. Test cases are written for each error condition. Exception Testing ensure that testing cover the test for incomplete or incorrect handling of unexpected situations and the system defects.

6.2.1 Rationale for Adopting Exception Testing

Exception Testing technique is adopted as the techniques allows the Testing sub-team the following benefits:

1. Focus on the error handling processes and the error message.
2. Ensure the level tolerance limits for input variances and situations are operating as designed or appropriate actions are taken by the system to response to the exception conditions.
3. Increase the possibility in finding defects and thus improve the quality of the system.
4. Useful for the system that require the system to behave as expect, taking into account the exception cases it handles and thus enchant the testing ability to verify the correct and adaptive system requirements of the Agents behaviour in testing.

6.2.2 Applying Exception Testing

Due to available time and resource, pure Exception Testing technique will not be adopt by Team Daedalus. As exception propagate from module to another, the number of exception handling paths that are required to be test is large and the planning for exception paths coverage is difficult. Therefore, the Testing sub-team will only employ this technique during the Unit Testing phase in order to reduce the testing effort required.

Exception points (where exception might occur) will not be formally identified and numbered, as the Testing sub-team current code standard and naming already allow the traceability of test cases and test result.

In adopting Exception Testing technique, the Testing sub-team members are required to:

1. Design test case considering error handling (exception conditions) as well as testing for normal conditions cases, during Unit Testing.
2. Generate the test cases to handle each exception arised from the internal logic of the code.
3. The number of the test cases being generate will be n, where n is the number of exception points occurs.
4. Stop generating the test cases once 80
5. Execute test cases.

6.3 Database Testing

Database Testing techniques are used to test a system with the database application. It determines how well a database meets the system requirements, identifies query response time, data integrity, data validity and data recovery.
The Intelligent Lifestyle project will use a database to store all the knowledge required by the agents to achieve their goals.

6.3.1 Types of Database

Following are the databases that required to be set up by Team Daedalus:

1. The **Production Database** - Life data. (No testing are required on this database.)
2. A **Populated Development Database** - Database used by coders and testers to run the application and test with realistic amounts of data.
3. A **Deployment Database**, or **Integration Database** - Database where the tests are run prior to deployment in order to ensure that any development database changes are applied.

6.3.2 Rationale of Database Testing

Database Testing technique is adopted as the techniques allows the Testing sub-team the following benefits:

1. Allow the knowledge required, which are stored in a database to be tested.
2. Increase the testing confidence that the knowledge, which is stored in the database will contains the information required in order to demonstrate the final system functionalities.
3. Focuses on actions perform to the database are done according to specification and design and allow the database to be tested directly, by checking whether the system could connect to the database, storing and retrieving required information.
4. Ensuring that data storing and retrieval are implemented as designed, by testing the validity of values storage and retrieval.
5. Verify the process of managing the database as efficient and fulfill the final system requirements.
6. Maximise the number of bugs found and errors discover in the early stage of testing, which will minismise the number of bugs and errors found in long term.

6.3.3 Applying Database Testing

As MySQL is the most popular Open Source SQL database management system and well supported, the stability of MySQL will not be tested.

In adopting Database Testing technique, the Testing sub-team members are required to:

1. Ensure that the following databases are set up for the purposes of testing:

- Populated Development Database: prior to an attempt to perform Database testing.
- Deployment Database: prior to an attempt to perform Integration of Database testing.

2. Ensuring that the specified database is being used for testing.

3. Perform testing on a Populated Development MySQL Database.

4. Test the database on the aspects as specified in section 6.3.3.1

6.3.3.1 Things to be tested

In order to perform Database Testing, the following things will need to be tested:

1. Connecting to and Disconnecting from the Server

2. Entering Queries

3. Creating and Using a Database

   (a) Creating and Selecting a Database
   (b) Creating a Table
   (c) Loading Data into a Table
   (d) Retrieving Information from a Table
      i. Selecting all data
      ii. Selecting Particular Rows
      iii. Selecting Particular Columns
      iv. Sorting Rows
      v. Date calculations
      vi. Working with NULL values
      vii. Pattern matching
      viii. Counting rows
      ix. Using More Than one Table

6.4 Boundary Value Analysis Testing

Boundary Value Analaysis Testing is a test data selection technique in which values chosen lie along data domain's extremes. These boundary values include maximum, minumum, norm and under the minimum and maximum boundary ranges values, thus there are 4n + 1 test case for n number of variables to be tested. Not all of the variables input space in each modules are applicable to be tested using this technique and only numerical values can be tested.

6.4.1 Rationale of Adopting Boundary Value Analysis Testing

Boundary Value Analysis Testing technique is adopted as the techniques allows the Testing sub-team the following benefits:

1. Focus on the boundary of the Input space to identify test cases, asssuming that error tend to occur near boundary.

2. Ensuring that the modules handle the values with in the allowed range as expected.
3. Consider most of the input space values during the test and increase the testers confidence.
4. Ensure that testing also consider exception cases and error handling, which caused by any out of range input values.
5. Required the least effort comparing with all other Functional Testing methodologies.

6.4.2 Applying Boundary Value Analaysis Testing

In adopting Boundary Value Analysis Testing technique, the Testing sub-team members are required to:

1. During Unit testing, identify the methods that take numerical values input(s) in each tested module.
2. During Integration and System testing, identify functions that take numerical values as input(s).
3. Apply Boundary Value Analysis to the methods and functions identified by:
    (a) identify the each input domain range;
    (b) generate the test cases that take boundary values as input(s) and compare the expected output with the actual output;
    (c) The number of test cases being generate will be 4n + 1, where n is the number of variables inputs.
    (d) executing the test case.

For example, if a function only takes an input of values between 4 and 9 (inclusive), the values that need to be considered are 3, 4, 9 and 10 (given that it only takes integer values). If a funciton takes floating numbers, values 3.9, 3.999, 9.001 will be tested.

6.5 Coverage Testing

Coverage Testing is testing a program by examining which lines or parts of executable code are visited, and aims to execute each lines or parts of the code. The Coverage Testing can find more bugs in a system than other techniques, especially those that are caused by data coupling. Therefore, a success in Coverage Testing will give the testers more confidence that the system is defect-free.

6.5.1 Types of Coverage Testing

There are four types of Coverage Testing as described as follow:

1. **Full Coverage Testing**
   Focuses on examining each line of the code and aims to execute and tested all lines. It tests all of the variables used, memory allocations and arithmatic operations.
2. **Branch Coverage Testing**
   Focuses on examining each branch of the code and aims to execute and tested all branchs. For example, for an IF statement, both true and false branchs of the statement will need to be tested.

3. **Conditional Coverage Testing**
   Focuses on the every possible conditions that a branch can have. If there are two simutaneous conditions in a branch (i.e. IF a AND b), there will be four test cases that are created to test all possible conditions.

4. **Path Coverage Testing**
   Focuses on loops present in the code. If every loops are identical then it may not be necessary to test more loops. However, depending on the code, more loops may yield different result, in fact, errors may be produced as the outcome of it and thus may need to be tested.

6.5.2 Rationale of Adopting Coverage Testing

Coverage Testing technique is adopted as the techniques allows the Testing sub-team the following benefits:

1. All lines of code are tested and thus, increase more confident upon the result of the test.
2. Branchings and all conditions possibilities are taken into account.
3. Ensure that no lines of code are unused.

6.5.3 Applying Coverage Testing

In applying Coverage Testing technique, the Testing sub-team members are required to:

1. Identify any loops and branches with all possible conditions each take.
2. For each of the branches and loops, create a test cases with all possible combinations of the conditions.
3. For each of the loops, examine the content of the loop, and decide whether it is necessary to test more loops. Usually, testers will required to write test cases that test one, two, and more than four loops.
4. Identify the rest of the lines of the codes outside branches and loops, to ensure that they all get executed.

7 Unit Testing

The following sections describe the general strategies, entry and exit criteria for Unit Testing. The test cases and report produced, pass/fail criteria, as well as the procedures for conducting unit testing activities will also be described here.

7.1 General Strategy

Unit Testing involves testing the smallest possible modules within a particular functionality of the system. A module may consist of several functions. Any changes made to the particular module will lead to further unit testing to be applied. The main purpose of the unit testing is to test the modules independently, and detect possible errors as early as possible, before accumulating into more serious ones.

Unit testing will be used as the first stage of testing activities. The purpose of unit testing is to verify that individual modules are working correctly on their own before integrated into the system, by passing all of the unit test cases provided. Each unit will be provided with a unit test suite which contains several unit test cases (Refer to section 16 for more information on the test cases design). Generally, all of the Java modules are going to be tested with the J-Unit testing tools, while the C++ modules will be tested with CppUnit. Refer to section 18 for more details on testing tools.

7.2 Entry Criteria

The entry criteria for unit testing are as follow:

1. Unit testing has been requested by coders, as specified in the procedures.
2. Source code of a module requested is available.
3. Unit test cases for the particular module requested to be tested are created completely.
4. The module requested to be tested needs to have no compilation errors.
5. The module requested needs to have a version number attached to it.

7.3 Exit Criteria

Unit Testing report must be produced at the conclusion of the Unit Testing activities (refer to section 7.5). Also, an update must be performed on the Testing Management Tools (refer to section 15).

7.4 Pass/Fail Criteria

The pass/fail criteria for unit testing are as follow:

1. A module has passed Unit Testing once it has passed all of the test cases per each unit;
2. A test case is considered to have passed unit testing if the actual output is equal to the expected output, for a given input.

7.5 Test Cases and Reports

This section describes the test cases and reports used in Unit Testing. The creation of the test cases need to follow the specified test cases design, as further described in section 16. For both Java and C++ modules, a template for the test cases are provided in section A.3 and section repectedly.

The test report will be automatically generated by the testing tools used, by executing the script 'unit' in conducting Unit Testing (refer to section 7.6.2). The test reports will be available in $GROUPCVS/Test/Unit/ .

7.6 Procedures

This section describes the procedures on requesting and conducting Unit Testing. However, not every Unit Testing requires requesting process. The coders will only be required to request if one of the following conditions are met:

1. The module to be unit tested hasn't undergone Unit Testing before.
2. The existing test cases for the particular module need to be updated, due to changes made to the module.

If the Unit Testing process doesn't require any request, it implies that the coders will conduct the Unit Testing, and act as the 'tester'. Otherwise, where requesting is required, the Testing sub-team will be responsible for conducting the Unit Testing.

7.6.1 Test Request

The procedures for requesting Unit Testing are as follow:

1. Coders are responsible for initiating the request for Unit Testing via e-mail only. They should follow the template provided, under $GROUPCVS/Template/template_test_request.txt.
2. The Testing Manager will response to the e-mail within 24 hours. The Testing Manager also needs to confirm that there are sufficient information (accordingly to the template followed) given by the coders upon Unit Testing request. Included in the response, the Testing Manager will specify the estimated duration of the Unit Testing for the requested module. Negotiation between the Testing Manager and the coders may take place, when necessary.
3. After accepting the request for Unit Testing, the Testing Manager is responsible for ensuring that the Unit Testing is being done within the agreed duration, by ensuring that the test cases are generated and assigning the task of conducting the Unit Testing to testing sub-team members.

7.6.2 Conduct Testing

This section describes the procedures that are to be followed by testers in conducting the Unit Testing. As previously mentioned, the testers could be the coders themselves, or the Testing sub-team. The procedures are as follow:

1. The tester has to check whether the entry criteria for the requested module has been met or not. If not, while it's not the coders that conduct the Unit Testing, the tester is required to send an e-mail with the following information:

- Tag : [440 Code]
- Subject : Unit Testing Failure
- Body : Need to have the following information:
  - Module requested
  - List of the entry criteria not met

Upon not meeting the entry criteria, the Unit Testing concludes here, and another request is required (if applicable).

2. Otherwise, given that the test cases are ready to be used, the tester will perform the Unit Testing (refer to section 7.6.3.
3. After running the Unit Testing, the tester needs to update the Testing Management Tools on the web. Refer to section 15 for more details.
4. When the testers are not the coders, an email then needs to be sent to the coder, with the template provided under $GROUPCVS/Template/template_test_report.txt.

7.6.3 Execute Unit Test Cases

This section describes the way in executing unit test cases, given that the entry criteria of Unit Testing has been met. In conducting Unit Testing, the script 'unit' will be used. Therefore, the following steps need to be performed in preparing the script:

1. Copy the 'unit' script into the $GROUPCVS/Test/ directory. The 'unit' script is available from $GROUPCVS/Script/.
2. Edit the script, and change ¡username¿ to the username of the tester.

The 'unit' script performs the following operations:

- Run the Unit Testing, which will produce the test report.
- Commit the report generated into CVS.
- Publish the report to the group website.

After having the 'unit' script ready, the tester needs to run the command:

unit *classname directory*

where the directory is the sub-directory of $GROUPCVS/Code/ where the class is located.

8 Integration Testing

This section describes the general strategy, entry/exit criteria, pass/fail criteria, artefacts, as well as the procedures for the Integration Testing

8.1 General Strategy

Integration Testing focuses on the functions that the final system should have (Refer to SRS Document section for the goals/functionalities expected from the system implemented). This should consist of a collection of modules, within the same function that have been unit tested previously. Any changes to any of the modules will require another Integration Testing. The changes meant here include both the changes made to correct errors, and changes to improve the modules. However, beside Integration Testing, these changes may also require Unit Testing before continue on the Integration Testing (Refer to section 8.3). Integration Testing mainly serves to ensure conformity of each of the functions to the requirements.

2 different methods will be used in Integration Testing. They are as follow:

1. Human Observation
2. Automated Generated Report

With Human Observation method, Integration Testing will involve manual observation, either in the output assertion, or in the input provision. However, a test report still needs to be produced as the result of Integration Testing. On the other hand, Automated Generated Report will do the Integration Testing automatically, given the test cases have been created. The test report will also be generated by the testing framework.

8.2 Entry Criteria

The entry criteria for Integration Testing are as follow:

1. The collection of modules that form a function to be tested are provided.
2. The test cases for the function to be tested have been produced.
3. The relevant collection of modules under that function must have passed the Unit Testing (the version that the modules have for Integration Testing need to be the same version as those that passed the Unit Testing).
4. The main code to be integration tested needs to have no compilation error.

8.3 Exit Criteria

This section describes the exit criteria for Integration Testing, which are as follow:

1. Integration Testing report

8.4 Pass/Fail Criteria

The pass/fail criteria for Integration Testing are as follow:

1. An Integration Testing is considered to be passed only if all of the integration test cases are passed for the items/modules that underwent integration;

2. A test case is considered to be passed if an given input produces an actual output that is the same as the expected output.

3. Subsequently, it fails if it doesn't pass, at least, one of the integration test cases.

8.5 Test Cases and Reports

The Integration Test Reports need to follow the template provided in section A.5 and A.6.

8.5.1 Naming Conventions

The naming convention for the test report is as follow:

TestReport*integration title*.txt

The 'integration title' is the short name of the integration description that the report describes. This title will be determined by the testers themselves.

8.6 Procedures

This section describes the procedures that need to be followed in order to initiate and conduct the Integration Testing. They are as follow:

1. The Testing Leader will assign tasks to the testers on the Integration Testing.

2. The tester will then confirm whether the entry criteria to conduct Integration Testing has been met or not. If not, the tester needs to inform the Testing Leader for further conduction.

3. Upon meeting the entry criteria, the tester will need to determine which method to be used. Thus, the tester could start creating the test cases for it.

4. A test report needs to be produced at the end of Integration Testing. Refer to section 8.5 for more details on integration test report.

5. Finally, the testee needs to send an e-mail, which consists of:

   - Tag : [440 Code][440 Test]
   - Subject : Integration Testing Completion - ¡name of the Function¿
   - Body – the following information need to be included:
     - The filenames of the modules tested
     - The Integration Testing report name and location
     - Result of the test (Pass/Fail)

9 System Testing

This section describes the general strategy, entry/exit criteria, pass/fail criteria, artefacts, as well as the procedures to conduct a System Testing. System Testing will be performed by the team as part of the testing activities.

9.1 General Strategy

The System Testing will focus on the functions of the product, as a whole entity. It aims to ensure that the product is validated against the clients' requirements. Therefore a success in system testing will mean that the product has conformed to all of the requirements stated in SRS. However, to perform System Testing, integration testing needs to be applied to all of the functions separately.

9.2 Entry Criteria

The entry criteria for System Testing are as follow:

1. All modules have gone through, and pass the System Testing and Integration Testing.
2. The versions of all of the modules requested for System Testing need to have the same version with those that underwent the System and Integration Testing.
3. System test cases have all been produced.
4. System can be started up without any compilation errors.

9.3 Exit Criteria

This section describes the exit criteria for System Testing, which are as follow:

1. System Testing report
2. Bug report

9.4 Pass/Fail Criteria

The pass/fail criteria for System Testing are as follow:

1. System Testing is considered to be passed only if all of the system test cases have been passed;
2. A test case is considered to be passed if an given input produces an actual output that is the same as the expected output.
3. The system fails the System Testing if it doesn't pass, at least, one of the system test cases.

9.5 Test Cases and Reports

The system test case and report are provided under
$GROUPCVS/Test/System/system_test_case.txt

9.6 Procedures

This section describes the procedures that are to be followed by testers in conducting the System Testing, as a task assigned by the Testing Manager. They are as follow:

1. The Testing Leader needs to ensure that the entry criteria for System Testing has been met and thus, assign the task to the relevant tester.

2. The tester will then take the system test cases provided and execute the System Testing.

3. At the conclusion of the System Testing, the tester needs to produce a testing report (refer to section 9.5). The report has to be made available, at the latest, at the end of the date of the System Testing task.

4. Finally, the tester needs to send an e-mail, which consists of:

   - Tag : [440 Test][440 Code]
   - Subject : System Testing Completion
   - Body – the following information need to be included:
     - The System Testing report name and location
     - Result of the test (Pass/Fail)

10 Agent Testing

This section describes the general strategy for Agent Testing, as well as its entry/exit criteria, procedures, and the artifacts produced.

10.1 General Strategy

Agent Testing focuses on the behaviour of a particular agent. Therefore, Agent Testing would be similar to Black-Box Testing. It won't be necessary for the tester to examine the source code, as they will only be required to test its expected functionalities. Instead, the tester will look at the design document (SADD and/or SDD Document).

For each of the agent tested, a state diagram needs to be derived. Based on these states, the test cases will be created. (Refer to section 16.4 for further details on creating agent test cases). A state represents the different behaviour a particular agent might have. Therefore, by testing each of the states, Agent Testing is testing each behaviour of the agent could have, and thus, ensuring that they 'behave' as expected.

10.2 Entry Criteria

The entry criteria in conducting Agent Testing are as follow:

1. The particular agent test cases have been made available.
2. No compilation error for the agent to be tested.

10.3 Exit Criteria

The exit criteria for Agent Testing are as follow:

1. A test report has been made available.

10.4 Pass/Fail Criteria

The pass/fail criteria for Agent Testing are as follow:

1. An agent has passed Agent Testing once it has passed all of the test cases per each agent;
2. A test case is considered to have been passed if the actual output is equal to the expected output, for a given input.

10.5 Test Cases and Reports

For the test cases and test reports, there are templates provided in the appendix section of this Document. The template needs to be followed, and change the attributes accordingly. Refer to section A.7 for test cases template and section A.8 for test report template.

10.5.1 Naming Conventions

The naming convention for agent test cases is:

Test *agent name index*.java

The agent name is the name of the agent to be tested, while the index is the number of the test cases available, start from 1.

The naming convention for agent test report is:

TestReport *agent name*.java

The agent name is the name of the agent tested.

10.6 Procedures

This section describes the procedures in conducting Agent Testing. Agent Testing is initiated once an agent has been finished coded, and thus, ready to be tested. The Testing Sub-Team Leader will assign the task for Agent Testing accordingly.

10.6.1 Conduct Testing

This section describes the procedures to be followed in the actual conduction of Agent Testing. They are as follow:

1. The tester assigned needs to run the Agent Testing, by following the description under section 10.6.2.
2. After executing the test cases, the tester needs to create the test report, by copying the template from the appendix section. The tester will then need to copy the result from the Agent Testing and insert it into the template. The test report will then needs to be committed into CVS repository.

10.6.2 Execute Agent Test Cases

This section explains on how to execute Agent Testing, with given test cases.

1. You need to be under $CVSGROUP/Test/
2. Run 'ant testagent'
3. A java GUI will come up. Then an agent container in the smaller window, by expanding the tree on the left side, and right click on the 'Container'.
4. Click on 'Start new agent'. A window will come up.
5. Fill in the top field with any name that you want to name the agent/ Fill the second field with the class package that you want to test. Click OK.
6. On the bigger window, click 'Open', and select the agent that you want to test.
7. Click 'Run'. It will then execute your test cases against the agent.

11 Acceptance Testing

This section describes the Acceptance Testing strategy, including the entry/exit criteria, pass/fail criteria, and the procedures in conducting the testing.

11.1 General Strategy

Acceptance testing is performed with the Clients to demonstrate to them that the system has met its requirements as specified in the acceptance criteria of the SRS.

11.2 Entry Criteria

The entry criteria for acceptance testing are as follows:

1. The system must have passed System Testing.

11.3 Exit Criteria

The exit criteria for Acceptance Testing are as follow:

1. An acceptance statement has been signed by both the Clients and the Project Manager and Requirements Manager.
2. A test report (according to the acceptance criteria)

11.4 Pass/Fail Criteria

The pass/fail criteria for acceptance testing are as follows:

1. If the Clients happily accept the system at the time of delivery, acceptance testing is considered to have passed.

11.5 Test Cases and Reports

For acceptance testing, a checklist will be used to perform such type of testing:

Date:	
Clients Name:	
Acceptance Test Run #:	
Checklist	
Item Number	**Results** (Check box if checklist items are satisfied)
1. insert checklist item 1	insert result
2. insert checklist item 2	insert result
Total # items satisfied:	

Table 2: Acceptance Testing Checklist Specification

11.6 Acceptance Testing Procedures

The following procedures must be used in acceptance testing:

11.6.1 Creating Acceptance Tests

1. The testing sub-team will allocate a person (or persons) to write the acceptance tests. This can be done after the system test cases.
2. The author(s) must decide what the aims of the acceptance tests are.
3. The author(s) must refer to the SRS, and determine which requirement(s) each acceptance test corresponds to, and also determine the expected response/output of the system for each step of the scenario.

11.6.2 Pre-acceptance Testing Activities

1. Once the Coding Leader deems the system is complete, the Testing Manager or the Coding Leader will allocate a testing team member to run through the acceptance test cases to ensure that they pass. This would occurr at the same time as system testing and this is to reduce the probability of a failure in front of the Clients.
2. Once the tester has performed the tests, the outcome of each acceptance test must be sent to the Coding Leader and testing via email.
3. Once all acceptance tests for the system have been checked by a testing team member, they can be performed with the Clients.

4. The Requirements Manager negotiates with the Clients a suitable time for acceptance testing and product delivery.

11.6.3 Performing Acceptance Testing

1. The Testing Manager or Project Manager will allocate team members to be the guides for the Clients. A minute taker will also have to be nominated.
2. At least one of the Clients, the Project Manager and a representative from the Requirements sub-team, Coding sub-team, and Testing sub-team must be present at the Acceptance Testing event.
3. The Clients will be provided with the acceptance criteria to be checked against.
4. As directed by the guides, the Clients will follow through the acceptance test cases developed and the acceptance criteria.
5. THe Clients will evaluate the product and assess it against every acceptance criteria as outlined in the SRS.
6. At each stage of the scenario, the Clients will be asked for their opinion on whether they are satisfied by the response/output of the system developed.
7. If the product is accepted, the Clients and the Project Manager will sign-off the product.
8. If the product is rejected, then the Project Manager would discuss dissatisfaction and possible extension with the Clients. The Project Manager would also discuss the problems with the Team to work out what can be done.

11.6.4 Post-acceptance Testing Activites

1. An acceptance report must be created by the minute taker within 48 hours of performing acceptance testing. This report will follow the minutes template.
2. Once this has been created, it must be sent to the Coding sub-team, Testing sub-team and Clients (through the Client Liaison Officer).

12 Reliability Modelling

This section describes the general strategy, entry/exit criteria, pass/fail criteria, artefacts,and the procedures for Reliability Modelling.

12.1 General Strategy

Reliability Modelling focuses on the reliability goal defined in the SRS Document. This goals states that the system needs to run with no errors throughout the final demonstration. The duration of the demonstration will be 10 minutes. Therefore, the system needs consistently run error-free for at least 10 minutes.

Reliability Modelling will test whether the system is capable of running without any errors for at least 10 minutes. Thus, the first error encountered in the final build will need to occur after 10 minutes.

Due to the fact that there is more than one method of measuring time we will be conducting reliability testing using two techniques:

1. An automated test suite; and
2. physically running the system for 10 minutes.

Using an automated test suite we are able to quickly supply input to the system. This allows us to very quickly and efficiently gather information on when errors occur in relation to system execution time. The problem with measuring the system execution time is that it does not accurately reflect the time the system will be run for. It measures the amount of time the system is processing information. This is why we need to physically run the system for 10 minutes.

By having a user run the system for 10 minutes we can compare the amount of system time used in that 10 minutes. Once we know how system time is related to actual time, we can more accurately track when errors occur in the system. In this way the data gathered through automated testing can be used to ensure the reliability of the system satisfactorily meets the reliability goal specified in the SRS.

12.2 Entry Criteria

The entry criteria for Reliability Modelling is as follows:

1. The system must be completely coded and integrated.
2. There are no compilation errors when compiling the system.
3. Test cases for Reliability Modelling have been provided.

12.3 Exit Criteria

This section describes the exit criteria for Reliability Modelling, which is as follows:

1. Reliability Modelling Report

12.4 Pass/Fail Criteria

The pass/fail criteria for Reliability Modelling is as follows:

1. Reliability Modelling is considered to be passed only if all of the reliability test cases are passed for the system underwent testing;

2. A test case is considered to be passed if a given input produces an actual output that is the same as the expected output.

12.5 Test Cases and Reports

The Reliability Modelling Report will consist of the time where an error has occurred. The report must follow the template provided in the Appendices section at the end of this document. The result of each Reliability Modelling will be appended to this report. Thus, there'll be only one report at the end of the project.

12.6 Procedures

This section describes the procedures that need to be followed in order to conduct the Reliability Modelling. They are as follow:

1. The Testing Leader has to ensure that the test cases for Reliability Modelling has been created.

2. The Testing Leader will then assign tasks to a tester to run Reliability Modelling on the system.

3. The tester needs to confirm that the entry criteria have been met for the system to go through Reliability Modelling.

4. The tester then conducts Reliability Modelling, and a test report needs to be produced at the end of the testing.

13 Installation Testing

This section describes the Installation Testing strategy, entry/exit criteria, pass/fail criteria, as well as the procedures in conducting the testing.

13.1 General Strategy

The Installation Testing focuses on the information sufficiency in installing the system. However, the platform where the system will installed to must follow the minimum hardware requirements specified in the SRS Document section. Also, the installation instructions will be specified in the UD Document. Given these two conditions are met, any user will be able to install the system without any difficulties.

In conducting the Installation Testing, it may be best to select the member that knows the system least. Or, people outside of the team could also be a perfect volunteer. By doing this, the main goal of the Installation Testing can be met. In addition, conducting the Installation Testing doesn't necessarily mean only one person will try to install, by following the installation instructions given. Instead, more than one person may give a better indication on how well the installation procedures are documented.

Beside testing the effectiveness of the installation instructions, the Installation Testing will aim to test whether with the minimum hardware requirement specified in the SRS Document are sufficient enough to run the system implemented. This can be seen as when the installation has completed, and the system can be run.

In summary, the Installation Testing will first test the installation instructions in UD Document, and then it can proceed to test the correctness of the minimum hardware requirements specified in the SRS Document.

13.2 Entry Criteria

As an entry criteria for Installation Testing:

1. The system must have undergone and passed a System Testing. Thus, the latest version that were system tested will be taken to be used under Installation Testing.
2. The platform where the system will be installed need to have the minimum hardware requirements specified in the SRS Document.
3. The Testing Sub-Team Leader confirms with the Coding Sub-Team that no further changes are made to the code, unless required due to failure in Acceptance Testing (if haven't been done).
4. The latest and baselined version of UD Document.

13.3 Exit Criteria

The exit criteria for the Installation Testing would be the test report. Refer to section for further details on the installation test report.

13.4 Pass/Fail Criteria

The pass/fail criteria for Installation Testing are as follow:

1. The Installation Testing is considered pass if the user performing the installation, could install the system on a given platform, by following the instructions specified in UD Document.
2. Also, the system needs to be able to run properly on the platform where the system is installed.
3. If it happens that the user requires assistance in installing the system, it will be considered that the system fails the Installation Testing.
4. The system also fails if it couldn't run on the platform where it is installed.

13.5 Test Cases and Reports

For the template for Installation Testing report, please refer to section A.15.

13.6 Procedures

This section describes the procedures on conducting the Installation Testing. They are as follow:

1. The Testing Manager will assign one of the team member to act as the user in installing the system. Otherwise, a person outside of the team may also be chosen.
2. The person assigned needs to reply back within 24 hours, acknowledging the task. Otherwise, the Testing Manager has to assign another person.
3. During Installation Testing, the user installing the system is not allowed to seek any assistance, nor being assisted by anyone.
4. At the conclusion of the Installation Testing, the user needs to produce a test report, following the template specified in section 13.5. The report has to be made available, at the latest, 24 hours after the Installation Testing. Refer to section for the location on where the report needs to be stored.
5. Also, when the test report has been made available, the user is required to send an e-mail to notify the Coding, Testing and Requirement Sub-Team upon the result of the testing. The e-mail should have the following information:
    - Tag : [440 Code][440 Test][440 Requirement]
    - Subject : Installation Testing Result
    - Body : Must specify the result of the testing.

14 Usability Testing

This section describes the Usability Testing strategy, entry/exit criteria, pass/fail criteria, as well as the procedures in conducting the testing.

14.1 General Strategy

Team Daedalus will adopt Heuristic Evaluations, Cognitive Walkthrough and using Questionaire as part of the general strategy for Usability Testing. Since the system and the final demonstration needs to be user friendly and hence accentuates the concept of an 'intelligent lifestyle', it is important to understand whether the user will have any difficulty in using the different devices in the system. Thus due to this it is decided that Heuristic Evaluations and Cognitive Walkthrough will be used due to the simplicity of these techniques and yet they yield the most straightforward and useful information needed to evaluate the usability of the system, demonstration and the animated presentation.
There are three major parts of the project that need to undergo seperate usability analysis using different techniques. These are as follows:

1. *Human - Device Interaction*: This category tests the points of interactions between the user and the system to be demonstrated. These points of interactions are through a PDA, STT (Speech To Text) and TTS (Text To Speech) programs. These will be analysed using Heuristic Evaluation it will be conducted in the CSSE Department's IDEA Lab.
2. *The Guider*: The guider is also an interactive part of the system. However doing usability analysis for The Guider requires the tester to be physically guided to a specifed destination from an origin by the PDA device. Thus usability analysis of this aspect of the system will not be conducted in the CSSE Department's IDEA Lab.
3. *Animated Presentation*: This aspect of the project will be analysed by applying Cognitive Walkthrough and also using Questionaires to determining the level of understandability of the animated presentation.

14.2 Entry Criteria

The following specifies the entry criteria for Usability Testing in general:

1. The system must have undergone and passed Unit Testing and Integration Testing.
2. The UD has been produced.

The following are the entry criteria for Heuristic Evaluations for the Human-Device Interaction:

1. IDEA Lab session booked;
2. A Moderator;
3. Three Evaluators - with pen and paper for note taking.

The following are the entry criteria for Heuristic Evaluations for the Guider:

1. A map containing the origin and destination;
2. A Moderator;

3. An Evaluator.

The following are the entry criteria for Cognitive Walkthrough:

1. A completed animated presentation.
2. A general description of who the users will be and what relevant knowledge they posess.
3. An Evaulator
4. A secretary

14.3 Exit Criteria

The following specifies the exit criteria for Heuristic Evaluator:

1. Three Heursistic Evaluation Reports for the three technologies from Human-Device Interaction.
2. One Heuristic Evaluation Report for the Guider.
3. One Cognitive Walkthrough Report recording the results and comments from the Evaluator of the animated presentation.
4. One completed questionaire filled in by the tester.

14.4 Pass/Fail Criteria

The pass/fail criteria for Usability Testing are as follows:

1. The system is deemed to have passed Usability Testing if the scoring of the three outcomes of this type of testing has 80% over the middle scoring level.

14.5 Test Cases and Reports

There will be three types of reports generated for the Usability Testing of the Intelligent Lifestyle project. They are as follow:

1. Reports for the Heuristic Evaluation on the HDI component;
2. Reports for the Heuristic Evaluation on the Guider component;
3. Reports for the Cognitive Walkthrough on the animated presentation, and
4. Completed Questionaires for the animated presentation.

14.6 Procedures

Usability Testing for the Intelligent Lifestyle project include three types of procedures, which correspond to the three techniques mentioned in section 14.1.

14.6.1 Heuristic Evaluation Procedures

The following section desribed the Heuristic Evaluation procedure for analysing the Human-Device Interaction technologies:

14.6.1.1 Human-Device Interaction

1. The Testing Manager must book a session in the IDEA Lab for this usability analysis.
2. The type of personnel required for this Heuristic Evaluation are: the Testing Manager who will act as the moderator and three evaluators.
3. The Testing Manager should ensure that all the devices and programs are gathered and installed on the computer used for this analysis. Such devices are PDA, web-cam, and programs include SST, TTS which should be installed on a portable labtop to be used for the analysis. He shall also need to make sure the types of personnel required for this analysis is present.
4. On the day of the usability analysis in the IDEA Lab, coders in charge of coding of the PDA, SST, and TTS modules are required to set up the devices, programs and the cameras in the IDEA Lab. The moderator is required to set up the editing equipment in the control room in the IDEA Lab.
5. When the equipment, devices and programs are set up, the Heuristic Evaluation may begin.
6. There shall be three evaluators for this part of Heuristic Evaluation. They will each be specifically focusing on evaluating a particular part of the system. Namely these will be the PDA GUI, TTS and SST.
7. The three Evaluators will be presented with three different Heuristic Evaluation Criteria corresponding to the particular technology they're focusing on.
8. The Moderator shall start the system to be demonstrated and each evaluator shall jog down notes as required in respect to the evaluation criteria and any other comments for their particular technology.
9. The Evaluators may wish to start the system as many times as required in the IDEA Lab session to evaluate their technology. However the Evaluators are expected to only run the system three times in maximum to evaluate their technology. This is so to keep the length of the evaluation session to minimum as each IDEA Lab session is time restricted.
10. The Heuristic Evaluation concludes when the Evaluators feel that they've evaluated all aspects of the criteria relevant to their technology, or that the IDEA Lab session has come to the end.
11. Evaluators are then required to formally produce a Heuristic Evaluation Report following the template in

    `$GROUPCVS/Template/template_usability_he_report.txt`

12. They shall then check in their report following the naming convention of

    `MMDD_he_[technology]_[login].txt`

    where [login] is the login of the evalutor, into

    `$GROUPCVS/Test/Usability/Report/`

14.6.1.2 Guiding

The guiding part of the system will not be actually tested in the IDEA Lab. The location to conduct this analysis will be the predetermined origin and destination of the Guider. The following describe the procedure for this part of the usability analysis:

1. The Moderator shall start the system and invoke the Guider to be active on the PDA.
2. The Evaluator shall follow the Guider and the instructions displayed on the PDA, and analyse the technology according to the specified criteria. The Moderator can accompany the Evaluator to strictly assist in holding the device while Evaluator is noting comments etc. However, the Moderator shall not contribute to the analysis in anyway as it may generate a biased analysis.
3. The Evaluator may wish to start the Guider as many times as reuqired for the evaluation. However it is suggested that the Evaluator should only need to run the Guider three times in maximum to evaluate the technology to keep the evaluation efficient.
4. The Heuristic Evaluation for the Guider concludes when the Evaluator feel that he/she has evaluated all aspects of the criteria relevant to the technology.
5. The Evaluator is then required to formally produce a Heuristic Evaluation Report following the template in

   `$GROUPCVS/Template/template_usability_he_report.txt`

6. He/she shall then check in the report following the naming convention of

   `MMDD_he_guider_[login].txt`

   where [login] is the login of the evalutor, into

   `$GROUPCVS/Test/Usability/Report/`

14.6.2 Cognitive Walkthrough Procedures

Cognitive Walkthrough will be conducted for the animated presentation. This may or may not be conducted in the IDEA Lab. The following is the procedure corresponding to the Cognitive Walkthrough for the animated presentation.

1. The Secretary shall have set up the animated presentation on any computer/laptop.
2. The Secretary shall reinforce to the Evaluator to speak out their thoughts on the presentation out loud as they go through the presentation. If at any point in the walkthrough the Evaluator seems to stopped speaking out their thoughts out loud, it is the Secretary's job to prompt the Evaluator to speak out their thoughts. However the Secretary shall not contribute to the evalution as it may generate a biased evaluation.
3. The Evaluator shall start the animated presentation and shall pause the presentation slide by slide and comment on their thoughts on the slide in regards to the evaluation criteria specified.

4. At the end of the presentation, the Evaluator shall summarise the walkthrough by giving general comments of the presentation, what was not well presented and understood etc.

5. The Secretary shall note down all comments given by the Evaluator, he/she shall document the report formally in a Cognitive Walkthrough Report following the template in

   `$GROUPCVS/Template/template_usability_cw_report.txt`

6. He/she shall then check in the report following the naming convention of

   `MMDD_cw_[login].txt`

   where [login] is the login of the evaluator, into

   `$GROUPCVS/Test/Usability/Report/`

14.6.2.1 Questionaire

The Questionaire shall be presented to the Evaluator at the end of the Cognitive Walkthrough so that his/her comments may be quantitively measured by the Testing sub-team.

1. The Secretary shall present the Evaluator with a hardcopy of the questionaire.

2. The Evaluator shall fill in the questionaire.

3. The Secretary shall transform the completed questionaire into softcopy and store it in the following location with the naming convention of

   `MMDD_questionaire_[login].txt`

   where [login] is the login of the evalutor:

   `$GROUPCVS/Test/Usability/Report/`

15 Testing Management Tools - TMT

This section describes how the process in managing and monitoring the test cases that have been created, as well as what codes that are currently being built, or under progress.

Basically, the management is done through the website, which is located in $GROUPWWW/coding.php. The following attributes will present:

1. Module - The name of the module
2. Version - The version of the module
3. Coder - The coder(s) of the module
4. Estimated Finish Date - The estimated finish date for the module to be finish coded
5. Completed - Whether the module has actually been complete coded.
6. Testing Request - Whether the particular module, with the particular version, has been requested for Unit Testing.
7. Test Cases - Whether the test cases for the module, with the particular version, are available.
8. Unit Tested - Whether the module with the particular version has been unit tested.
9. Result - The result of the Unit Testing for the particular version of that module.
10. Comment - Further comment
11. Date Modified - Last date the details have been modified

The purpose of having this management is for the following reason:

1. To see the overall progress on the Unit Testing activities.
2. For the Testing Sub-Team to be aware of the modules created.
3. To record the history of the result of the Unit Testing.
4. To make it easier for the Integration Testing, when ensuring whether the particular modules have been unit tested or not.

15.1 Tester's Responsibilities

This section describes the responsibilities of the testers in updating the Testing Management Tools. 'Tester' is not limited to the Testing Sub-Team only, but every person that conduct testing. The followings are tester's responsibilities:

1. When the tester finish conducting the Unit Testing, the tester is required to update TMT, for the following attribute:
   - Unit Tested
   - Result

15.2 Coding Sub-Team Responsibilities

This section describes the responsibilities of every members in Coding Sub-Team in updating the Testing Management Tools. They are as follow:

1. When the coder start coding a module, that coder needs to ensure that the module has already been inserted into the TMT. If it's not in TMT, that coder is required to add the module.
2. When the coder requests a Unit Testing for particular module, that coder is responsible for updating the TMT accordingly, i.e. the 'Testing Request' attribute.
3. When the coder finished coding particular module, that coder needs to update the TMT, i.e. the 'Completed' attribute.
4. When the coder is updating any module that has been unit tested before, that coder is required to update TMT by :
   - Resetting the 'Completed', 'Test Request', 'Unit Tested', 'Result' attributes.
   - Update the version number of the module.

15.3 Testing Sub-Team Responsibilities

This section describes the responsibilities of every members in Testing Sub-Team in updating the Testing Management Tools. They are as follow:

1. Monitoring the progress of the testing activities.
2. When a test case has been finished created, the person responsible for creating that test case need to update the TMT, i.e. the 'Test Case' attribute.

16 Test Case Design

This section describes the processes on designing the test cases required for all of the testing activities.
This section describes how the test cases for each of the types of testing should be designed. It would certainly be different for the different type of testing activities, which are to be explained in the following subsections.

16.1 Unit Testing

This section describes how the unit test cases should be designed. Basically, the Unit Testing will test all modules, separately, in the system. These modules are listed as the part of SDD. The unit test cases design should be based on the SDD, in terms of the informations required, as follows:

1. Methods used
2. Number and types of arguments/outputs required for each of the methods
3. Description on what the purpose of each method

If SDD doesn't provide enough information on creating the test cases, then it may be necessary to look at the source code directly.

The test cases themselves have to cover the functional testing (Black Box) and coverage testing (White Box). Refer to section for further details on the testing techniques. It may be sufficient enough to create the functional test cases by looking at the SDD, but in order to create the coverate test cases, it may be required to look at the source code directly.

16.2 Integration Testing

This section describes how the integration test cases should be designed. Mainly, the integration test cases would be based on the SADD and SRS Document. The SADD document can be used to look at the functions implemented, while the SRS Document can be used to conform which requirement the functions implement. Therefore, the integration test cases would be designed as purely Black Box Testing. Looking at the code directly would not be necessary, as the test cases are designed based on their functionalities.

16.3 System Testing

This section describes how the system test cases should be designed. They are designed based on the requirements set out in the SRS Document.

16.4 Agent Testing

This section describes how the agent test cases should be designed. Basically, the test cases should be based on the the design for each agent tested. The requirements may also be used as the base for testing them.

16.5 Rationale in Agent Testing

In creating the test cases for Agent Testing, a state diagram for each of the agent must be derived first. This is so because the test cases will mainly be based on the state of each agent. After deriving the state diagrams, the test cases then need to be created at least one for each state. However, some state may require more than one test case, in order to cover all possibilities the agent would take as an input. The state diagrams have been put into appendices for each of the agent.

Since our agent-oriented system is mainly coded within the JADE framework, using ACL Message as a communication tools between agents, the testing framework itself will use JADE (refer to Testing Tools for more details). The input given and output obtained will be constructed in ACL Message format, which will be specified in the test cases. Also, testers need to have a 'tester agent' that serves as an agent that will communicate, by providing input and obtaining output, to the actual agent tested. Therefore, these ACL Messages will be sent and received through the 'tester agent'.

16.6 Creating Agent Test Case

Several steps need to be followed in order to create a test case for Agent Testing. They are as follow:

1. Copy the 'tester agent' from the template provided in the appendix section. Change the attributes accordingly.
2. Copy the test list file (.xml) from the template provided in the appendix section. This serves as the list of the files of the test cases to be executed in Agent Testing. Change the attributes accordingly.
3. Use the template from the template for the agent test case.

After following the above steps, the Agent Testing can now be executed. Read on the section ?? for more details on how to execute Agent Testing.

17 Language Constraints

This section list out the language constraints that testing process will need to consider. For the purpose of testing, the choice of the testing tools will also need to be able to support the choice for these languages.

The languages that Team Daedalus will use in development the demonstration system are as follows:

1. C#, which is coded in .NET framework
2. C++, which will use Visual C++
3. Java

18 Testing Tools

This section outlines the testing tools choosen by the Testing Sub-Team, the reasons for choosing them and briefly describe how to use them.

18.1 Tools

This section lists the tools that are chosen by the Testing Sub-Team. The chosen tools are as follows:

1. NUnit will be used for testing C++ and C# code.
2. JUnit will be used for testing Java code.
3. ANT will be used to automate the unit, integration and agent testing of JUnit test cases.

18.2 NUnit

NUnit is the a unit-testing framework for all .NET languages, initially ported from JUnit. It is written entirely in C# and has been completely redesigned to take advantage of many .NET language features and is the standard unit testing tools for .NET. For integration testing, NUnit can be integrated with TestRunner to accommodate integrate testing.

18.2.1 Rationale for Choosing NUnit

The reason that the Testing sub-team has chosen NUnit are:

1. support our required language framework, which are C# and C++;
2. open source and suitable for our budget scope;
3. quite easy to use;
4. widely recognised and well documented, which assist the Testing sub-team to learn how to use it;

18.2.2 Benefits of NUnit

The advantages of NUnit are:

1. Open source
2. Documentation and resource provided.
3. Provide both fast (non-interactive) and interactive GUI test runner
4. Support C++ and C#
5. Provides a variety of attributes that can be used when creating unit tests
6. Coherent and cohesive framework

18.2.3 How to Use NUnit

NUnit provides both Consort Runner and the Graphical Runner that will run the test cases. While Graphical Runner, provides a visual indication of the success or failure of the tests and selecting tests to be run, Consort Runner (text based) can be used when you want to run all your tests and do not need colour code indication of success or failure. The testers are required to:

1. create a project in NUnit and add the required NUnit reference
2. write test cases and add them to the project
3. compile the project
4. start NUnit Graphical Runner
5. execute or select dll files of compiled project and run them
6. NUnit will report back with yellow (not being run)/ green (successfully run) or red (failed) marked.

18.3 JUnit

JUnit is an instance of the xUnit architecture for unit testing frameworks. It is an open source Java testing, which allow tester to write and run repeatable tests.

18.3.1 Rationale for choosing JUnit

The reason that the Testing sub-team has chosen JUnit are:

1. support our required language framework, which are Java;
2. open source and suitable for our budget scope;
3. easy to use;
4. widely recognised and well documented, which assist the Testing sub-team to learn how to use it;
5. allow quicker test (as it can be automate with ANT) and reusable testing;
6. can customise test report to suite the need of the team;

18.3.2 Benefits of JUnit

The advantages of JUnit are:

1. Open source
2. Documentation and resource provided.
3. Provide both fast (non-interactive) and interactive GUI test runner
4. Support Java
5. Provide systematic and less stress to test method than normal debugging;

18.3.3 How to Use JUnit

JUnit provides both Text Runner and the Graphical Runner that will run the test cases. Similarly with NUnit, Graphical Runner of JUnit provides a visual indication of the success or failure of the selected tests, Text Runner can be used for faster test case compilation and run. The testers are required to:

1. write test cases for the Java class to be tested according to the methodologies adopted as specified in the TP
2. optionally, the test cases can be added to the test suite for easiler selective of the tests to be run
3. compile the test cases
4. run JUnit Runner (either Text or Graphical)
5. for graphical runner, JUnit will report back with green (successfully pass the test cases) or red (failed) or just report back in text form (for text runner)
6. for our project, ANT will be used to automate the running and reporting of JUnit (Refer to 18.4.2 for the information on this).

18.4 Apache Ant

Apache Ant is a Java-based build tool. Basically, the way it's working is very similar with Makefile. It does take a arguments to run certain shell-script to build and run pariticular functions. However, the huge difference between Makefile and Apache Ant is that Makefile is only inherently shell-based, that execute commands specified as arguments.

Ant is different. Instead of a model where it is extended with shell-based commands, it is extended using Java classes. Its configuration files are XML-based, which can generate a test report in HTML files to be published. Furthermore, Apache Ant is able to compile both the source files and the test case files, and run the set of tests prepared.

18.4.1 Benefits of Apache Ant

Below is the benefits of using Apache Ant:

1. Easy to extend and run. With a few commands, Apache Ant is able to compile and run the test cases.
2. Apache Ant is able to produce the test reports automatically in HTML format.
3. Easy and simple to learn. This is due to other members outside of the Testing Sub-Team are going to run the test, which this benefit will be very useful for us.

18.4.2 How to Use Apache Ant

A Templates

A.1 Change Log

Date	**Section**	**Descriptions**
DD/MM/YY	Section number	Description of the changes.
31/07/04	4.1.2	Inserted review and inspection reports and checklist naming conventions.

Table 3: Change Log

A.2 Build File

```
--------------------------------------------------------------------------------
<?xml version="1.0"?>

<project name="vtest" basedir="." default="run">
    <description>433-340 Team V build file for testing Intelligent Lifestyle</description>

    <!-- =========== directory definitions ============= -->

    <!-- Java sources of the application                   -->
    <property name="source.dir"    value="../Code/Common"/>

    <!-- .class file directory                             -->
    <property name="build.dir"     value="./Build"/>

    <!-- Java sources of the unit tests                    -->
    <property name="uTest.dir" value="./Unit"/>

    <!-- Java sources of the integration tests             -->
    <property name="iTest.dir"  value="./Integration" />

    <!-- Java sources of the javadocs             -->
    <property name="doc.dir"  value="./Doc" />

    <path id="Intelligent">
        <pathelement path="${source.dir}"/>
        <pathelement path="${source.dir}/Common"/>
        <pathelement path="${source.dir}/Application"/>
        <pathelement path="${source.dir}/Context"/>
        <pathelement path="${source.dir}/Hardware"/>
        <pathelement path="${source.dir}/Jade/lib/Base64.jar"/>
        <pathelement path="${source.dir}/Jade/lib/iiop.jar"/>
        <pathelement path="${source.dir}/Jade/lib/jade.jar"/>
        <pathelement path="${source.dir}/Jade/lib/jadeTools.jar"/>
        <pathelement path="${build.dir}"/>
    </path>

    <target name="buildinit">
        <mkdir dir="${build.dir}"/>
    </target>

    <target name="compilesource" depends="buildinit" description="Compiling all source codes"
        <javac srcdir="${source.dir}" destdir="${build.dir}">
    <classpath refid="Intelligent"/>
            <include name="**/*.java"/>
        </javac>
    </target>
```

```
    <target name="clean" description="Cleaning up the .class files from build directory">
        <delete dir="${build.dir}"/>
    </target>

    <target name="run" depends="compilesource" description="Running Intelligent Lifestyle">
<java classname="MainWindow" fork="true">
    <classpath refid="Intelligent"/>
</java>
    </target>

    <target name="JUNIT" description="Checking if JUnit present">
         <echo message="Checking if there is JUnit.."/>
        <available property="junit.present" classname="junit.framework.TestCase" />
    </target>

    <target name="unitinit" depends="JUNIT">
<fail unless="cn"
message="Need to specify -Dcn="/>
<fail unless="ln"
message="Need to specify -Dln="/>
<tstamp/>
<property name="testbuild.dir" value="${build.dir}"/>
<property name="test.dir" value="${uTest.dir}/${ln}"/>
        <echo message="${test.dir}"/>
<property name="logfile"
value="${test.dir}/test_results/log_${cn}_${DSTAMP}__${TSTAMP}"/>

        <path id="testclasspath">
            <pathelement path="${testbuild.dir}"/>
            <pathelement path="${test.dir}"/>
            <pathelement path="${test.dir}/test_results"/>
<!--
    <pathelement path="${classpath}"/>
-->
        <pathelement path="${source.dir}"/>
        <pathelement path="${source.dir}/Common"/>
        <pathelement path="${source.dir}/Application"/>
        <pathelement path="${source.dir}/Context"/>
        <pathelement path="${source.dir}/Hardware"/>
        <pathelement path="${source.dir}/Jade/lib/Base64.jar"/>
        <pathelement path="${source.dir}/Jade/lib/iiop.jar"/>
        <pathelement path="${source.dir}/Jade/lib/jade.jar"/>
        <pathelement path="${source.dir}/Jade/lib/jadeTools.jar"/>
        <pathelement path="${build.dir}"/>
        </path>
    </target>
```

```
    <target name="compileunit"
            depends="unitinit, buildinit, compilesource"
    description="Compiling unit and its driver">
        <javac srcdir="${test.dir}"
               destdir="${testbuild.dir}"
               debug="on">
    <classpath refid="testclasspath"/>
        </javac>
    </target>

    <target name="testunit" depends="compileunit" if="junit.present" description="Running uni
        <echo file="${logfile}">Test run at ${DSTAMP} ${TSTAMP} for file ${cn}</echo>
<record name="${logfile}" append="yes" action="start"/>

        <junit fork="true" printsummary="false" errorProperty="test.failed"
failureProperty="test.failed" >
    <classpath refid="Intelligent"/>
            <formatter type="brief" usefile="false" />
            <formatter type="xml" />
    <batchtest fork="true" todir="${test.dir}/test_results">
        <fileset dir="${testbuild.dir}" >
    <include name="**/${cn}TestCase.class" />
        </fileset>
            </batchtest>
        </junit>

        <junitreport todir="${test.dir}/test_results">
    <fileset dir="${test.dir}/test_results">
        <include name="TEST-*.xml" />
    </fileset>
    <report format="frames" todir="${test.dir}/test_results" />
        </junitreport>

        <fail message="test failed. Check log and reports." if="test.failed" />
    </target>

    <target name="cleanunit" depends="unitinit" description="Cleaning up after unit testing">
        <delete>
            <fileset dir="${build.dir}" includes="**/${cn}TestCase.class"/>
        </delete>

    </target>

    <target name="threadinit" depends="JUNIT">
<fail unless="tn"
         message="Need to specify -Dtn="/>
<tstamp/>
<property name="test.dir" value="${iTest.dir}/${tn}"/>
```

```
<property name="testbuild.dir" value="${build.dir}"/>
<property name="logfile"
    value="${test.dir}/test_results/log_${tn}_${DSTAMP}_${TSTAMP}"/>

        <path id="testclasspath">
            <pathelement path="${testbuild.dir}"/>
            <pathelement path="${test.dir}"/>
            <pathelement path="${test.dir}/test_results"/>
    <pathelement path="${classpath}"/>
        </path>
    </target>

    <target name="compilethread"
    depends="threadinit, buildinit, compilesource"
    description="Compiling thread and its driver">
        <javac srcdir="${test.dir}"
               destdir="${testbuild.dir}"
               debug="on">
    <classpath refid="testclasspath"/>
        </javac>
    </target>

    <target name="testthread" depends="compilethread" if="junit.present"
            description="Running integration test">
        <echo file="${logfile}">Test run at ${DSTAMP} ${TSTAMP} for file ${cn}</echo>
<record name="${logfile}" append="yes" action="start"/>

        <junit fork="true" printsummary="false" errorProperty="test.failed"
failureProperty="test.failed" >
    <classpath refid="testclasspath"/>
            <formatter type="brief" usefile="false" />
            <formatter type="xml" />
    <batchtest fork="true" todir="${test.dir}/test_results">
        <fileset dir="${testbuild.dir}" >
    <include name="**/*TestCase.class" />
        </fileset>
            </batchtest>
        </junit>

        <junitreport todir="${test.dir}/test_results">
    <fileset dir="${test.dir}/test_results">
        <include name="TEST-*.xml" />
    </fileset>
    <report format="frames" todir="${test.dir}/test_results" />
        </junitreport>

        <fail message="test failed. Check log and reports." if="test.failed" />
    </target>
```

```
    <!-- can also generate the javadocs, if you want -->
    <target name="javadocs" description="Outputting the documentation">
        <mkdir dir="api"/>
        <javadoc
                destdir="api"
                author="true"
                version="true"
                use="true"
                windowtitle="Test API">

            <fileset dir="${source.dir}" defaultexcludes="yes">
                <include name="**/*.java" />
            </fileset>
        </javadoc>

    </target>

</project>



--------------------------------------------------------------------------------
```

A.3 Unit Testing Test Cases for Java

```
------------------------------------------------------------------------------

/*
 * <filename>
 *
 * Copyright (c) 2004-2009
 * Department of Computer Science and Software Engineering
 * The University of Melbourne
 * Victoria 3010, Australia
 *
 * This file is part of the test codes of the Intelligent Lifestyle project
 * undertaken by Team Daedalus. The Intelligent Lifestyle project will
 * demonstrate the concepts of intelligent agents with open spaces in an
 * adaptive environment.
 */


/**
* <p>This file serves as the part of Unit Testing for source codes that
* are written in java only.<p>
*
*
* Author   : <author of this file>
* Module   : <name of module tested>
*
*/


import junit.framework.*;

<import some other packages here>

public class <module name>TestCase extends TestCase {

  public <module name>TestCase(String name){
    super(name);
  }

  /* Initializing object state */
  public void setUp() {
    <insert declaration of the new class here>;
  }

  //================
  //GREY BOX TESTING
  //================
```

```
 /* Test cases for the module using grey box testing technique*/
 public void test_grey_<method name>() {
   <insert the code to test here>;
 }

 public void test_grey_<method name>() {
   <insert the code to test here>;
 }

 //=================
 //EXCEPTION TESTING
 //=================

 /* Test cases for the module using exception testing technique*/

public void test_exception_<method name>_1() {
   <insert the code to test here>;
 }

 public void test_exception_<method name>_2() {
   <insert the code to test here>;
 }

 //================
 //DATABASE TESTING
 //================

 /* Test cases for the module using database testing technique*/
 public void test_database_<method name>_1() {
   <insert the code to test here>;
 }

 public void test_database_<method name>_2() {
   <insert the code to test here>;
 }

 //==============================
 //BOUNDARY VALUE ANALYSIS TESTING
 //==============================

/* Test cases for the module using boundary value analysis
    testing technique */
 public void test_boundary_<method name>_1() {
   <insert the code to test here>;
 }

 public void test_boundary_<method name>_2() {
```

```
    <insert the code to test here>;
  }

  //================
  //COVERAGE TESTING
  //================

 /* Test cases for the module using coverage testing technique*/
  public void test_coverage_<method name>_1() {
    <insert the code to test here>;
  }

  public void test_coverage_<method name>_2() {
    <insert the code to test here>;
  }

  /* Clearing object state */
  public void tearDown() {
    <insert the variable to be freed>;
  }
}
```

A.4 JUnit Assertions

```
--------------------------------------------------------------------------------

The followings are the assertions that are to be used in asserting some values
to be tested in creating the unit testing cases.  These assertions will be the
one that JUnit looks at and test.


- assertEquals(expected, actual)

- assertEquals(String message, expected, actual)

This method inserts a message whenever a failure for this assertion
occurs.  This way it can clear up which assertions are failed.  Useful
when you have more than one assertions in one testing method.

This goes with the other assertions too.

- assertNull(Object object)

- assertNull(String message, Object object)

- assertNotNull(Object object)

- assertNotNull(String message, Object object)

- assertSame(Object expected, Object actual)
This is similar to assertEquals

- assertSame(String message, Object expected, Object actual)

- assertTrue(boolean condition)

- assertTrue(String message, boolean condition)

- fail()
This forces a failure.  This is useful to close off paths through the
code that should not be reached.

- fail(String message)

--------------------------------------------------------------------------------
```

A.5 Integration Testing Test Cases

```
--------------------------------------------------------------------------------


/*
 * <filename>
 *
 * Copyright (c) 2004-2009
 * Department of Computer Science and Software Engineering
 * The University of Melbourne
 * Victoria 3010, Australia
 *
 * This file is part of the test codes of the Intelligent Lifestyle project
 * undertaken by Team Daedalus. The Intelligent Lifestyle project will
 * demonstrate the concepts of intelligent agents with open spaces in an
 * adaptive environment.
 */


/**
* <p>This file serves as the part of Unit Testing for source codes that
* are written in java only.<p>
*
*
* Author   : <author of this file>
* Function : <description on the function tested>
*
*/


<insert test case here>


--------------------------------------------------------------------------------
```

A.6 Integration Testing Report

```
-------------------------------------------------------------------------------


/*
 * <filename>
 *
 * Copyright (c) 2004-2009
 * Department of Computer Science and Software Engineering
 * The University of Melbourne
 * Victoria 3010, Australia
 *
 */


/**
* <p>This file serves as the test result for Integration Testing. <p>
*
*
* Author   : <author of this file>
* Result   : <result of the test>
* Description : <insert the description of the integration here>
*/

==========
RESULT LOG
==========


<insert the result log obtained here>

===============================================================




-------------------------------------------------------------------------------
```

A.7 Agent Testing Test Cases

```
--------------------------------------------------------------------------------


/*
 * <filename>
 *
 * Copyright (c) 2004-2009
 * Department of Computer Science and Software Engineering
 * The University of Melbourne
 * Victoria 3010, Australia
 *
 * This file is part of the test codes of the Intelligent Lifestyle project
 * undertaken by Team Daedalus. The Intelligent Lifestyle project will
 * demonstrate the concepts of intelligent agents with open spaces in an
 * adaptive environment.
 */


/**
* <p>This file serves as the part of Agent Testing.  This file will act
* as the test cases, that provide inputs and expected output for the
* agent tested.<p>
*
*
* Author   : <author of this file>
* Agent    : <name of the agent tested>
*
*/


package test.adder.tests;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;
import jade.util.leap.Properties;
import test.common.*;

import test.<name of the dir where this test stored>.<name of the tester agent>;

public class <classname of this test case> extends Test {

  public Behaviour load(Agent a) throws TestException {
   // The test must complete in 10 sec
   setTimeout(10000);
```

```
   try {
   Behaviour b = new CyclicBehaviour() {
   private boolean finished = false;
   public void onStart() {
<construct the input, as ACL message format, here>
       myAgent.send(request);
       System.out.println("request has been sent");
   }

   public void action() {
ACLMessage reply = myAgent.receive();

<construct the assertion here>

   }
   };

   return b;
   }
   catch (Exception e) {
   throw new TestException("Error loading test", e);
   }
  }

  public void clean(Agent a) {
   try {
   //TestUtility.killAgent(a, resp);
   Thread.sleep(1000);
   }
   catch (Exception e) {
   e.printStackTrace();
   }
  }
}
```

A.8 Agent Testing Report

```
--------------------------------------------------------------------------------


/*
 * <filename>
 *
 * Copyright (c) 2004-2009
 * Department of Computer Science and Software Engineering
 * The University of Melbourne
 * Victoria 3010, Australia
 *
 */


/**
* <p>This file serves as the test result for Agent Testing. <p>
*
*
* Author   : <author of this file>
* Agent    : <name of the agent tested>
* Result   : <result of the test>
*/

==========
RESULT LOG
==========


<insert the result log obtained from the 'screen' here>

================================================================


--------------------------------------------------------------------------------
```

A.9 Agent Testing Tester Agent

```
/*
 * <filename>
 *
 * Copyright (c) 2004-2009
 * Department of Computer Science and Software Engineering
 * The University of Melbourne
 * Victoria 3010, Australia
 *
 * This file is part of the test codes of the Intelligent Lifestyle project
 * undertaken by Team Daedalus. The Intelligent Lifestyle project will
 * demonstrate the concepts of intelligent agents with open spaces in an
 * adaptive environment.
 */


/**
* <p>This file serves as the part of Agent Testing.  This files will act
* as an agent that will communicate with the agent tested.  The test cases
* created will send and receive message through this agent.<p>
*
*
* Author   : <author of this file>
* Agent    : <name of agent tested>
*
*/

package test.<directory where the test cases are stored>;

import jade.core.Agent;
import jade.core.Runtime;
import jade.core.Profile;
import jade.core.ProfileImpl;
import jade.core.AID;
import jade.wrapper.*;
import jade.util.leap.*;
import test.common.*;
import java.io.*;
import java.net.InetAddress;

public class <name of this class> extends TesterAgent {

    // Names and default values for group arguments
    public static final String REMOTE_AMS_KEY = "remote-ams";
```

```
    public static final String REMOTE_PLATFORM_NAME = "Remote-platform";
    public static final String REMOTE_PLATFORM_PORT = "9003";

    public static final String MTP_KEY = "mtp";
    public static final String PROTO_KEY = "proto";
    public static final String MTP_URL_KEY = "url";
    public static final String MTP_URL_DEFAULT = "";
    public static final String ADDITIONAL_CLASSPATH_KEY = "classpath";
    public static final String ADDITIONAL_CLASSPATH_DEFAULT =
    "c:/jade/add-ons/http/classes";

    protected TestGroup getTestGroup() {
        TestGroup tg = new TestGroup("test/<directory>/<xml files>"){

            private JadeController jc;

            public void initialize(Agent a) throws TestException {
                try {
String addClasspath = "+"+((String) getArgument(
ADDITIONAL_CLASSPATH_KEY));
String mtp = (String) getArgument(MTP_KEY);

                    // Start a local container with the specified MTP
                    String url = (String) getArgument(MTP_URL_KEY);
                    jc = TestUtility.launchJadeInstance("Container-mtp",
                    addClasspath, new String("-container -host "+TestUtility.
                    getLocalHostName()+" -port "+String.valueOf(Test.
                    DEFAULT_PORT)+" -mtp "+mtp+"("+url+")"), null);
                }
                catch (TestException te) {
                    throw te;
                }
                catch (Exception e) {
                    throw new TestException("Error initializing TestGroup", e);
                }
            }

            public void shutdown(Agent a) {
            try {
                // Kill the remote platform and the mtp container
                Thread.sleep(1000);
                jc.kill();
            }
            catch (Exception e) {
                e.printStackTrace();
            }
          }
```

```
        };

        tg.specifyArgument(MTP_KEY, "MTP", Test.DEFAULT_MTP);
        tg.specifyArgument(PROTO_KEY, "Protocol", Test.DEFAULT_PROTO);
        tg.specifyArgument(MTP_URL_KEY, "Local MTP URL", MTP_URL_DEFAULT);
        tg.specifyArgument(ADDITIONAL_CLASSPATH_KEY, "Additional classpath",
        ADDITIONAL_CLASSPATH_DEFAULT);
        return tg;
    }

    // Main method that allows launching this test as a stand-alone program
    public static void main(String[] args) {
try {
// Get a hold on JADE runtime
Runtime rt = Runtime.instance();

// Exit the JVM when there are no more containers around
rt.setCloseVM(true);

Profile pMain = new ProfileImpl(null, Test.DEFAULT_PORT, null);

AgentContainer mc = rt.createMainContainer(pMain);

AgentController tester = mc.createNewAgent("<name of tester agent>",
"test.<directory>.<name of tester agent>", args);
tester.start();

AgentController <name of agent> = mc.createNewAgent("<name of agent>",
"<directory where the tester agent class is stored", args);
<name of agent>.start();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

A.10 Agent Testing Test List

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Relative reference to the DTD file does not work when the test suite is in
a jar file
<!DOCTYPE TestsList SYSTEM "../common/xml/listTests.dtd" [
    <!ENTITY % inline "#PCDATA|em|br|UL|LI|b">
    ]>
-->
    <TestsList>
    <Test name="<name of testing activity>" skip="false">
    <TestClassName><directory where class of the test case is stored></TestClassName>
            <WhatTest>describes what is being tested...</WhatTest>
            <HowWorkTest>describes how test works...</HowWorkTest>
            <WhenTestPass>describes when test passes...</WhenTestPass>
        </Test>
    </TestsList>
```

A.11 Reliability Modelling Report

```
--------------------------------------------------------------------------------


/*
 * <filename>
 *
 * Copyright (c) 2004-2009
 * Department of Computer Science and Software Engineering
 * The University of Melbourne
 * Victoria 3010, Australia
 *
 */


/**
* <p>This file serves as the test result for Reliability Modelling. <p>
*
*/

================================================================
RESULT LOG #1
================================================================

Tester : <the name of the tester>

Date : <the date when the reliability modelling is
conducted>

Result : <Pass / Fail>
<This result will be based on the realibility
goal>

Log :

<Insert the time recorded for each error occurred>

================================================================


================================================================
RESULT LOG #2
================================================================

Tester : <the name of the tester>

Date : <the date when the reliability modelling is
```

```
conducted>

Result : <Pass / Fail>
<This result will be based on the realibility
goal>

Log :

<Insert the time recorded for each error occurred>

===============================================================
```

```
-------------------------------------------------------------------------------
```

A.12 Usability Testing Templates - PDA Heuristic Evaluation Checklist

1. RESPONSE TIME: The system responds to user input within 3 seconds.
2. VISIBILITY: The messages appearing on the devices is noticeable.
3. UNDERSTANDABILITY: The system's feedback is simple and does not contain any extra information that is irrelevant. Instructions/navigations of the Guider is easy to interpret and coherent, such that it lead you to the correct venue.
4. ERROR PREVENTION: The system provides error prevention mechanism such that the GUI is not misleading to the user such that they will choose the wrong option.
5. REVERSIBLE ACTIONS: The GUI caters for the user to go back to previous operations if user choose the wrong option by mistake.
6. FEEDBACK - There is adequate feedback when you had anticipated it from the system. ¡If you feel otherwise please note it down as side comments as to when you think there was a lack of feedback¿
7. PREDICTABILITY: It can be predicted what the current operation will lead to in the next step/operation.
8. CONSISTENCY - The GUI is consistent in its look and feel.

A.13 Usability Testing Templates - STT Heuristic Evaluation Checklist

1. NORMAL INTERACTION - You feel that you didn't have to talk in a different tone/manner to particularly emphasize words when using STT.
2. ACCURACY - The SST was accurate in translating verbal commands.
3. RESPONSE TIME: The system responds to user input within 3 seconds.
4. CONSISTENCY - The commands used for voice activation consistent (eg. if the user needs to say some keyword ie "Daedalus" to acitivate the SST, and if so, is this keyword used for all other operation commands etc).
5. FEEDBACK - There is adequate feedback when you had anticipated it from the system. ¡If you feel otherwise please note it down as side comments as to when you think there was a lack of feedback¿

A.14 Usability Testing Templates - TTS Heuristic Evaluation Checklist

1. FRIENDLINESS - The system display friendliness and politeness in the feedback of the system.
2. HUMAN-LIKE TONE - The feeling of interacting with human; such that you feel as though the tone exhibits human like qualities and that you're communicating with human.
3. AUDIBLE - The feedback given by the system could be heard or perceptible by the ear.
4. RESPONSE TIME: The system responds to user input within 3 seconds.
5. UNDERSTANDABILITY: The level of simplicity of the system's feedback of speech output and whether it is easily understood by a random user.
6. FEEDBACK - There is adequate feedback when you had anticipated it from the system. ¡If you feel otherwise please note it down as side comments as to when you think there was a lack of feedback¿

A.15 Installation Testing Report

```
--------------------------------------------------------------------------------


/*
 * <filename>
 *
 * Copyright (c) 2004-2009
 * Department of Computer Science and Software Engineering
 * The University of Melbourne
 * Victoria 3010, Australia
 *
 */


/**
* <p>This file serves as the test result for Installation Testing. <p>
*
*
* Author   : <author of this file>
* Result   : <result of the test>
* Date    : <date of testing>
*/

==========
RESULT LOG
==========


<insert the outcome of the installation testing here>

================================================================


--------------------------------------------------------------------------------
```