Team Redmond

Master Test Plan

Version 1.2

Revision History

Date	Version	Description	Author
11/15/2003	1.0	Modified template as per group decision on the sections to include / exclude.	Robert Hanna
12/01/2003	1.1	Integration of individual parts together	Stefan Thibeault
12/04/2003	1.2	Finalize document	Stefan Thibealt Robert Hanna

Table of Contents

Master Test Plan

1. Introduction

The primary goal of this project is to develop the Montrealopoly game. This game is based on the original Monopoly© game, with some modifications. Some of the original rules of the game have been changed. Further, the game board and cell names have been modified to a Montreal-based theme. This is the final phase of the project, which includes the test plan and the implementation of the game. This test plan contains a comprehensive list of tests that will be performed along with a workflow of how the tests will be executed.

1.1 Purpose

The purpose of the Iteration Test Plan is to gather all of the information necessary to plan and control the test effort for this phase.

This Test Plan for the Montrealopoly game supports the following objectives:

- Identify the requirements that are to be tested.
- Outline the testing approach that will be used.
- Describe the workflow of the testing process that must be executed.
- Provide a timeline with milestones for the testing phase.

1.2 Scope

This document is intended to provide a test plan to test the Montrealopoly game, which Team Redmond developed. The test plan will consist of unit, integration, function, user interface, performance profiling, load, configuration and installation testing. Testing techniques that will be performed include white box and black box testing, boundary testing and basis path testing. Some tests that were omitted in the test plan include: Data and Database Integrity, Business Cycle, Stress, Volume, Security and Access Control, Failover and Recovery testing. A test plan workflow will also be included along with milestones that have been set for this phase.

1.3 Document Terminology and Acronyms

Term	Definition
BVA	Boundary Value Analysis
GUI	Graphical User Interface
AI	Artificial Intelligence
QA	Quality Assurance
API	Application Programming Interface
VB	Visual Basic

1.4 References

- Pressman, Roger S. Software Engineering: A Practitioner's Approach. 5th ed. Toronto: McGraw-Hill, 2001.
- Dr. Joey Paquet, "COMP 354 Course Notes" http://newton.cs.concordia.ca/~paquet/teaching/354/notes/COMP354F2003notesAll.pdf (Current December 1, 2003)
- Paula Bo Lu, "COMP 354 Tutorial 3" http://www.cs.concordia.ca/~grad/blu/comp354-2.ppt (Current December 1, 2003)
- Microsoft, “Virtual PC”, http://www.microsoft.com/windowsxp/virtualpc/ (Current December 1, 2003)

1.5 Document Structure

The remainder of this document is divided into following major parts: evaluation mission and test motivation, target test items, outline of planned tests, test approach and testing workflow, iteration milestones. The evaluation mission and test motivation contains a brief background on this project, its objectives and motivators for testing. The target test items and outline of planned tests include what will be tested and what tests will not be performed. The test approach contains the actual tests that were performed and how the tests were carried out. The testing workflow contains the workflow that Team Redmond followed in this phase. The last two sections contain the milestones of this phase and the team member's log sheets.

2. Evaluation Mission and Test Motivation

The goal of this test plan is to ensure that the Montrealopoly game meets the specifications and design criteria of the two previous phases. Moreover, the test plan will provide a methodology on what the implementation team should test and the types of tests they will perform. Finally, the test plan will enable Team Redmond to release a stable and bug-free Montrealopoly game.

2.1 Background

The third phase of the COMP 354 project involves creating the actual Montrealopoly game based on the requirements and design documents of the two previous phases. The game will be developed by the implementation using Visual Basic. A comprehensive test plan has been developed to ensure that the game conforms to the specifications, design and to perform quality assurance on the final product. This will enable Team Redmond to release a complete and bug free Montrealopoly game and minimize the risk of software failure.

The requirements document outlines the game’s specifications and high-level requirements along with an analysis model with use cases, class diagrams, sequence diagrams and state transition diagrams of the game. The design document contains architectural, software interface and internal module designs, which is a foundation that the implementation team can create Montrealopoly. The test plan will allow Team Redmond to verify if the final product successfully meets these specifications with a variety of testing techniques. The plan will also help in fault detection with the test cases that have been designed.

The requirements and design documents are available at http://montrealopoly.maverick.to

2.2 Evaluation Mission

The three main objectives of the third phase are:

- Ensuring that the specifications of the requirements document have been achieved.
- Ensuring that the specifications of the design document have been achieved.
- Ensuring that the risk of software failure is reduced to a minimum.

To achieve these objectives, Team Redmond has developed a test plan to verify that these objectives have been met. Meeting these objectives will enable Team Redmond to release a stable version on Montrealopoly.

2.3 Test Motivators

The targeted test items listed below will be the motivation for testing in this phase.

Unit Testing: A select number of methods will be tested in a couple of classes with black and white box testing to ensure that they function correctly.
Integration Testing: Units will be integrated with other units to see if they work correctly together.
Function Testing: Will ensure that the use cases have been met.

User Interface Testing: Will verify if the requirements of the GUI have been implemented as specified.
Performance Profiling: Ensure that the game's performance is at an acceptable playable level.
Load Testing: See how the game performs when being played at its limits.
Configuration Testing: Ensure that the game works correctly under different environment configurations.
Installation Testing: Verify that the game installs itself correctly under different environment configurations.

3. Target Test Items

In this section, we will list the target test items. These are the items that should be tested. Due to time restrictions, we were not able to document and generate test cases for all the target test items; therefore, although we list all the target test items, we only provide a detailed test plan for a few of the major test items. For ease of reference, we have categorized the test items by motivation.

Unit Testing

Unit testing consists of testing all the different units of the system, in isolation. In essence, we must therefore test each class in isolation, and each method in isolation using white box and black box techniques. The list of test items for unit testing consists of all the classes and all their methods, as per the design document. For a complete list of the classes and methods, please refer to the design document - section 4 – Internal Module Design. Below is a list of the test items for which test cases have been generated and included in this document:

- Function move
- Function payRent
- Function canBuy
- Function buyProperty
- Function canBuild
- Function doTrade

Integration Testing

During integration testing, we will be testing components separately, and then integrating them together one by one, and testing them again. Due to time restrictions, we have not included full test cases for all the integration tests that are to be done. Below is a list of the test items for which integration tests were documented and tested:

- Game Start Window
- Main Window
- JFL Window
- Cell Info Window
- Trade Window
- Game End Window

Function Testing

Function testing consists of testing all the requirements and specifications, as per the requirements and specifications document. In essence, the list of functions to test corresponds to the list of use cases and requirements in the requirements document. Due to the importance of function testing, we have included detailed test cases for all the product functions. Below is the list of functions that were tested:

- Start Game
- Roll Dice
- Pass Go
- Pay Rent
- Buy Property
- Build/Sell Hotel
- Mortgage/Un-Mortgage
- Tax
- JFL Cards

- Jail
- Trade
- End Turn
- Bankruptcy
- End Game
- Game Winner

User Interface testing

User interface testing is concerned with making sure that each functionality concerning the user interface is works as per the requirements defined in the design document. For the user interface, the possible interactions with the game will be tested in great detail. During the test, the objective will be to compare and check the validity of an implemented functionality with the expected functionality elaborated and described in previous phases. Below is a list of the User Interface items that were tested:

- Start Panel
- Game board
- Title deed cards
- Metro / Utility cards (as the title deed cards)
- Trading cards
- JFL cards
- Income / Luxury tax cards
- Winner interface

Performance Profiling

Performance profiling is concerned with testing the different response times of the software. In these types of tests, we have focused mainly on the following test items:

- Token Movements
- AI Response Time

Load Testing

Load Testing is concerned with testing the system beyond the limits it was designed for. In this type of test, we have focused mainly on testing the game when the board is fully loaded. This will be described in detail in section 5.5. Below are the test items that were identified:

- Functionality of Game with Fully Loaded Board
- AI Response Time with Fully Loaded Board

Configuration Testing

Configuration testing is concerned with testing the system under different environment configurations. In this type of test, we have focused on testing the game under different versions of the Windows ™ operating system. Below is a list of the operating systems the game will be tested under:

- Windows 95
- Windows 98
- Windows Me
- Windows 2K
- Windows XP

Installation Testing

During installation testing, we will focus on testing the packaged installation program that will be produced once the implementation is completed. For more details on this, refer to section 5.7. Below is a list of the test items that were identified:

- Installer
- Un-Installer

4. Outline of Planned Tests

Team Redmond will perform the following test: unit testing, integration testing, function testing, user interface testing, performance profiling, load testing, configuration testing and installation testing. The following tests will not be performed: data and database integrity testing, business cycle testing, stress testing, volume testing, security and access control testing and failover and recovery testing. A list of other candidates for potential inclusion is also provided.

4.1 Outline of Test Inclusions

The following tests will be performed to test the Montrealopoly game.

4.1.1 Unit Testing

Unit testing will be performed with black box and white box testing. Black box testing will include boundary value analysis and equivalence partitioning. White box testing will include basis path testing.

4.1.2 Integration Testing

Integration testing will allow testing of all the individually tested units together as a whole. Sandwich testing will be performed in the integration testing.

4.1.3 Function Testing

Function testing will ensure that the use cases have been implemented correctly by verifying if they are present in the game.

4.1.4 User Interface Testing

The GUI will be tested by comparing the requirements in the design document and with the actual implementation of the game.

4.1.5 Performance Profiling

Performance profiling will verify that the game's performance is at an acceptable playable level. The speed of the game's AI will be monitored to see whether the rate that it plays the game at is acceptable.

4.1.6 Load Testing

Load testing will see how the game performs when being played at its limits. This will be achieved by testing the game with the maximum allowable players, with all the properties owned and with hotels built on all streets.

4.1.7 Configuration Testing

Configuration testing is concerned with testing the application under different environment configurations the users may have.

4.1.8 Installation Testing

Installation testing will verify that the game installs itself correctly under different environment configurations the users may have.

4.2 Outline of Other Candidates for Potential Inclusion

Team Redmond's test plan contains a comprehensive amount of tests to help reduce the risk of software failure. However, with the extensive use of AI, several potential tests could be developed to test the effectiveness of the game's AI. These tests were not developed, as Team Redmond's knowledge of AI is limited and these tests are

complex and time consuming.

Playing the game with many computer players at the same time is difficult to test as there are many different paths that the computer player may take, depending on the state of the game. The more computer players a game has, the greater the difficulty in testing the different paths that any computer player can take. For example, trading performed between a couple of computers may execute correctly, but what if several computer players are interested in the same streets in a district? A property "fight" may break out and the computer players may keep trading the same streets back and forth. This could result in an endless loop or bankrupt players early on in the game.

The game's AI has been designed based on a decision tree which a computer player follows each time it plays it's turn. Each computer player uses the same decision tree and it never changes or adapts to the state of the game. As a result, someone playing the game often enough may start to notices patterns on how the computer reacts to certain situations. The game will then become less challenging, as human players will be able to predict what the computer player will do next. Another possibility is that weaknesses in the computer's decision-making abilities may be discovered. Human players who notice these patterns and weaknesses may use them to their advantages and trick the computer into performing poor moves. This will result in problems with game playability.

Game playability will be poor if the AI turns out to be ineffective as mention above. If the game's AI ends up in semi-infinite loops or makes poor decisions, the fun factor will quickly disappear. This will lead to a game that has no challenge if the computer player's decisions can be predicted or very difficult to play if property "fights" break out between computer players. Perfecting AI and testing it properly is difficult and is beyond the scope of this project and has been left out by Team Redmond.

4.3 Outline of Test Exclusions

Due to the nature of Montrealopoly's implementation, certain tests will be excluded, which are listed below.

4.3.1 Data and Database Integrity Testing

Montrealopoly does not use a database system, as no information is saved or retrieved. Any data that needs to be saved during game play is stored in main memory and is released when the game has ended.

4.3.2 Business Cycle Testing

Business cycle testing is not applicable to Montrealopoly as the game is not design to be played over long periods of time. It also is not time/date-sensitive and has been designed to be played within a maximum of several hours.

4.3.3 Stress Testing

Montrealopoly has been designed to be played with a maximum of eight players and be able to function correctly. Since the game cannot be played with any more players, stress testing cannot be applied. Furthermore, Team Redmond does not have the capabilities to simulate low system resources to test Montrealopoly. However, Team Redmond will conduct load testing to ensure that the game can be played at its designed limits.

4.3.4 Volume Testing

Volume testing will not be performed, as the game does not process large amounts of data. Besides mouse clicks, the only data that will be inputted into the game are the players' names and dollar amounts.

4.3.5 Security and Access Control Testing

No security testing will be performed as the game does not contain or manipulate any sensitive data. The game can be played by all and no sensitive information can be revealed while playing the game. All users playing the game are assumed to be allowed to use the computer that they are playing the game on.

4.3.6 Failover and Recovery Testing

Team Redmond does not have sufficient resources to perform failover and recovery testing. Moreover, the nature of Montrealopoly does not warrant these types of testing as there is little benefit of such testing as Montrealopoly is not a mission critical application.

5. Test Approach

The Test Approach describes the recommended strategy for designing and implementing the required tests. In this section, we will be describing the details of the tests that need to be performed for each target test item that was identified. These tests will be organized into the following sub-sections:

- Unit Testing
- Integration Testing
- Function Testing
- User Interface Testing
- Performance Profiling
- Load Testing
- Configuration Testing
- Installation Testing

Moreover, for each of these test motivators, test cases will be described in detail. For each test case, we will provide a description of the test case, the inputs (or steps to reproduce) of the test case, and the outputs (the expected results) of the test case.

5.1 Unit Testing

Unit testing will test individual components along with their functions in isolation. This low level form of testing will include black box testing and white box testing. In black box testing, the function's boundaries will be tested to see if any errors occur there. White box testing will verify that all the paths in the function are correct through basis path testing.

5.1.1 Function move

Tests will be conducted on the move function which is suppose to move the player from a starting position (x) to it's expected final destination (y). The movement is defined by the argument numCells so that (y = x + numCells). The function move takes a second argument beside numCells, which is penalty. Penalty is a Boolean argument. When the penalty is set too false the player collect 200$ when it passes go. When penalty is set to true, the player doesn't collect 200$ when it passes go.

5.1.1.1 Black Box Testing

Every test case will be tested starting from position 0, which is the GO cell. Moreover, the penalty argument will be set to false.

Test Case 1: Pass an argument that is under the lower bound for the variable numCells.
(numCells = -1)

Test Case 2: Pass an argument that is on the lower bound for the variable numCells.
(numCells = 0)

Test Case 3: Pass an argument that is between the bound.
(numCells = 10)

Test Case 4: Pass an argument that is exactly one lap around the board, there are 40 cells
(numCells = 40)

Test Case 5: Pass an argument that is more than one lap around the board, there are 40 cells.
(numCells = 45)

The expected result is the new player position or y as defined before.

Tester name	Patrice Michaud			Test date	November 29, 2003
Class name	Player	Method name	move	File name	Player.cls

Variable name	numCells	Lower bound	0	Upper bound:	40

less than lower bound	Value: -1
on lower bound	Value: 0
between the bounds	Value: 10
on the upper bound	Value: 40
greater than upper bound	Value: 45

Test case	less than lower bound	on lower bound	between the bounds	on the upper bound	greater than upper bound
Expected output	Position = -1	Position = 0	Position = 10	Position = 0 Balance + 200	Position = 5 Balance + 200
Actual output	Position = -1	Position = 0	Position = 10	Position = 0 Balance + 200	Position = 5 Balance + 200
Bug found?	No	No	No	No	No

5.1.1.2 White Box Testing

Basis Path Testing

```
Private position As Integer 'Actual Position Of The Player
Public Sub move(numCells As Integer, penalty As Boolean)
    'Move the player by the number of cells passed
    'Update the balance if you pass go and penalty is
    'equal to false

    Dim oldPos As Integer           'Starting position
    oldPos = position
    position = position + numCells   'New Position
    bBoard.getCell(oldPos).display   'Display the oldPos cell

    If (position >= 40 And penalty = False) Then
        MainWindow.MoveToken Me.getId, oldPos, 39
        oldPos = 0
        position = position mod 40
        credit 200
    Else
        If position >= 40 Then
            MainWindow.MoveToken Me.getId, oldPos, 39
            oldPos = 0
            position = position mod 40
        End If
    End If

    'Display the movement of the token
    MainWindow.MoveToken Me.getId, oldPos, position
End Sub
```

1 2 3 4 5 6

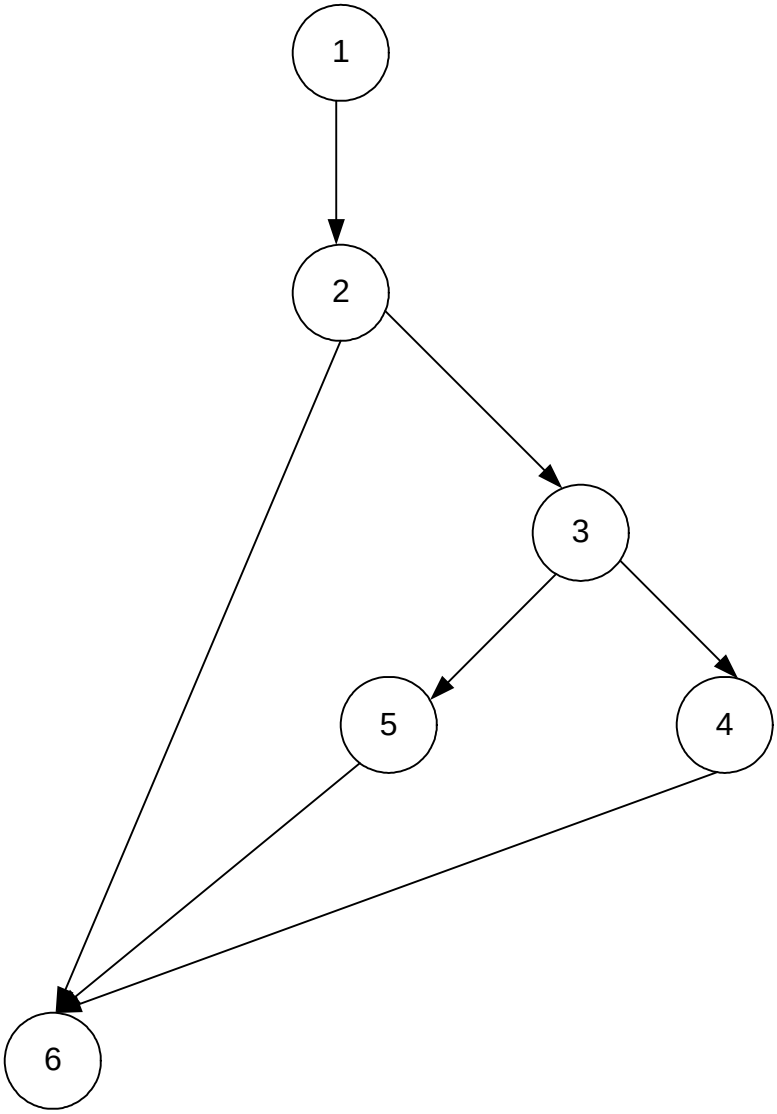
Path 1	1-2-6
Path 2	1-2-3-4-6
Path 3	1-2-3-5-6

Path 1	1-2-6
Variables	Position = 0 <= Position < 40.
Expected result	Return the correct position.

Path 2	1-2-3-4-6
Variables	Position = (Position >= 40). Penalty = false.
Expected result	Return the position and the balance go up by $200.

Path 3	1-2-3-5-6
Variables	Position = (Position >= 40) Penalty = true.
Expected result	Return correct position and the balance stays the same.

Path diagram for function: move



5.1.2 Function payRent

Tests will be conducted on the payRent function, which is suppose to make a player pay if he lands on land owned by someone else, or if the player landed on a tax cell. When called, the function takes the current position of the player and makes the player pay, if required, by the amount X. The function payRent does not take any argument.

5.1.2.1 Black Box Testing

Every test case will be tested using different player positions.

Test Case 1: Call the function when the player is on a negative position.
(Position = -1)

Test Case 2: Call the function when the player is on position 0.
(Position = 0)

Test Case 3: Call the function when the player is on a property.
(Position = 1)

Test Case 4: Call the function when the player is on the last property
(Position = 39)

Test Case 5: Call the function when the player is above the last possible position.
(Position = 40)

The expected result is the balance changed depending on the rent they have to pay. The rent is of 0 if a player land on a cell he owns a cell not owned or a mortgaged cell. It is important to notice that the function is only called if the player as enough money to pay the rent.

Tester name	Patrice Michaud			Test date	November 29, 2003
Class name	Player	Method name	payRent	File name	Player.cls

Variable name	cellId	Lower bound	0	Upper bound:	39

less than lower bound	Value: -1
on lower bound	Value: 0
between the bounds	Value: 1
on the upper bound	Value: 39
greater than upper bound	Value: 40

Test case	less than lower bound	on lower bound	between the bounds	on the upper bound	greater than upper bound
Expected output	nothing	nothing	Balance=-rent	Balance=-rent	nothing
Actual output	nothing	nothing	Balance=-rent	Balance=-rent	nothing
Bug found?	No	No	No	No	No

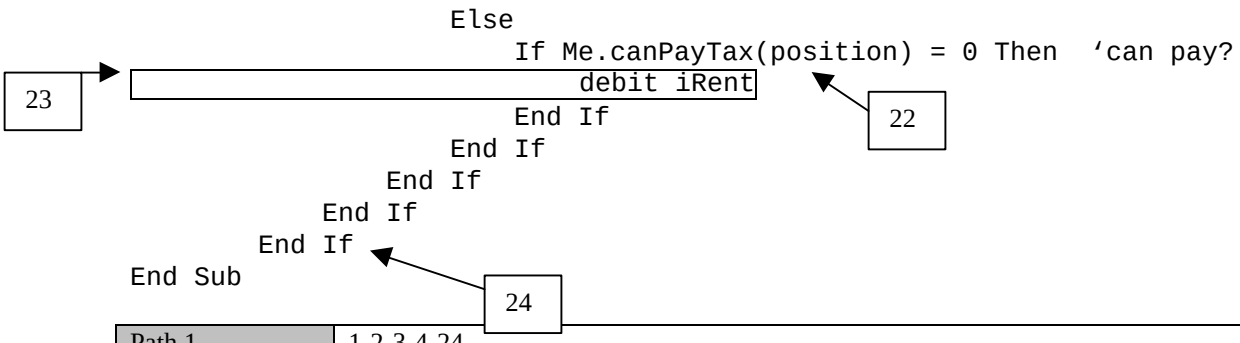
5.1.2.2 White Box Testing

Basis Path Testing

```
Private position As Integer ‘Actual Position Of The Player
Public Sub payRent()
‘Pay the rent of the cell you landed on
‘If tax pay tax, if income tax, check if assets are smaller then 2000
        Dim iRent As Integer
        If TypeOf bBoard.getCell(position) Is Property Then
            Dim pProp As Property
            Set pProp = bBoard.getCell(position)
            iRent = bBoard.getCell(position).getRent  ‘get the rent
            If pProp.getOwners Is Me Or pProp.isMortgaged2 Then

            Else
                If Me.isComputer2 Then    ‘if is a computer player
                    debit iRent
                    pProp.getOwners.credit iRent      ‘credit player
                Else
                    If Me.canPayRent(position) = 0 Then        ‘can pay?
                        debit iRent
                        pProp.getOwners.credit iRent  ‘credit player
                    End If
                End If
            End If
        Else
            If TypeOf bBoard.getCell(position) Is LuxuryTax Then
                Dim lTax As Variant
                Set lTax = bBoard.getCell(position)
                iRent = lTax.getRent      ‘get the tax
                If Me.isComputer2 Then                ‘is computer
                    debit iRent
                Else
                    If Me.canPayTax(position) = 0 Then      ‘can pay?
                        debit iRent
                    End If
                End If
            Else
                If TypeOf bBoard.getCell(position) Is IncomeTax Then
                    Dim iTax As Variant
                    Set iTax = bBoard.getCell(position)
                    iRent = iTax.getRent
                    If ((Me.calcAssets * 0.1) < iRent) Then‘assets<rent?
                        iRent = (Me.calcAssets * 0.1)
                    End If
                    If Me.isComputer2 Then            ‘is computer?
                        debit iRent
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21



Path 1	1-2-3-4-24
Path 2	1-2-3-4-5-24
Path 3	1-2-3-4-6-7-24
Path 4	1-2-3-4-5-6-8-24
Path 5	1-2-3-4-5-6-8-9-24

Path 1	1-2-3-4-24
Variables	Position is property. Player is the owner.
Expected result	Nothing.

Path 2	1-2-3-4-5-24
Variables	Position is property. Player is not the owner. Property is mortgaged.
Expected result	Nothing.

Path 3	1-2-3-4-6-7-24
Variables	Position is property. Owner is not me. Property is not mortgaged. Player is a computer.
Expected result	Balance of player is decreased by rent, Balance of owner increased by rent

Path 4	1-2-3-4-5-6-8-24
Variables	Position is property. Owner is not me. Property is not mortgaged. Player is not a computer. Player cannot pay rent.
Expected result	Nothing.

Path 5	1-2-3-4-5-6-8-9-24
Variables	Position is property. Owner is not me. Property is not mortgaged. Player is not a computer. Player can pay rent.
Expected result	Balance of player is decreased by rent, Balance of owner increased by rent.

Path 6	1-2-10-11-12-13-24
Path 7	1-2-10-11-12-14-24
Path 8	1-2-10-11-12-14-15-24

Path 6	1-2-10-11-12-13-24
Variables	Position is Luxury Tax. Player is computer.
Expected result	Player is debited $75.

Path 7	1-2-10-11-12-14-24
Variables	Position is Luxury Tax. Player is human. Player cannot pay tax.
Expected result	Player must make money first.

Path 8	1-2-10-11-12-14-15-24
Variables	Position is Luxury Tax. Player is human. Player can pay tax.
Expected result	Player is debited $75.

Path 9	1-2-10-16-17-18-20-21-24
Path 10	1-2-10-16-17-18-19-20-21-24
Path 11	1-2-10-16-17-18-20-22-24
Path 12	1-2-10-16-17-18-20-22-23-24
Path 13	1-2-10-16-17-18-19-20-22-24
Path 14	1-2-10-16-17-18-19-20-22-23-24

Path 9	1-2-10-16-17-18-20-21-24
Variables	Position is Income Tax. Player is computer. Player assets are more then $200.
Expected result	Player is debited $200.

Path 10	1-2-10-16-17-18-19-20-21-24
Variables	Position is Income Tax. Player is computer. Player assets are less then $200.
Expected result	Player is debited 10% of is total assets value.

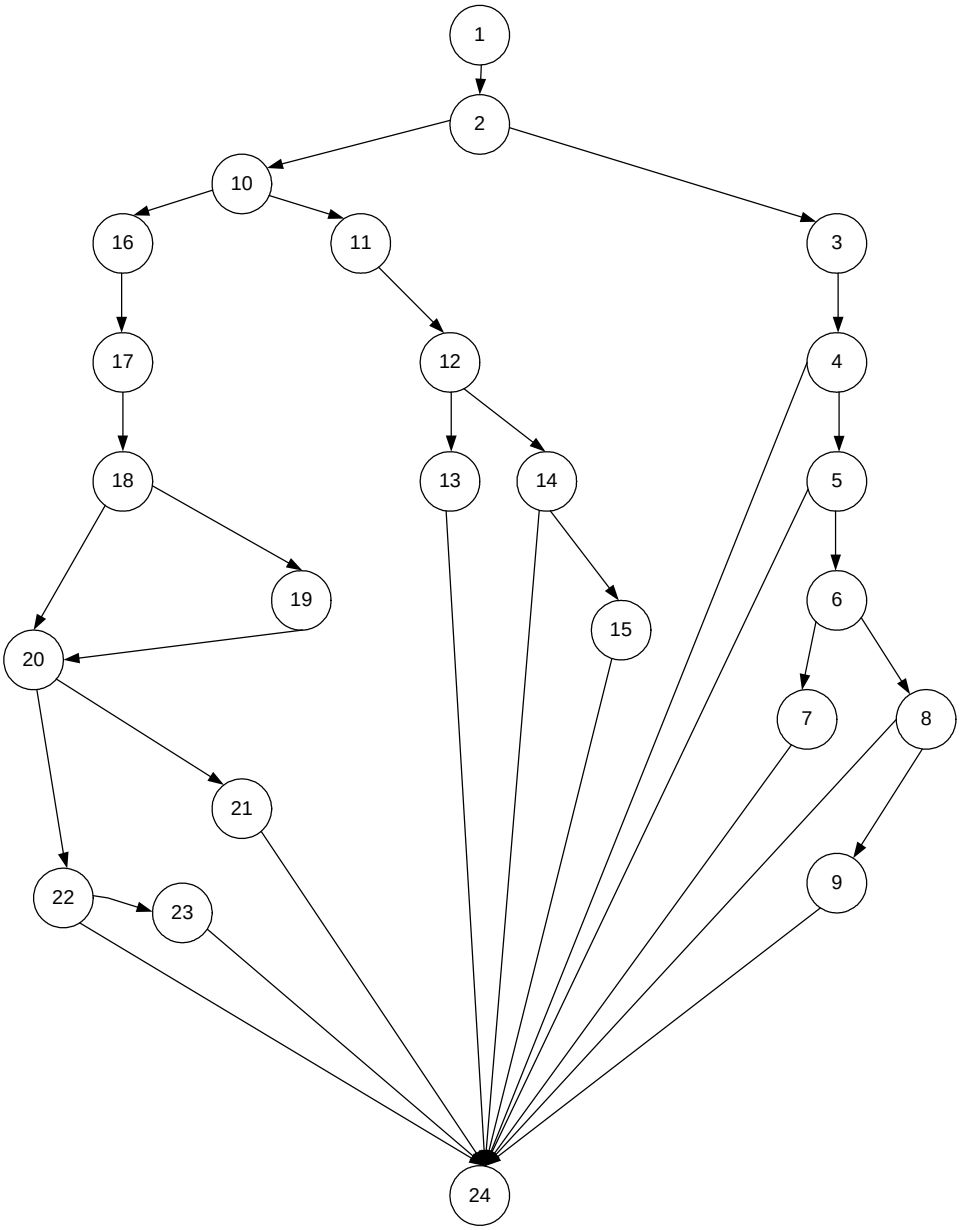
Path 11	1-2-10-16-17-18-20-22-24
Variables	Position is Income Tax. Player is human. Player assets are more then $200. Player cannot pay tax.
Expected result	Player must make money first.

Path 12	1-2-10-16-17-18-20-22-23-24
Variables	Position is Income Tax. Player is human. Player assets are more then $200. Player can pay tax.
Expected result	Player is debited $200.

Path 13	1-2-10-16-17-18-19-20-22-24
Variables	Position is Income Tax. Player is human. Player assets are less then $200. Player cannot pay tax.
Expected result	Player must make money first.

Path 14	1-2-10-16-17-18-19-20-22-23-24
Variables	Position is Income Tax. Player is human. Player assets are less then $200. Player can pay tax.
Expected result	Player is debited 10% of is total assets value.

Path diagram for function: payRent



5.1.3 Function canBuy

Tests will be conducted on the canBuy function which is suppose to return true if the player can buy a specified property X, and false if cannot buy a specified property X. The function takes the cellId argument, which is the cell id of the cell the player is interested in.

5.1.3.1 Black Box Testing

Every test case will be on a cell, and if the cell is a property then the owner is set to nothing and the balance is sufficient. If those conditions are false, every test case result will be false.

Test Case 1: Pass an argument that is under the lower bound for the variable cellId.
(cellId = -1)

Test Case 2: Pass an argument that is on the lower bound for the variable cellId.
(cellId = 0)

Test Case 3: Pass an argument that is between the bound.
(cellId = 1)

Test Case 4: Pass an argument that is exactly on the upper bound of cellId.
(cellId = 39)

Test Case 5: Pass an argument that is more than the upper bound of cellId.
(cellId = 40)

The expected result is the answers to the question can the player buy this property.

Tester name	Patrice Michaud			Test date	November 29, 2003
Class name	Player	Method name	canBuy	File name	Player.cls

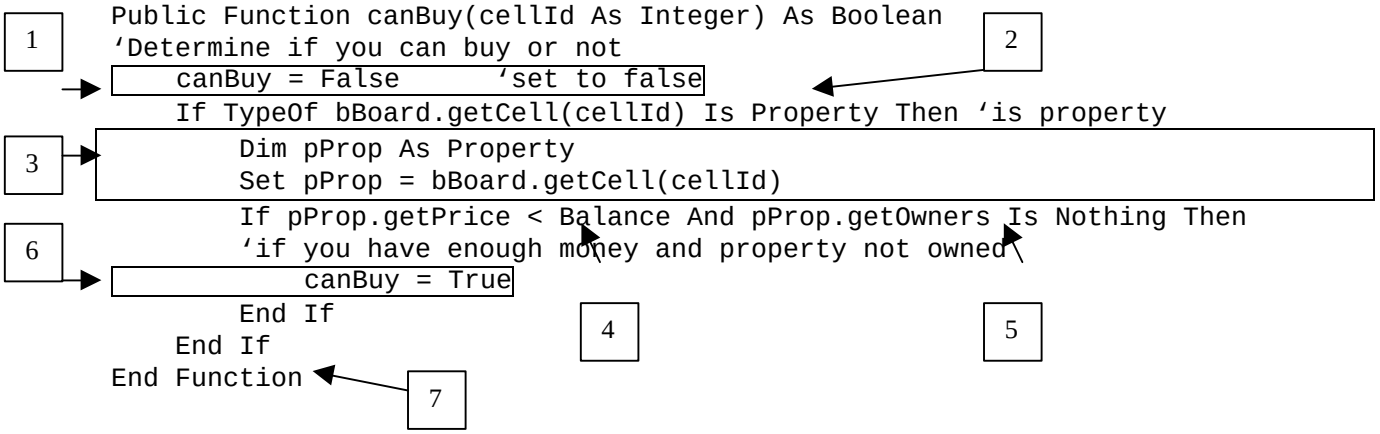
Variable name	cellId	Lower bound	0	Upper bound:	39

less than lower bound	Value: -1
on lower bound	Value: 0
between the bounds	Value: 1
on the upper bound	Value: 39
greater than upper bound	Value: 40

Test case	less than lower bound	on lower bound	between the bounds	on the upper bound	greater than upper bound
Expected output	false	false	true	true	false
Actual output	false	false	true	true	false
Bug found?	No	No	No	No	No

5.1.3.2 White Box Testing

Basis Path Testing



Path 1	1-2-7
Path 2	1-2-3-4-7
Path 3	1-2-3-4-5-7
Path 4	1-2-3-5-6-7

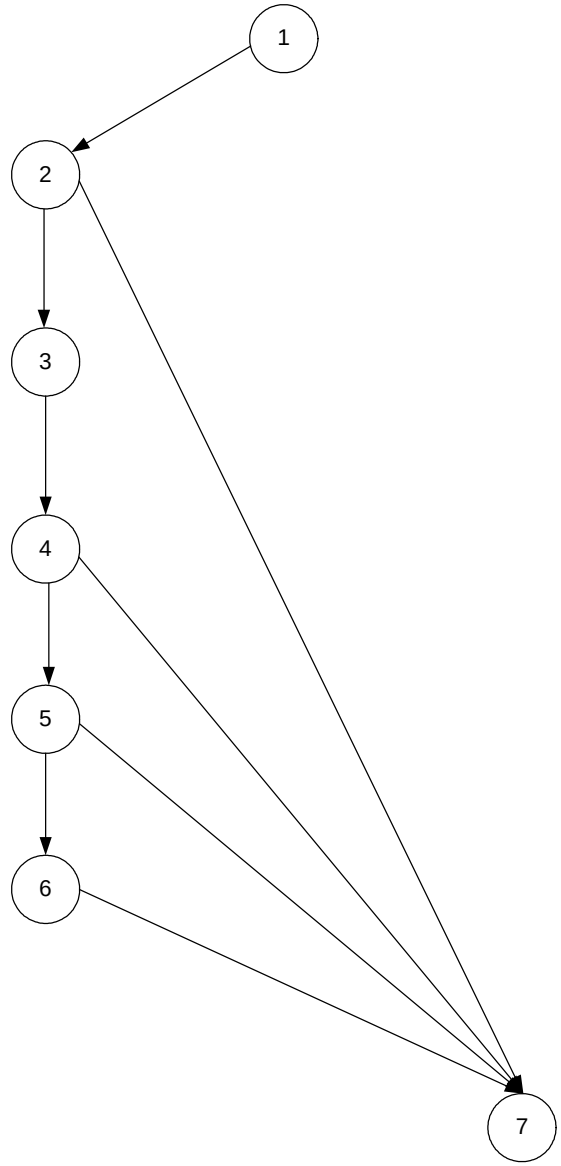
Path 1	1-2-7
Variables	cellId is not a property.
Expected result	Cannot buy.

Path 2	1-2-3-4-7
Variables	cellId is a property. Player balance is less then the price of the property.
Expected result	Cannot buy.

Path 3	1-2-3-4-5-7
Variables	cellId is a property. Player balance is more then the price of the property. The property is already owned.
Expected result	Cannot buy.

Path 4	1-2-3-5-6-7
Variables	cellId is a property. Player balance is more then the price of the property. The property is not owned.
Expected result	Can buy the property.

Path diagram for function: canBuy

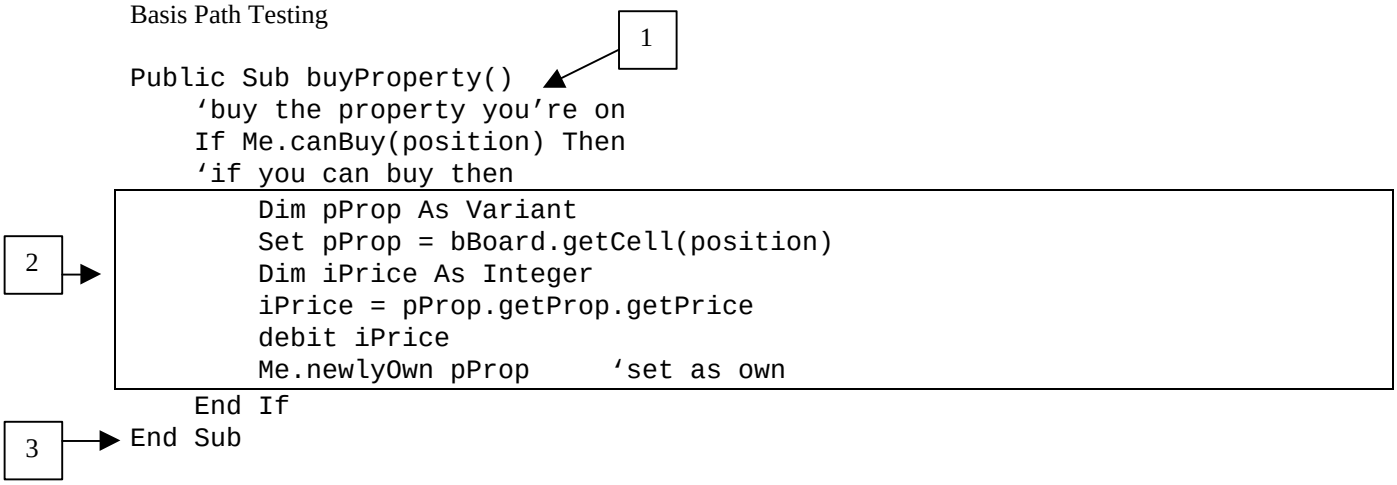


5.1.4 Function buyProperty

Test will be conduct on the buyProperty function, which is supposed to do the necessary transaction to buy a property. The function takes no argument, because it uses the current position of the player. There is an integration of the previously tested function canBuy.

5.1.4.1 White Box Testing

Basis Path Testing

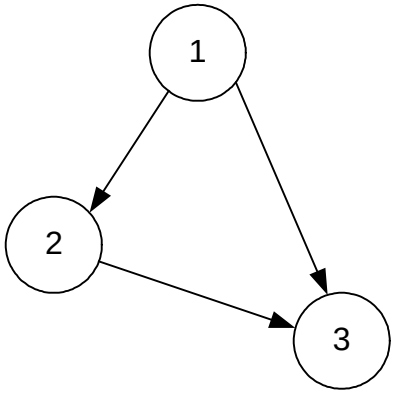


Path 1	1-3
Path 2	1-2-3

Path 1	1-3
Variables	Player cannot buy.
Expected result	Nothing.

Path 2	1-2-3
Variables	Player can buy.
Expected result	Buy the property. Debit the price of the property . Set as newly owned with the function newlyOwn.

Path diagram for function: buyProperty



5.1.5 Function canBuild

Tests will be conducted on the canBuild function, which is supposed to return true if the player can build on a specified property, and false if cannot build on a specified property. The function take a cellId argument, which is the cell id of the cell the player is interested in building on.

5.1.5.1 Black Box Testing

Every test case will be on a cell, and if the cell is a street then the player is the owner, the balance is sufficient, there are less then four hotels, the player own the whole district and the property is not mortgaged. If one of those conditions is false, then every tests cases are supposed to be false.

Test Case 1: Pass an argument that is under the lower bound for the variable cellId.
(cellId = -1)

Test Case 2: Pass an argument that is on the lower bound for the variable cellId.
(cellId = 0)

Test Case 3: Pass an argument that is between the bound.
(cellId = 1)

Test Case 4: Pass an argument that is exactly on the upper bound of cellId.
(cellId = 39)

Test Case 5: Pass an argument that is more than the upper bound of cellId.
(cellId = 40)

The expected result is the answer to the question cans the player build on this property.

Tester name	Patrice Michaud			Test date	November 29, 2003
Class name	Player	Method name	canBuild	File name	Player.cls

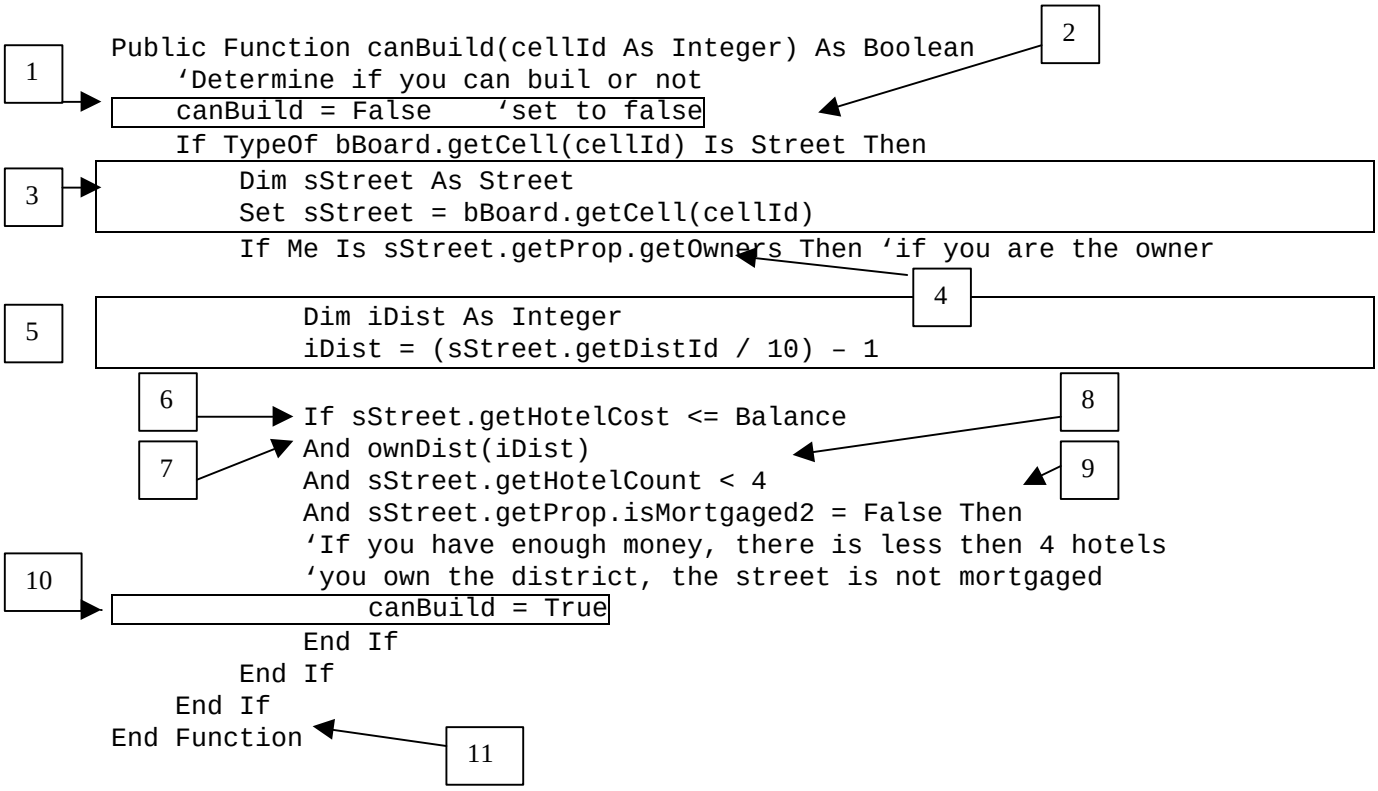
Variable name	cellId	Lower bound	0	Upper bound:	39

less than lower bound	Value: -1
on lower bound	Value: 0
between the bounds	Value: 1
on the upper bound	Value: 39
greater than upper bound	Value: 40

Test case	less than lower bound	on lower bound	between the bounds	on the upper bound	greater than upper bound
Expected output	false	false	true	true	false
Actual output	false	false	true	true	false
Bug found?	No	No	No	No	No

5.1.5.2 White Box Testing

Basis Path Testing



Path 1	1-2-11
Path 2	1-2-3-4-11
Path 3	1-2-3-4-5-6-11
Path 4	1-2-3-4-5-6-7-11
Path 5	1-2-3-4-5-6-7-8-11
Path 6	1-2-3-4-5-6-7-8-9-11
Path 7	1-2-3-4-5-6-7-8-9-10-11

Path 1	1-2-11
Variables	cellId is not a street.
Expected result	Cannot build.

Path 2	1-2-3-4-11
Variables	cellId is a street. Player is not the owner.
Expected result	Cannot build.

Path 3	1-2-3-4-5-6-11

Variables	cellId is a street. Player is the owner. Player balance is less then the cost of a hotel.
Expected result	Cannot build.

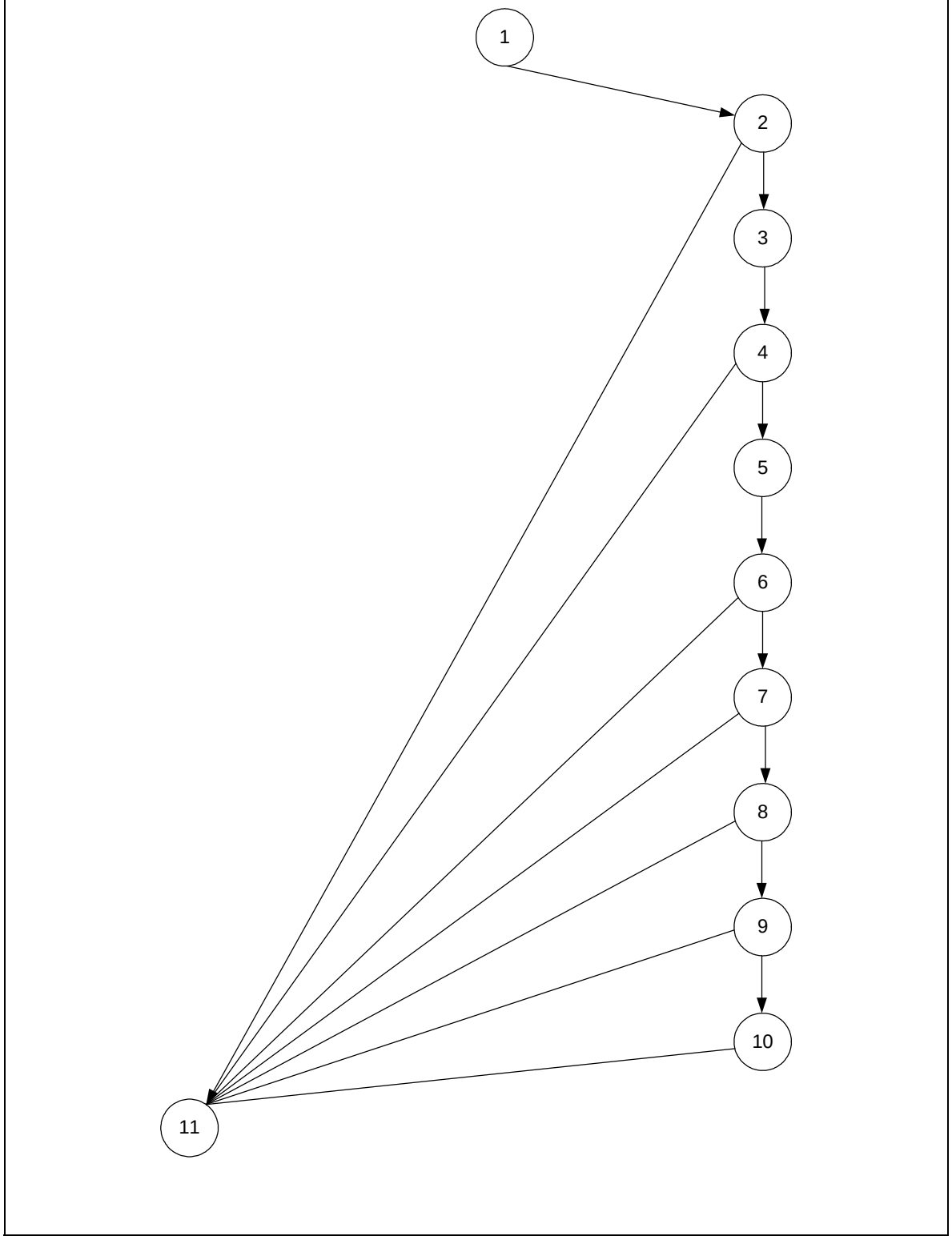
Path 4	1-2-3-4-5-6-7-11
Variables	cellId is a street. Player is the owner. Player balance is more then the cost of a hotel. Player doesn't own the whole district.
Expected result	Cannot build.

Path 5	1-2-3-4-5-6-7-8-11
Variables	cellId is a street. Player is the owner. Player balance is more then the cost of a hotel. Player does own the whole district. Street already has four hotels.
Expected result	Cannot build.

Path 6	1-2-3-4-5-6-7-8-9-11
Variables	cellId is a street. Player is the owner. Player balance is more then the cost of a hotel. Player does own the whole district. Street has less then four hotels. Street is mortgaged.
Expected result	Cannot build.

Path 7	1-2-3-4-5-6-7-8-9-10-11
Variables	cellId is a street. Player is the owner. Player balance is more then the cost of a hotel. Player does own the whole district. Street has less then four hotels. Street is not mortgaged.
Expected result	Can build a hotel on the street.

Path diagram for function: canBuild



5.1.6 Function doTrade

Test will be conduct on the doTrade function, which is supposed to trade a property for a certain amount of money. The function takes a cellId argument, which is the cell id of the cell the player is interested in trading for. The function also takes an amount argument, which is the amount the players agreed on for the property.

5.1.6.1 Black Box Testing

Every test case takes cellId and amount as arguments.

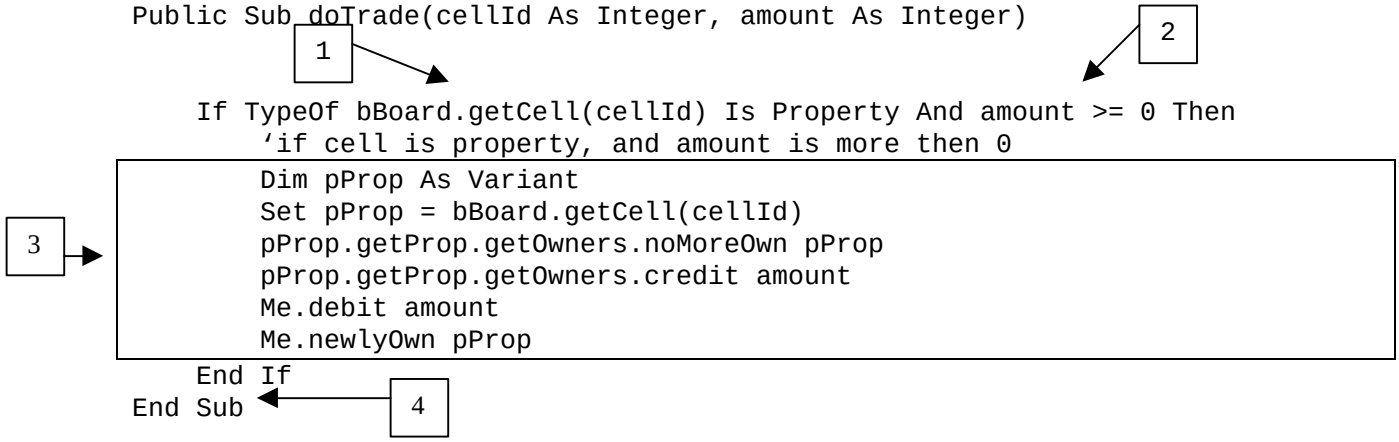
Name of tester	Patrice Michaud			Test date	December 1, 2003	
Class name	Player	Name of method	doTrade	Filename	Player.cls	
v: (1st variable name)	cellId	v: Lower bound	0	v: Upper bound	39	
w: (2nd variable name)	amount	w: Lower bound	0	w: Upper bound	n	

v1: 1st variable less than lower bound	Value v1: -1
v2: 1st variable on lower bound	Value v2: 0
v3: 1st variable between the bounds	Value v3: 1
v4: 1st variable on the upper bound	Value v4: 39
v5: 1st variable greater than upper bound	Value v5: 40
w1: 2nd variable less than lower bound	Value w1: -1
w2: 2nd variable on lower bound	Value w2: 0
w3: 2nd variable between the bounds	Value w3: 5

Variable w1					
Test case	v1 ~ w1	v2 ~ w1	v3 ~ w1	v4 ~ w1	v5 ~ w1
Expected output	nothing	nothing	nothing	nothing	nothing
Actual output	nothing	nothing	nothing	nothing	nothing
Bug found?	No	No	No	No	No
Variable w2					
Test case	v1 ~ w2	v2 ~ w2	v3 ~ w2	v4 ~ w2	v5 ~ w2
Expected output	nothing	nothing	Trade the cell for 0$	Trade the cell for 0$	nothing
Actual output	nothing	nothing	Trade the cell for 0$	Trade the cell for 0$	nothing
Bug found?	No	No	No	No	No
Variable w3					
Test case	v1 ~ w3	v2 ~ w3	v3 ~ w3	v4 ~ w3	v5 ~ w3
Expected output	nothing	nothing	Trade the cell for 5$	Trade the cell for 5$	nothing
Actual output	nothing	nothing	Trade the cell for 5$	Trade the cell for 5$	nothing
Bug found?	No	No	No	No	No

5.1.6.2 White Box Testing

Basis Path Testing



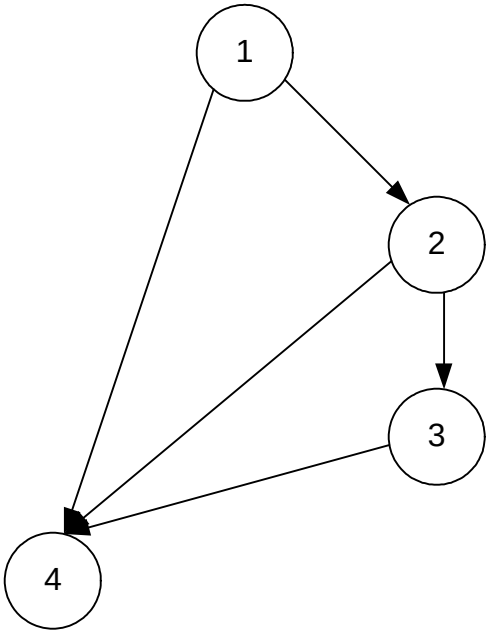
Path 1	1-4
Path 2	1-2-4
Path 3	1-2-3-4

Path 1	1-4
Variables	cellId is not a property.
Expected result	No trade.

Path 2	1-2-4
Variables	cellId is a property. Amount is less then 0.
Expected result	No trade.

Path 3	1-2-3-4
Variables	cellId is a property. Amount is more then 0.
Expected result	Trade *amount* of dollars for the property. Credit amount to the ex-owner. Set no more own for the ex-owner. Set player newly own. Debit amount for the player.

Path diagram for function: doTrade



5.2 Integration Testing

Integration Testing is a type of testing in which software is combined and tested to confirm that they interact according to their requirements. Integration testing can continue progressively until the entire system has been integrated.

5.2.1 The testing order

Integration testing begins after each unit is tested individually. To save time, we do integration testing in the following order:

- Firstly, we divide the integration into several steps, and we call each of them a specific integration test.
- Secondly, for each integration test, we design several test cases. In each test case, exactly one new component is analyzed.
- Thirdly, for each required component we test the integration between one and another.

For example, we first test the player class followed by the Game Start window.

5.2.1.1 Integration test component and order

Consider the implementation and unit testing order below. We divide the integration testing into the following six components:

- Game Start Window;
- Main Window;
- JFL Window;
- Cell Info Window;
- Trade Window;
- Game End Window.

5.2.1.2 Prerequisite unit testing for each integration test

Integration testing has to be done after a sufficient amount of unit tests have been performed. For each integration test, we list the prerequisite unit tests.

1. Game Start Window
   - Player

   For the game start window we do not test all the functions of the player class, only the constructors require testing.
2. Main Window
   - Board, Player, Dice, JFLDeck, JFLCard, Cell, Go, Go to jail, Olympic park, Jail, JFL, Income Tax, Luxury Tax, Property, Street, Utility, and Metro.

   All the components should be tested before doing main window testing.
3. JFL Window
   - JFLDeck and JFLCard

   The functions of both these classes should be tested.
4. Cell Info Window
   - Street, Utility, and Metro.

   All functions of these property cells should be tested before doing “cell info window”, since cell info window does not only display cell information. It also displays some other functions like trade, build hotel.
5. Trade Window
   - Player, board, Street, Utility, Metro.
6. Game End Window

Note, for the end game testing, there is no prerequisite unit testing. Logically, it should be done after all other integration testing is done.

5.2.2 Test method

We will perform the integration testing by using the Sandwich method because this method is a combination of

bottom-up and top-down integration testing. We can start integration as early as possible in the software development phase.
For integration testing, the only way of testing integration for each case is to add one component and test it to see if it works with other existing components.

5.2.3 Game Start Window

Only two cases are designed for the game-starting window.

Test Case 1	Initialize the window
Test Case Description	To test if the window can initialize normally. This test case should be done when the icon of the executable file is clicked.
Test result	Ok.

Test Case 2	Add the player objects
Test Case Description	After selecting tokens and input names, click the add player button to test if the player object can work well in this window.
Test result	Ok.

Test Case 3	Call the main window
Test Case Description	Click the “let’s start game” button to test if this window can call the main window.
Test result	Ok.

5.2.4 Main Window

There are three types of the test cases. Type one is an integrated class object. The second type is to call the other windows. The third type is closing the window. In all, there are 22 test cases for the main window test, which is the main playing area or board.

Test Case 1	Initialize the main window
Test Case Description	Test by: • Click “let’s start game” of the start game window to see if the main window can be initialized normally.
Test result	Ok.

Test Case 2	Add the board object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 3	Add the player object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 4	Add dice object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 5	Add the JFLCard object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 6	Add the JFLDeck object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 8	Add Go cell object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 9	Add Go To Jail cell object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 10	Add Olympic park cell object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 11	Add the Gail cell object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 12	Add JFL cell object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 13	Add the Income Tax cell object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 14	Add the Luxury Tax cell object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 15	Add street cell object
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 16	Add utility cell object
Test Case Description	This test should be done automatically when the main window loads.
Test result	

Test Case 17	Add Metro cell object
Test Case Description	This test should be done automatically when the main window loads.
Expected Results	

Test Case 18	Call JFL Window
Test Case Description	This test should be done automatically when the main window loads.
Test result	Ok.

Test Case 19	Call Cell Info Window
Test Case Description	Click following cells: • Street

	• Utilities • Metros Before rolling dices or after rolling dices.
Test result	Ok.

Test Case 20	Call Trade Window
Test Case Description	Click each of the following cells: • An owned cell: owner is himself (should not call trade window) • An owned cell: owner is other player(should call trade window if the player has enough money.) • An unowned cell. (Should not call trade window.)
Test result	Ok.

Test Case 21	Call Game End Window
Test Case Description	This test should be done automatically if there is a winner.
Test result	Ok.

Test Case 23	Close this window
Test Case Description	Testing by: • Click exit in the file menu. • Click to close the window.
Test result	Ok.

5.2.5 JFL Window

Test Case 1	Initialize the window
Test Case Description	When a player lands on JFL cell, this window should be displayed.
Test result	Ok.

Test Case 2	Add JFLDeck object
Test Case Description	This test should be done automatically when the window displays.
Test result	Ok.

Test Case 1	Add JFLCard
Test Case Description	This test should be done automatically when the window displays.
Test result	Ok.

Test Case 2	Call the main window and closing this window
Test Case Description	Test by: • Click the “ok” button; • Close the window directly; or • Closing automatically when the player is a computer.
Test result	Ok.

5.2.6 Cell Info Window

Note that only one cell object can be added once.

Test Case 1	Initial this window
Test Case Description	When a property cell (utility, metro, and street) is clicked.
Test result	Ok.

Test Case 2	Add street cell object
Test Case Description	This case should be done automatically when the window is displayed
Test result	Ok.

Test Case 3	Add utility cell object
Test Case Description	This case should be done automatically when the window is displayed
Test result	Ok.

Test Case 4	Add Metro cell object
Test Case Description	This case should be done automatically when the window is displayed
Test result	Ok.

Test Case 5	Call JFL Window
Test Case Description	This case should be done automatically when the window is displayed
Test result	Ok.

Test Case 6	Close this window and back to the main window.
Test Case Description	Tested by: • Click “ok” button, • Click to close the window.
Test result	Ok.

5.2.7 Trade Window

This window is initialized with the player object and the object that is being traded, either a street, metro or utility. Once the trade has been completed, the window is closed and control is returned to main window. Only one type of object can be traded at a time.

Test Case 1	Initial the trade window
Test Case Description	Tested when a property cell that is owned by another player is clicked.
Test result	Ok.

Test Case 2	Add player objects
Test Case Description	The initiator and the property owner are players should be automatically added.
Test result	Ok.

Test Case 3	Add street cell object
Test Case Description	Click a street cell(that is owned by another player).
Test result	Ok.

Test Case 4	Add utility cell object
Test Case Description	Click a street cell(that is owned by another player).
Test result	Ok.

Test Case 5	Add Metro cell object
Test Case Description	Click a street cell(that is owned by another player).
Test result	Ok.

Test Case 6	Close this window and back to the main window.
Test Case Description	Tested by:

	• Click “ok” button, • Click to close the window.
Test result	Ok.

5.2.8 Game End Window

To test the game end window, the only option that can be selected is the “start another game”, which starts a new game. The end game window closes and the start game window opens.

Test Case 1	Close this window and back to the game start window.
Test Case Description	Test by: Click the “start another game” button
Test result	Ok.

Test Case 2	Close this window and game over.
Test Case Description	Test by: Click “end game” button.
Test result	Ok.

5.3 Function Testing

This section is concerned with testing the functions (or requirements) of the software. This is a critical aspect of the testing effort, as it ensures that the software meets the requirements, and thus ensures acceptance by the users. For completeness, each requirement should be associated with a set of test cases, some with valid data, and some with invalid data. Despite time restrictions, we have included all the major product functions as well as test cases for each one. In fact, one of the benefits of having this section as complete as possible is that the implementation team can consult this list of test cases to ensure that they have properly implemented the functions, and that the software works both in the normal cases and exceptional cases.

The following sections are devoted to the major functions that were selected as testing targets. Each section lists and describes the different test cases that are important to check.

5.3.1 Start Game

Test Case	Add a Human Player
Test Case Description	Add a human player to the list of players – normal case.
Input (Steps to produce test)	1. Open the Start Game window 2. Enter a player name 3. Select the Human button 4. Select a token 5. Click on Add Player
Output (Expected Results)	• The player’s name is added to the list of players (without a # sign) • The player’s token appears beside the player’s name in the list of players • The selected token is disabled (disappears) from the available tokens

Test Case	Add a Computer player
Test Case Description	Add a computer player to the list of players – normal case.
Input (Steps to produce test)	1. Open the Start Game window 2. Enter a player name 3. Select the Computer button 4. Select a token 5. Click on Add Player
Output (Expected Results)	• The player’s name is added to the list of players • A # sign appears beside the player’s name, indicating that it’s a computer player

	• The player's token appears beside the player's name in the list of players • The selected token is disabled (disappears) from the available tokens

Test Case	Delete a Human Player
Test Case Description	Delete a human player from the list of players – normal case
Input (Steps to produce test)	1. Open the Start Game window 2. Select a human player from the list of players 3. Click on Delete Player
Output (Expected Results)	• The player's name disappears from the list of players • The player's token disappears from the list of players • The player's token is enabled (re-appears) in the available tokens

Test Case	Delete a Computer Player
Test Case Description	Delete a computer player from the list of players – normal case
Input (Steps to produce test)	1. Open the Start Game window 2. Select a computer player (name starts with #) from the list of players 3. Click on Delete Player
Output (Expected Results)	• The player's name disappears from the list of players • The player's token disappears from the list of players • The player's token is enabled (re-appears) in the available tokens

Test Case	Blank Name
Test Case Description	Add a player with a blank name – abnormal case
Input (Steps to produce test)	1. Open the Start Game window 2. Leave the player name blank (or clear it if it has some text) 3. Click on add player
Output (Expected Results)	• An error message appears indicating that a name must be provided

Test Case	Blank Token
Test Case Description	Add a player with a blank name – abnormal case
Input (Steps to produce test)	1. Open the Start Game window 2. Enter a player name 3. Select a player type 4. Do not select a token 5. Click on add player
Output (Expected Results)	• An error message appears indicating that a token must be selected

Test Case	Duplicate Names
Test Case Description	Add a player with a name that already exists – abnormal case
Input (Steps to produce test)	1. Open the Start Game window 2. Enter the name of a player that has already been added to the list of players 3. Click on add player
Output (Expected Results)	• An error message appears indicating that the player has already been added

Test Case	Duplicate Tokens
Test Case Description	Two players cannot have the same token
Input (Steps to produce test)	1. Open the Start Game window 2. Add a player by entering a name, type and token, then click add player 3. Enter a different name

	4. Do not change the player type 5. Do not click on any token
Output (Expected Results)	• An error message appears indicating that a token must be selected
Comments	In this test case, we are attempting to have two players with the same token. This could happen (would be a bug) if after selecting a token and adding a player, we add another player without selecting a token. Perhaps the system remembers the token selection done previously and erroneously adds another player with the same token.

Test Case	Cancel Start Game
Test Case Description	A user decides not to start the game
Input (Steps to produce test)	1. Open the Start Game window 2. Enter the name, type and token of a player 3. Click on the Quit or Cancel button
Output (Expected Results)	• The Start Game window disappears • The application stops executing with no errors

Test Case	Start Game with 0 Players
Test Case Description	Start the game with no players added – abnormal case
Input (Steps to produce test)	1. Open the Start Game window 2. Click on Let's Start
Output (Expected Results)	• An error message appears indicating that there must be at least 2 players

Test Case	Start Game with 1 Player
Test Case Description	Start the game with 1 player – abnormal case
Input (Steps to produce test)	1. Open the Start Game window 2. Enter a player name 3. Select a player type 4. Select a player token 5. Click on Let's Start
Output (Expected Results)	• The added player's info appears in the player list • An error message appears indicating that there must be at least 2 players

Test Case	Start Game with 2 Players
Test Case Description	Start the game with 2 players added.
Input (Steps to produce test)	1. Open the Start Game window 2. Enter the first player's name, type and token 3. Enter the second player's name, type and token 4. Click on Let's Start
Output (Expected Results)	• The Start Game window disappears • The Game Board window appears • The info in the player's list matches the players' info that was entered

Test Case	Start Game with 8 Players
Test Case Description	Start the game with 8 players added.
Input (Steps to produce test)	1. Open the Start Game window 2. Enter the name type and token of 8 players 3. Click on Let's Start
Output (Expected Results)	• The Start Game window disappears • The Game Board window appears • The info in the player's list matches the players' info that was entered

Test Case	Add more than 8 Players
Test Case Description	Add more than 8 players – abnormal case
Input (Steps to produce test)	1. Open the Start Game window 2. Enter the name type and token of 8 players 3. Enter the name type and token of a 9th player 4. Click on Add Player
Output (Expected Results)	• An error message appears indicating that a 9th player cannot be added • The player is not added to the list of players • The token that was selected does not disappear

Test Case	Random Player Order
Test Case Description	When the game is started, the order of the players is randomized
Input (Steps to produce test)	1. Open the Start Game window 2. Enter the name type and token of 2-8 players 3. Click on Let's Start 4. Take note of the player's order 5. Select Exit from the File menu 6. Redo steps 1-4, following the exact same steps done previously 7. Take note of the player's order
Output (Expected Results)	• The Start Game window disappears • The Game Board window appears • The info in the player's list matches the players' info that was entered • The order of the players in the player's list is randomized, and is not the same as the order in which the players were added. • If the application is closed and restarted, the way the player's order is randomized is not the same
Notes	Here, we test the randomization of the player's order by testing it once, then re-starting the application and testing it again. This is important to ensure that the random (pseudo-random) function is working properly, and does not behave in a predictable manner.

Test Case	Standard Cash Distribution
Test Case Description	When a game is started, each player is given 1500$
Input (Steps to produce test)	1. Open the Start Game window 2. Enter the name type and token of 2-8 players 3. Click on Let's Start 4. Take note of the balance of each player
Output (Expected Results)	• Each player has exactly 1500$

Test Case	JFL Deck Shuffled
Test Case Description	When a game is started, the JFL deck is shuffled properly
Input (Steps to produce test)	1. Open the Start Game window 2. Enter the name type and token of 2-8 players 3. Click on Let's Start 4. Play the game, taking note of the sequence of JFL cards that are withdrawn 5. Select Exit from the File menu 6. Redo steps 1-3, following the exact same steps done previously 7. Play the game, taking note of the sequence of JFL cards that are withdrawn
Output (Expected Results)	• The sequence of JFL cards that are withdrawn is random and changes every time the application is restarted.

Notes	Here, we test the randomization of the deck shuffling by testing it once, then re-starting the application and testing it again. This is important to ensure that the random (pseudo-random) function is working properly, and does not behave in a predictable manner.

5.3.2 Roll Dice

Test Case	Dice rolls are random
Test Case Description	The values of the dice rolls are truly random, and not predictable.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, taking note of the values of the dice rolls (individually for each die) 3. Exit the game. 4. Start another game in the exact same manner as done previously. 5. Take note of the values of the dice rolls.
Output (Expected Results)	• The values of the dice rolls are truly random and the sequence is not repeated.

Test Case	If Roll Doubles, Roll again
Test Case Description	A player who rolls doubles is allowed to roll again.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a player rolls a double on the dice. 3. Try to end turn. 4. Roll the dice again. 5. End Turn.
Output (Expected Results)	• The player who rolled doubles is not allowed to end turn until he has rolled the dice again.

Test Case	Token is moved properly
Test Case Description	To ensure the token is moved properly on the cells.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns taking note of the values of the dice rolls and the number of steps the player's token is moved.
Output (Expected Results)	• The player's token is moved the number of steps according to the value of the dice rolls. • No other token is moved (other player's tokens)

5.3.3 Pass Go

Test Case	Pass Go, Collect 200$
Test Case Description	If the player has passed the Go square, the player collects 200$.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a player passes the Go square (makes a full turn around the board)
Output (Expected Results)	• The player who passes Go collects 200$

Test Case	GoToJail does not collect 200$
Test Case Description	If the player is sent to jail, the player does not collect 200$ for passing Go.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a player lands on the GoToJail cell.
Output (Expected Results)	• The player's token is moved to the Jail cell • The player does not collect 200$.

5.3.4 Pay Rent

Test Case	Land on other-player-owned property, pay rent
Test Case Description	If a player lands on a property owned by another player, he pays rent to the owner.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, buying all the properties that the players land on. 3. Keep playing until a player lands on a property owned by another player. 4. Take note of the change in balance of the current player and the owner of the cell. 5. Click on the cell that the player has landed on, and take note of the rent amount.
Output (Expected Results)	• The balance of player who lands on the cell is decreased by the rent amount. • The balance of the owner of the cell is increased by the rent amount.

Test Case	Land on un-owned property, don't pay rent
Test Case Description	If a player lands on an un-owned property, the player does not pay rent.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a player lands on an un-owned property.
Output (Expected Results)	• The player's balance remains the same. • The player is given the option to buy the property.

Test Case	Land on property owned by player, don't pay rent
Test Case Description	If a player lands on a property that he owns himself, he doesn't pay rent.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy all the properties they land on. 3. Keep playing until a player lands on a cell that he owns himself.
Output (Expected Results)	• The player does not pay rent. His balance remains the same.

Test Case	Rent amount paid for 0 hotels
Test Case Description	If a player lands on a property owned by another player, with 0 hotels, he pays rent.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, buying all the properties that the players land on. 3. Keep playing until a player lands on a property owned by another player. 4. Take note of the change in balance of the current player and the owner of the cell. 5. Click on the cell that the player has landed on, and take note of the rent amount.
Output (Expected Results)	• The balance of player who lands on the cell is decreased by the rent amount for 0 hotels. • The balance of the owner of the cell is increased by the rent amount for 0 hotels.

Test Case	Rent amount paid for 1 hotel
Test Case Description	If a player lands on a property owned by another player, with 1 hotel, he pays rent.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, buying all the properties that the players land on. 3. Start trading properties to force a player into owning a whole district. 4. Build 1 hotel on each property of that district. 5. Keep playing until a player lands on a one of the properties in that district. 6. Take note of the change in balance of the current player and the owner of the cell. 7. Click on the cell that the player has landed on, and take note of the rent amount.
Output (Expected Results)	• The balance of player who lands on the cell is decreased by the rent amount for 1 hotel.

	• The balance of the owner of the cell is increased by the rent amount for 1 hotel.

Test Case	Rent amount paid for 2 hotels
Test Case Description	If a player lands on a property owned by another player, with 2 hotels, he pays rent.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, buying all the properties that the players land on. 3. Start trading properties to force a player into owning a whole district. 4. Build 2 hotels on each property of that district. 5. Keep playing until a player lands on a one of the properties in that district. 6. Take note of the change in balance of the current player and the owner of the cell. 7. Click on the cell that the player has landed on, and take note of the rent amount.
Output (Expected Results)	• The balance of player who lands on the cell is decreased by the rent amount for 2 hotels. • The balance of the owner of the cell is increased by the rent amount for 2 hotels.

Test Case	Rent amount paid for 3 hotels
Test Case Description	If a player lands on a property owned by another player, with 3 hotels, he pays rent.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, buying all the properties that the players land on. 3. Start trading properties to force a player into owning a whole district. 4. Build 3 hotels on each property of that district. 5. Keep playing until a player lands on a one of the properties in that district. 6. Take note of the change in balance of the current player and the owner of the cell. 7. Click on the cell that the player has landed on, and take note of the rent amount.
Output (Expected Results)	• The balance of player who lands on the cell is decreased by the rent amount for 3 hotels. • The balance of the owner of the cell is increased by the rent amount for 3 hotels.

Test Case	Rent amount paid for 4 hotels
Test Case Description	If a player lands on a property owned by another player, with 4 hotels, he pays rent.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, buying all the properties that the players land on. 3. Start trading properties to force a player into owning a whole district. 4. Build 4 hotels on each property of that district. 5. Keep playing until a player lands on a one of the properties in that district. 6. Take note of the change in balance of the current player and the owner of the cell. 7. Click on the cell that the player has landed on, and take note of the rent amount.
Output (Expected Results)	• The balance of player who lands on the cell is decreased by the rent amount for 4 hotels. • The balance of the owner of the cell is increased by the rent amount for 4 hotels.

Test Case	Land on mortgaged property, don't pay rent
Test Case Description	If a player lands on a mortgaged property, he doesn't pay rent.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, buying all the properties that the players land on. 3. Start trading properties to force a player into owning a whole district. 4. Mortgage each property in that district. 5. Keep playing until a player lands on a one of the properties in that district. 6. Take note of the balance of the current player and the owner of the cell.

Output (Expected Results)	• The balance of the player who lands on the cell is not changed. • The balance of the owner of the cell is not changed.

5.3.5 Buy Property

Test Case	Buy a property, cash is deducted, owner status is changed
Test Case Description	When a player buys a property, the price of the property is deducted and the owner status token is updated.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, until a player lands on an un-owned property. 3. Take note of the current balance of the player. 4. Take note of the price of the property. 5. Buy the property. 6. Take note of the ending balance of the player.
Output (Expected Results)	• The balance of the player is decreased by the price amount of the property. • The token of the player is displayed at the top left corner of the cell, indicating that he owns this property.

Test Case	Can only buy property after rolling dice
Test Case Description	To ensure that a player can only buy a property after having rolled the dice.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, until a player lands on an un-owned property. 3. Do not buy the property. 4. Play the other player's turns, until the current turn comes back to this player. 5. Before rolling the dice, click on the cell that the player is on.
Output (Expected Results)	• The Cell Info window pops-up and does not have a "Buy It" button. • The player is not allowed to buy a property, unless he has already rolled the dice.

Test Case	Can only buy property landed on
Test Case Description	To ensure that a player can only buy a property he "lands" on (ie: has rolled the dice and landed on that property).
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, until a player lands on an un-owned property. 3. Do not buy the property. 4. Play the other player's turns, until the current turn comes back to this player. 5. Before rolling the dice, click on the cell that the player is on. 6. Take note of the options on the Cell Info pop-up window. 7. Roll the dice 8. Click on an un-owned cell, other than the one the player has landed on. 9. Take note of the options on the Cell Info pop-up window.
Output (Expected Results)	• In both cases, the Cell Info window pops-up and does not have a "Buy It" button. • The player is not allowed to buy a property, unless he has already rolled the dice and landed on that property.

Test Case	Can only buy un-owned property
Test Case Description	A player cannot buy an un-owned property.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy all the properties they land on. 3. Keep playing until a player lands on a property owned by another player. 4. Click on the cell the player has landed on. 5. Take note of the options on the Cell Info pop-up window. 6. Click on another cell that is owned by another player. 7. Take note of the options on the Cell Info pop-up window.

Output (Expected Results)	• In both cases, the Cell Info window pops-up and does not have a "Buy It" button. • The player is not allowed to buy a property that is owned by another player.

Test Case	Can only buy property if enough money
Test Case Description	A player cannot buy a property if he does not have enough money to buy it.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Keep playing until a player lands on an un-owned property, but does not have enough money to buy it.
Output (Expected Results)	• The player is not allowed to buy the property, since he would then have a negative balance, and might have to declare bankruptcy or mortgage properties.

5.3.6 Build/Sell Hotel

Test Case	Build 1 hotel
Test Case Description	Build 1 hotel on a property owned by player
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. When it is that player's turn to play, click on one of the cells in the district. 5. Buy 1 hotel. 6. Take note of the change in balance of the player. 7. Take note of the status of the cell.
Output (Expected Results)	• The player's balance is decreased by the cost of the hotel. • A hotel icon is displayed at the top right corner of the cell.

Test Case	Build 2 hotels
Test Case Description	Build 2 hotels on a property owned by player
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. When it is that player's turn to play, click on one of the cells in the district. 5. Buy 2 hotels. 6. Take note of the change in balance of the player. 7. Take note of the status of the cell.
Output (Expected Results)	• The player's balance is decreased by the cost of 2 hotels. • 2 hotel icons are displayed at the top right corner of the cell.

Test Case	Build 3 hotels
Test Case Description	Build 3 hotels on a property owned by player
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. When it is that player's turn to play, click on one of the cells in the district. 5. Buy 3 hotels. 6. Take note of the change in balance of the player. 7. Take note of the status of the cell.
Output (Expected Results)	• The player's balance is decreased by the cost of 3 hotels. • 3 hotel icons are displayed at the top right corner of the cell.

Test Case	Build 4 hotels
Test Case Description	Build 4 hotels on a property owned by player
Input	1. Start the game with 2-8 players.

(Steps to produce test)	2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. When it is that player's turn to play, click on one of the cells in the district. 5. Buy 4 hotels. 6. Take note of the change in balance of the player. 7. Take note of the status of the cell.
Output (Expected Results)	• The player's balance is decreased by the cost of 4 hotels. • 4 hotel icons are displayed at the top right corner of the cell.

Test Case	Build more than 4 hotels
Test Case Description	Build more than 4 hotels on a property owned by player
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. When it is that player's turn to play, click on one of the cells in the district. 5. Buy 4 hotels. 6. Take note of the status of the buy hotel button.
Output (Expected Results)	• After having already purchased 4 hotels, the buy hotel button is disabled, preventing the user from buying more than 4 hotels.

Test Case	Build a hotel on a property owned by another player
Test Case Description	To ensure that a player cannot build hotels on a property owned by another player.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. When it is one of the other players' turn to play, click on one of the cells in the district that is owned by the other player. 5. Take note of the options given to the player.
Output (Expected Results)	• The player is not allowed to build a hotel on that property, since another player owns it.

Test Case	Build a hotel with not enough money
Test Case Description	To ensure that a player is not allowed to build a hotel if he does not have enough money to build it.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. Try to trade for small amount of money to force that player's balance to be as low as possible (but not negative). 4. When it is that player's turn to player, click on one of the cells in the district he owns, and try to build a hotel.
Output (Expected Results)	• The player is not allowed to build a hotel since he does not have enough money to build it.

Test Case	Sell 1 hotel
Test Case Description	Sell 1 hotel on a property owned by player
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. When it is that player's turn to play, click on one of the cells in the district. 5. Buy 4 hotels. 6. On the player's next turn, click on the same cell, and click on the Sell Hotel

	button once. 7. Take note of the change in balance of the player.
Output (Expected Results)	• The player's balance is increased by the cost of the hotel. • The number of hotel icons appearing on the cell is decreased to 3.

Test Case	Sell 2 hotels
Test Case Description	Sell 2 hotels on a property owned by player
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. When it is that player's turn to play, click on one of the cells in the district. 5. Buy 4 hotels. 6. On the player's next turn, click on the same cell, and click on the Sell Hotel button twice. 7. Take note of the change in balance of the player.
Output (Expected Results)	• The player's balance is increased by the cost of 2 hotels. • The number of hotel icons appearing on the cell is decreased to 2.

Test Case	Sell 3 hotels
Test Case Description	Sell 3 hotels on a property owned by player
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. When it is that player's turn to play, click on one of the cells in the district. 5. Buy 4 hotels. 6. On the player's next turn, click on the same cell, and click on the Sell Hotel button three times. 7. Take note of the change in balance of the player.
Output (Expected Results)	• The player's balance is increased by the cost of 3 hotels. • The number of hotel icons appearing on the cell is decreased to 1.

Test Case	Sell 4 hotels
Test Case Description	Sell 4 hotels on a property owned by player
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. When it is that player's turn to play, click on one of the cells in the district. 5. Buy 4 hotels. 6. On the player's next turn, click on the same cell, and click on the Sell Hotel button four times. 7. Take note of the change in balance of the player.
Output (Expected Results)	• The player's balance is increased by the cost of 4 hotels. • The number of hotel icons appearing on the cell is decreased to 0.

Test Case	Sell More than hotels built
Test Case Description	Sell 4 hotels on a property owned by player
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. When it is that player's turn to play, click on one of the cells in the district. 5. Buy 4 hotels. 6. On the player's next turn, click on the same cell, and click on the Sell Hotel

	button five times.
Output (Expected Results)	• The player's balance is increased by the cost of 4 hotels. • The number of hotel icons appearing on the cell is decreased to 0. • On the fifth attempt to sell a hotel, either the "Sell Hotel" button is disabled or an error message appears indicating that there are no more hotels to sell.

Test Case	Sell hotel on a property owned by another player
Test Case Description	To ensure that a player cannot sell hotels on a property owned by another player.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. When it is one of the other players' turn to play, click on one of the cells in the district that is owned by the other player. 5. Take note of the options given to the player.
Output (Expected Results)	• The player is not allowed to sell a hotel on that property, since another player owns it.

5.3.7 Mortgage/Un-Mortgage

Test Case	Mortgage a property – Valid Case
Test Case Description	Mortgage a property owned by player, whole district is owned, no hotels
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. Ensure that there are no hotels built on any of the properties in the district. 5. When it is that player's turn to play, click on one of the cells in the district. 6. Take note of the mortgage value of the property. 7. Click on the mortgage button to mortgage the property. 8. Take note of the change in balance of the player.
Output (Expected Results)	• The player is allowed to mortgage the property. • The player's balance is increased by the mortgage value. • The "Mortgaged" icon appears at the top left corner of the cell on the board.

Test Case	Mortgage a property owned by player, whole district is owned, other property in district has hotels
Test Case Description	Mortgage a property owned by player, whole district is owned, no hotels
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Start trading properties, until a player owns a district. 4. Build a hotel on one of the properties in the district. 5. When it is that player's turn to play, click on one of the cells in the district that does not have a hotel. 6. Click on the mortgage button to mortgage the property. 7. Take note of what happens next.
Output (Expected Results)	• The player is allowed to mortgage the property. • The player's balance is increased by the mortgage value. • The "Mortgaged" icon appears at the top left corner of the cell on the board.

Test Case	Mortgage a property, whole district is not owned
Test Case Description	Mortgage a property owned by player, whole district is not owned
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing the players to buy the properties they land on. 3. Pick a player to test with.

	4. When it is that player's turn to play, click on one of the cells that he owns, but make sure that he does not own the district. 5. Click on the mortgage button to mortgage the property.
Output (Expected Results)	• The player is allowed to mortgage the property. • The player's balance is increased by the mortgage value. • The "Mortgaged" icon appears at the top left corner of the cell on the board.

Test Case	Mortgage a property that has hotels
Test Case Description	Mortgage a property owned by player, but has hotels build on it.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, until a player lands on an un-owned property. 3. Buy that property. 4. Build a hotel on it. 5. Try to mortgage the property.
Output (Expected Results)	• The player is not allowed to mortgage the property. • Either the mortgage button is disabled or an error message appears indicating that all hotels must be sold before mortgaging the properties.

Test Case	Mortgage a property not owned
Test Case Description	Mortgage a property that is not owned by anyone.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, until a player lands on an un-owned property. 3. Try to mortgage the property.
Output (Expected Results)	• The player is not allowed to mortgage the property. • Either the mortgage button is disabled or an error message appears indicating that all hotels must be sold before mortgaging the properties.

Test Case	Mortgage a property owned by another player
Test Case Description	To ensure a player cannot mortgage a property owned by another player
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing players to buy the properties they land on. 3. Pick a player to test with. 4. Click on a cell that is owned by another player. 5. Try to mortgage the property.
Output (Expected Results)	• The player is not allowed to mortgage the property. • Either the mortgage button is disabled or an error message appears indicating that all hotels must be sold before mortgaging the properties.

Test Case	Un-Mortgage a property
Test Case Description	Un-Mortgage a property that is mortgaged, owned by player (pay mortgage+10%)
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Pick a player to test with. 3. Play turns until the player lands on an un-owned property. 4. Buy the property. 5. Take note of the mortgage value. 6. Mortgage the property. 7. Un-Mortgage the property.
Output (Expected Results)	• The player's balance is increased by the mortgage value + 10%.

Test Case	Un-Mortgage a property that is not mortgaged, owned by player
Test Case Description	Un-Mortgage a property that is mortgaged, owned by player (pay mortgage+10%)

Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Pick a player to test with. 3. Play turns until the player lands on an un-owned property. 4. Buy the property. 5. Try to Un-Mortgage the property.
Output (Expected Results)	• The player is not allowed to un-mortgage the property. • Either the un-mortgage button is disabled or an error message appears indicating that the property is not mortgaged.

Test Case	Un-Mortgage a property that is mortgaged, owned by another player
Test Case Description	To ensure that a player cannot un-mortgage a mortgaged property owned by another player.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing players to buy the properties they land on. 3. Force the players to mortgage the properties they own. 4. Pick a player to test with. 5. Click on a cell that is owned by another player. 6. Try to un-mortgage the property.
Output (Expected Results)	• The player is not allowed to un-mortgage the property. • Either the mortgage button is disabled or an error message appears indicating that a player cannot un-mortgage the property of another player.

Test Case	Un-Mortgage a property that is not mortgaged, owned by another player
Test Case Description	To ensure that a player cannot un-mortgage an un-mortgaged property owned by another player.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing players to buy the properties they land on. 3. Pick a player to test with. 4. Click on a cell that is owned by another player. 5. Try to un-mortgage the property.
Output (Expected Results)	• The player is not allowed to un-mortgage the property. • Either the mortgage button is disabled or an error message appears indicating that a player cannot un-mortgage the property of another player.

Test Case	Un-Mortgage a property that is not owned
Test Case Description	To ensure that a player cannot un-mortgage a property that is not owned.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play a few turns until a player lands on an un-owned property. 3. Try to un-mortgage the property.
Output (Expected Results)	• The player is not allowed to un-mortgage the property. • Either the mortgage button is disabled or an error message appears indicating that a player cannot un-mortgage a property that is not owned.

5.3.8 Tax

Test Case	Land on Income Tax, pay (smallest of 200$ and value of assets) to bank
Test Case Description	To ensure that the correct income tax is calculated and paid.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play a few turns until a player lands on the Income Tax cell. 3. Take note of the player's balance change.
Output (Expected Results)	• The player's balance is decreased by the smallest of 200$ and the value of his assets (properties + hotels + money)

Test Case	Land on Luxury Tax, pay 75$ to bank
Test Case Description	To ensure that the correct luxury tax is paid.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play a few turns until a player lands on the Luxury Tax cell. 3. Take note of the player's balance change.
Output (Expected Results)	• The player's balance is decreased by the amount of 75$.

5.3.9 JFL Cards

Test Case	Land on JFL Card, picks a JFL Card
Test Case Description	To ensure that a JFL card is picked when a player lands on a JFL cell.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a player lands on a JFL Cell. 3. Take note of what happens. 4. Repeat steps 2-3 until all the JFL cells on the board have had a player land on them.
Output (Expected Results)	• Every time a player lands on a JFL cell, a JFL card is displayed.

Test Case	JFL Cards are shuffled
Test Case Description	To ensure that the order that the JFL Cards appear in is randomized.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until several JFL Cards have been picked. 3. Take note of the sequence of JFL Cards. 4. Exit the game. 5. Restart the game with the parameters (number of players, players' names and tokens). 6. Play several turns until several JFL Cards have been picked. Take note of the sequence of JFL Cards.
Output (Expected Results)	• The sequence of JFL Cards should be different, after restarting the game, indicating that the sequence is truly randomized.

Test Case	Pay Card
Test Case Description	If the JFL card is a pay card, the player pays the amount indicated by the card.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a Pay JFL Card is picked. 3. Take note of the change in balance of the player.
Output (Expected Results)	• The player's balance is decreased by the amount indicated by the card.

Test Case	Collect Card
Test Case Description	If the JFL card is a collect card, the player collects the amount indicated by the card.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a Collect JFL Card is picked. 3. Take note of the change in balance of the player.
Output (Expected Results)	• The player's balance is increased by the amount indicated by the card.

Test Case	Advance Card
Test Case Description	If the JFL card is an advance card, token is moved, and appropriate action is taken when landed on cell.
Input	1. Start the game with 2-8 players.

(Steps to produce test)	2. Play several turns until an Advance JFL Card is picked. 3. Take note of the movement of the player's token.
Output (Expected Results)	• The player's token is moved forward according to the number of steps indicated by the card. • Appropriate action is taken according to the cell the player lands on.

Test Case	GoBack Card
Test Case Description	If the JFL card is a GoBack Card, token is moved, and appropriate action is taken when landed on cell
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a GoBack JFL Card is picked. 3. Take note of the movement of the player's token.
Output (Expected Results)	• The player's token is moved back according to the number of steps indicated by the card. • Appropriate action is taken according to the cell the player lands on.

Test Case	GOJFC Card
Test Case Description	If the JFL card is a GOJFC Card, the player keeps card and can use it later
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a GOJFC JFL Card is picked. 3. Keep playing until the player is sent to jail. 4. Click on the Use Get Out of Jail Free Card
Output (Expected Results)	• When a player obtains the GOJFC, an icon appears indicating that he has it. • When a player is in jail, has the GOJFC and uses it, he gets out of jail. • When a player uses the GOJFC, the icon that indicates that he has the card disappears.

Test Case	GOJFC Card removed from deck
Test Case Description	If a player gets the GOJFC Card, the card is removed from deck
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a GOJFC JFL Card is picked. 3. Do not used the GOJFC. Force the player who has it to keep the card. 4. Play several turns, taking note of the JFL cards that are picked. 5. Keep playing until the sequence of JFL cards repeats itself.
Output (Expected Results)	• The sequence of JFL Cards should not contain the GOJFC, since the player who picked it keeps it and does not use it.

Test Case	GoToJail Card
Test Case Description	GoToJail Card, player goes to jail
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a GoToJail JFL Card is picked.
Output (Expected Results)	• The player's token is moved to the jail status.

5.3.10 Jail

Test Case	Visiting Jail
Test Case Description	If a player land on the Jail cell, the player is "just visiting"
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a player's token lands on the Jail cell.
Output (Expected Results)	• The player's token should be placed in the "Just Visiting" portion of the jail cell.

Test Case	Roll Doubles 3 times
Test Case Description	If a player rolls doubles 3 times, goes to jail
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a player rolls doubles 3 times in a row (on the same turn).
Output (Expected Results)	• The player is sent to jail. • The player's token is moved to the "In Jail" portion of the jail cell. • The player is not allowed to roll the dice again.

Test Case	Land on GoToJail
Test Case Description	If a player lands on the GoToJail cell, he goes to jail
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a player lands on the GoToJail cell.
Output (Expected Results)	• The player is sent to jail. • The player's token is moved to the "In Jail" portion of the jail cell. • The player is not allowed to roll the dice again.

Test Case	In Jail Roll Doubles
Test Case Description	If a player is in jail and rolls doubles, he gets out of jail and does not roll again
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a player goes to jail and rolls doubles while in jail.
Output (Expected Results)	• The player gets out of jail. • The player's token is moved according to the value of the dice roll.

Test Case	Pay 50 Get Out of Jail
Test Case Description	If a player is in jail, he can pay 50$ and get out of jail on 1st, 2nd, or 3rd turn.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns until a player goes to jail. 3. On his next turn. 4. Click the "Pay 50 to get out of jail" button.
Output (Expected Results)	• The player gets out of jail. • The player's token is moved according to the value of the dice roll. • The player's balance is decreased by 50$

5.3.11 Trade

Test Case	Offer Trade, Accept Trade
Test Case Description	A player makes a trade offer to an owner, and the owner accepts the trade.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing players to buy the properties they land on. 3. Pick a player to test with. 4. On that player's turn to player, click on a property that is owned by another player. 5. Click the Trade button. 6. Enter a trade amount. 7. Click the "Accept Trade" button.
Output (Expected Results)	• The player's balance is decreased by the trade amount. • The owner's balance is increased by the trade amount. • The status of the cell (token at top-left corner) is updated … the new owner's token is displayed instead of the old owner's.

Test Case	Offer Trade, Reject Trade
Test Case Description	A player makes a trade offer to an owner, and the owner rejects the trade.
Input	1. Start the game with 2-8 players.

(Steps to produce test)	2. Play several turns, forcing players to buy the properties they land on. 3. Pick a player to test with. 4. On that player's turn to player, click on a property that is owned by another player. 5. Click the Trade button. 6. Enter a trade amount. 7. Click the "Reject Trade" button.
Output (Expected Results)	• The player and owner's balances are not changed. • The status of the cell (token at top-left corner) is not changed.

Test Case	Offer Trade, Counter Offer, Accept Trade
Test Case Description	A player makes a trade offer to an owner, the owner makes a counter offer and the player accepts the counter offer.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing players to buy the properties they land on. 3. Pick a player to test with. 4. On that player's turn to player, click on a property that is owned by another player. 5. Click the Trade button. 6. Enter a trade amount. 7. Click the "Counter Offer" button. 8. Enter a counter offer amount (different from the trade amount). 9. Click the "Accept Trade" button.
Output (Expected Results)	• The player's balance is decreased by the counter offer amount. • The owner's balance is increased by the counter offer amount. • The status of the cell (token at top-left corner) is updated … the new owner's token is displayed instead of the old owner's.

Test Case	Offer Trade, Counter Offer, Reject Trade
Test Case Description	A player makes a trade offer to an owner, the owner makes a counter offer and the player rejects the counter offer.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Play several turns, forcing players to buy the properties they land on. 3. Pick a player to test with. 4. On that player's turn to player, click on a property that is owned by another player. 5. Click the Trade button. 6. Enter a trade amount. 7. Click the "Counter Offer" button. 8. Enter a counter offer amount (different from the trade amount). 9. Click the "Reject Trade" button.
Output (Expected Results)	• The player and owner's balances are not changed. • The status of the cell (token at top-left corner) is not changed.

Test Case	Offer Trade on an un-owned property
Test Case Description	To ensure that a trade cannot be made on an un-owned property.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Pick a player to test with. 3. On that player's turn to player, click on a property that is un-owned. 4. Try to make a Trade.
Output (Expected Results)	• The player is not allowed to make a trade. • Either the "Trade" button is disabled or an error message appears indicating that

	a trade cannot be made on an un-owned property.

Test Case	Offer Trade on a self-owned property
Test Case Description	To ensure that a trade cannot be made on a property that is owned by the player himself.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Pick a player to test with. 3. Buy the first property that the player lands on. 4. Click on that property. 5. Try to make a Trade.
Output (Expected Results)	• The player is not allowed to make a trade. • Either the "Trade" button is disabled or an error message appears indicating that a trade cannot be made on a property owned by the player himself.

5.3.12 End Turn

Test Case	End Turn
Test Case Description	To ensure that when a player clicks end turn, the control is passed to the next player.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Take note of the sequence of the players. 3. Play several turns, until control is passed back to the first player, taking note of the "Current Player" indicator.
Output (Expected Results)	• The "Current Player" indicator indicates the player who's turn it is to play.

Test Case	End Turn before rolling dice
Test Case Description	To ensure that a player cannot end turn before rolling the dice.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Try to click the "End Turn" button before rolling the dice.
Output (Expected Results)	• The player is not allowed to pass control to the next player, until he has rolled the dice. • Either the "End Turn" button is disabled or an error message appears indicating that the player must roll the dice first.

Test Case	End Turn with a negative balance
Test Case Description	To ensure that a player cannot end turn if his balance is negative.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Pick a player to test with. 3. Play several turns, forcing all players except the test player to buy the properties they land on. 4. Keep playing turns, until the test player gets a negative balance. 5. Click "End Turn"
Output (Expected Results)	• The player is not allowed to pass control to the next player, until he brought his balance to be positive or declared bankruptcy. • Either the "End Turn" button is disabled or an error message appears indicating that the player must have a positive balance before ending the turn.

5.3.13 Bankruptcy

Test Case	Declare bankruptcy with balance >= 0
Test Case Description	To ensure that a player cannot declare bankruptcy if he has a positive balance.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Pick a player to test with. 3. Try to declare bankruptcy.

Output (Expected Results)	• The player is not allowed to declare bankruptcy, since he has a positive balance. • Either the "Declare Bankruptcy" button is disabled or an error message appears indicating that the player cannot declare bankruptcy unless he has a negative balance.

Test Case	Declare bankruptcy with debt to a player
Test Case Description	If a player declares bankruptcy with a debt to a player, his properties and cash (negative) are transferred to the player he is in debt to.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Pick a player to test with. 3. Play several turns, forcing all players to buy the properties they land on. 4. For the test player, to buy only two properties. 5. Keep playing turns, until the test player gets a negative balance due to him landing on another player's property and having to pay rent. 6. Click "Declare Bankruptcy" 7. Keep playing turns, to ensure that the bankrupt player's turn is skipped.
Output (Expected Results)	• The ownership of the player's properties are transferred to the player he is in debt to. • The player's balance (negative balance) is transferred to the player he is in debt to. • The player is withdrawn from the game and does not get to play a turn.

Test Case	Declare bankruptcy with debt to bank
Test Case Description	If a player declares bankruptcy with a debt to the bank, his properties become un-owned and his cash disappears (goes to the bank).
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Pick a player to test with. 3. Play several turns, forcing all players except the test player to buy the properties they land on. 4. Keep playing turns, until the test player gets a negative balance due to him picking a Pay JFL card 5. Click "Declare Bankruptcy" 6. Keep playing turns, to ensure that the bankrupt player's turn is skipped.
Output (Expected Results)	• The player's properties become un-owned. • The player's balance disappears (goes to the bank). • The player is withdrawn from the game and does not get to play a turn.

Test Case	Declare bankruptcy with debt to player and mortgaged properties
Test Case Description	If a player declares bankruptcy with a debt to player, his mortgaged properties are transferred as mortgaged
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Pick a player to test with. 3. Play several turns, forcing all players to buy the properties they land on. 4. For the test player, to buy only two properties. 5. Mortgage one of the properties. 6. Keep playing turns, until the test player gets a negative balance due to him landing on another player's property and having to pay rent. 7. Click "Declare Bankruptcy" 8. Keep playing turns, to ensure that the bankrupt player's turn is skipped.
Output (Expected Results)	• The ownership of the player's un-mortgaged properties are transferred to the player he is in debt to and the property remains un-mortgaged. • The ownership of the player's mortgaged properties are transferred to the player

	he is in debt to and the property remains mortgaged. • The player's balance (negative balance) is transferred to the player he is in debt to. • The player is withdrawn from the game and does not get to play a turn.

Test Case	Declare bankruptcy with debt to bank and mortgaged properties
Test Case Description	If a player declares bankruptcy with a debt to the bank, his mortgaged properties become un-owned and un-mortgaged.
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Pick a player to test with. 3. Play several turns, forcing all players to buy the properties they land on. 4. For the test player, to buy only two properties. 5. Mortgage one of the properties. 6. Keep playing turns, until the test player gets a negative balance due to him owing the bank some money. 7. Click "Declare Bankruptcy" 8. Keep playing turns, to ensure that the bankrupt player's turn is skipped. 9. Keep playing turns until another player buys the property that was mortgaged and became un-owned.
Output (Expected Results)	• The ownership of the player's un-mortgaged properties are transferred to the player he is in debt to and the property remains un-mortgaged. • The ownership of the player's mortgaged properties are transferred to the player he is in debt to and the property remains mortgaged. • The player's balance (negative balance) is transferred to the player he is in debt to. • The player is withdrawn from the game and does not get to play a turn.

5.3.14 End Game

Test Case	End Game
Test Case Description	Select End Game from menu, should exit game properly, even if balance < 0
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Keep playing until the balance of a player becomes negative. 3. Select the "End Game" option from the menu.
Output (Expected Results)	• The game ends, even if a player has a negative balance.

5.3.15 Game Winner

Test Case	Game Winner
Test Case Description	If 1 player left, game winner should be declared
Input (Steps to produce test)	1. Start the game with 2-8 players. 2. Keep playing until only one player remains in the game.
Output (Expected Results)	• The "Game Winner" window appears, displaying the name of the winner.

5.4 User Interface Testing

To test the User Interface, each functionality described in the design document will be verified to see if it has been implemented correctly, if it responds normally and also if no errors occur during the process between the user and the game.

A schema will be used to test (unit), what is the purpose of the test (what is tested), what are the inputs (from the user for instance), what is the expected result and also what is the effective (real result).

5.4.1.1 Start Panel

Interface	Start Panel 
What is tested?	Adding players
Inputs (requested, given by the program)	The user fills the “Nickname” text box, chooses a token, chooses the type of the player (human or computer) by clicking the adequate button and then click add player to add the current player to the list.
Expected result	When a user clicks on add player, the system must check the inputs validity. If the nickname field is left blank, an error message (“You must choose a username”) should appear. If no token is chosen, an error message (“You must choose a token”) should appear. When a username is given and a token chosen, after clicking “Add Player”, the chosen token should disappear. One different token per player.
Effective result	After choosing one token, giving a nickname and clicking on “Add Player”, the player is added to the players’ list. When the field is left blank, if you click on the “Add Player”, there is an error message “You must enter a name for this player”. If no token is chosen, the button “Add Player” is not visible. So you cannot add a player without selecting a token. No error message. One different token per player.

	When a token is assigned to a player, and the player is added to the game, then the token disappear.

Interface	Start Panel
What is tested?	Selection of a player type
Inputs (requested, given by the program)	The user clicks on the buttons “Human” or “Computer” to choose the type of the player.
Expected result	When a button, which status is up, is clicked, it should be down (to show that it is currently selected). When a player whose type is Computer is added, the name of the player should be prefixed by the character #.
Effective result	When a button is up and is clicked, then its status is down. When a player whose type is Computer is added, its nickname is not prefixed by the character #, also the name given by the user is changed for “Computer number”. When the type Computer is selected, the text box allowing to enter a nickname is disabled.

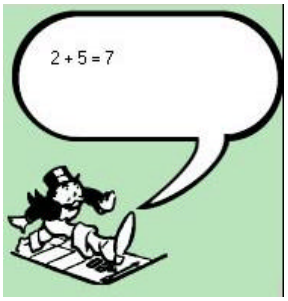
Interface	Start Panel
What is tested?	Adding players - The number of players – Starting a game
Inputs (requested, given by the program)	The user chooses several players, when he has finished, he clicks on “Let’s Start” to start a game.
Expected result	It should not be possible to start a game without at least 2 players, and more than 8 players. The players’ list must indicate all the players created. If the user try to start a game without creating 2 players, or with creating more than 8 players, an error message should appear (“You must have between 2 to 8 players to start a game”). When a game is started (click on “Let’s start”), the game board must be loaded with the created players, and the start panel should be closed.
Effective result	The user must at least create 2 players in order to make the button “Let’s start” (to start a game) visible. No possibility to create more than 6 players and less than 2 players. If less than 2 players the button “Let’s start” is invisible. If there are 6 players, the button “Add player” is invisible. Normally (in the design document), these buttons should have been enabled and an error message provided. The players’ list indicates all the players created. No possibility to create more than 6 players (should be 8). Clicking on “Let’s start” closes the start panel interface and launches the Game board interface with the created players.

5.4.1.2 Game board

Interface	Game board
What is tested?	Moving the window
Inputs (requested, given by the program)	User clicks on the title bar of the window to move it elsewhere on the screen.
Expected result	The window should be moved and placed where the user wants.
Effective result	As expected.

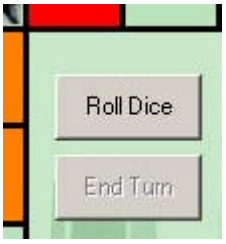
Interface	Game board 
What is tested?	Players list
Inputs (requested, given by the program)	No inputs needed
Expected result	The players list should list each player (what is the player's token, what is its name, and what is its amount of money in the bank). When a player is playing, its nickname should be highlighted.
Effective result	Each players (up to 6, normally should have been up to 8) are listed correctly in the list (token + nickname + bank credit). When a user is playing, it is highlighted (through a bar under its nickname).

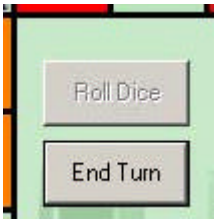
Interface	Game board
What is tested?	Players list, changing the amount of money
Inputs (requested, given by the program)	No user interaction directly needed. Paying a fine, taxes, rent. Mortgaging or unmortgaging a property. Passing though the "Go" cell. Collecting a rent or money. Finalising a trade with another user.
Expected result	The amount of money should be changed for the concerned player(s).
Effective result	As expected.

Interface	Game board 
What is tested?	Message Area
Inputs (requested, given by the program)	No user interaction is need. Actions performed during the game.
Expected result	For each action accomplished during the game, an explicit message should appear in this area. For instance, if the user runs double, it should be indicated that the user has run doubles and can throw dice again. Different type of messages: . Run doubles . You have landed on … . You must pay …. / You have paid …. . Go to jail . The trade offer has been rejected
Effective result	As expected.

Interface	Game board 
What is tested?	Now playing panel
Inputs (requested, given by the program)	No user interaction is needed. Ending a turn, the next player must play.
Expected result	This interface indicates who is currently playing. When a player has finished its turn, this interface should be reloaded and should indicate the new player who has to play. During trading, a player makes an offer to a player B, when the player B receives the offer, B must know it's it turn to play. When B replies, it's player A time to play. The "Now Playing" interface should reflect these

	states.
Effective result	When a turn it's finished, the hand goes to the next player and the "Now playing" panel is updated in consequence. Nevertheless, the panel is not updated during the trading process.

Interface	Game board 
What is tested?	Rolling dice
Inputs (requested, given by the program)	Mouse Click on the Roll Dice button.
Expected result	The dice must be rolled. The token must move to the adequate cell. The button must be disabled and end turn must be enabled (if no double). If double, normally the end turn button is still disabled and roll dice enabled.
Effective result	The dice are rolled. The token moves to the adequate cell. The button "Roll Dice" is disabled, and "End Turn" is enabled. If the player runs double, then a click on "End Turn" is needed before running the dice again.

Interface	Game board 
What is tested?	Ending a turn
Inputs (requested, given by the program)	Mouse click on the "End Turn" button.
Expected result	The hand goes to the next player. Roll Dice button is enabled for the next player. It should not be possible to end a turn when actions are incomplete.
Effective result	The hand goes to the next player.

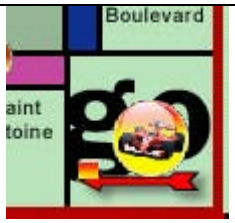
	Roll Dice button is enabled for the next player. BUT If you click on End Turn while a title deed card is opened, and then you close the title deed, the game freezes.

Interface	Game board
What is tested?	Landing on a cell.
Inputs (requested, given by the program)	No user input needed. The token moves to a cell.
Expected result	If the cell is a property (street or metro or utility): . free: title deed card appears and the player has the possibility to buy it. . belongs to the player: nothing. . belongs to another player and is not mortgaged (and contains hotels or not): money is collected automatically (if sufficient founds) . belongs to another player and is mortgaged: nothing If the cell is a JFL: . the JFL card appears. If the cell is Jail: . nothing If the cell is Go: . nothing If the cell is Olympic Park: . nothing If the cell is Go To Jail: . the token moves to the Jail Cell If the cell is a tax: . a tax card appears and the money is automatically collected
Effective result	If the cell is a property (street or metro or utility): . free: title deed card appears and the player has the possibility to buy it. . belongs to the player: nothing. . belongs to another player and is not mortgaged (and contains hotels or not): first you have to pay the rent (see below – Title deed section), then the money is collected. . belongs to another player and is mortgaged: nothing If the cell is a JFL: . the JFL card appears. If the cell is Jail:

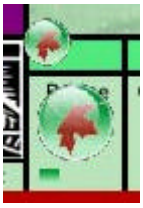
	. nothing If the cell is Go: . nothing If the cell is Olympic Park: . nothing If the cell is Go To Jail: . the token moves to the Jail Cell If the cell is a tax: . a tax card appears and the player has to click on “Pay Rent”

Interface	Game board
What is tested?	Clicking on a cell
Inputs (requested, given by the program)	Mouse click on a cell
Expected result	If the cell is a property (street or metro or utility): . the corresponding title deed card appears If the cell is not a proprerty: . a card containing information concerning the cell (the purpose of the cell, what will happened if you land on it, etc.) If the cell contains a token on it, and the player clicks on the token, the system should act as if there was no token on it.
Effective result	If the cell is a property (street or metro or utility): . the corresponding title deed card appears If the cell is not a proprerty: . nothing If a token is on the cell, and the user clicks on the token, then nothing appears.

Interface	Game board
What is tested?	Several players on the same cell.
Inputs (requested, given by the program)	No user interaction needed. Several token are placed on the same cell.
Expected result	The cell should indicate which players are on it. Each token has a little square, with the same colour. When several token are on the same cell, these little squares from the different players should be visible on the cell.

Effective result	 *2 players on the same cell. Only one appears.* When 2 or several players are on the same cell, nothing indicates that several players are on it. Sometimes it works, sometimes it bugs.

Interface	Game board 
What is tested?	Getting out of jail
Inputs (requested, given by the program)	The user s in jail and it's its turn to play.
Expected result	User has several choices: . Clicking on Roll Dice to run double and get out. If double are run, the player gets out of jail automatically. . Clicking on "Pay $50 to get out of jail" to get out of jail immediately . Clicking on "Use Your JFL get out of jail card" (if the user has this in card in its inventory)
Effective result	As expected.

Interface	Game board 
What is tested?	Owning a property.
Inputs (requested, given by the	A player has bought a property.

program)	
Expected result	The player token (the little one) is placed on the top of the property.
Effective result	As expected.

Interface	Game board 
What is tested?	Mortgaging/Unmortgaging a property.
Inputs (requested, given by the program)	A player has mortgaged a property. A player has unmortgaged a property.
Expected result	It the player has mortgaged a property, the “M” icon is placed on the top of the property. It the player has unmortgaged a property, the “M” disappears.
Effective result	As expected.

Interface	Game board 
What is tested?	Building hotels / Selling hotels.
Inputs (requested, given by the program)	A player has build or sold a hotel.
Expected result	It the player has build a hotel, a hotel icon is placed on the top of he cell. It the player has sold a hotel, a hotel icon is removed from the top of the cell.
Effective result	As expected.

Interface	Game board 
What is tested?	Finishing the game. Declaring bankruptcy.
Inputs (requested, given by the program)	The player has no money left and cannot collect money in anyway. The only alternative is to press “Declare Bankruptcy”.

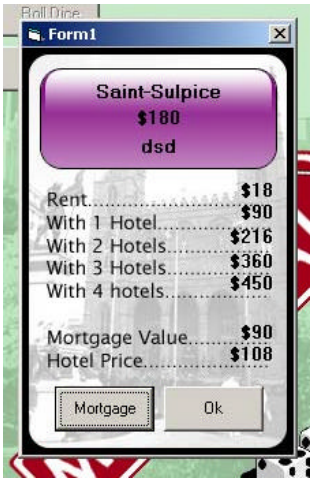
Expected result	The player presses “Declare Bankruptcy”. Its name is stroked from the players list. All its mortgaged properties return to the bank. The player cannot play anymore.
Effective result	As expected, the name is not stroked, but “Bankrupt” is written in the amount for the money.

5.4.1.3 Title deed cards

Interface	Title deed 
What is tested?	Buying a property
Inputs (requested, given by the program)	Player lands on a cell.
Expected result	The corresponding title deed card appears. Two choices are offered: . Buy It: to buy the property . Forget It!: to cancel the proposal. When the choice is made, the title deed disappears. If the property has been bought, then the token is placed on the top of the cell on the Game board.
Effective result	As expected.

Interface	Title deed card

	
What is tested?	Getting information on a free title deed.
Inputs (requested, given by the program)	Clicking on a vacant property.
Expected result	The corresponding title deed pop-ups and the player can click on the button “OK” to close it and return to the game.
Effective result	The title deed pop-ups but no button to close it. The user must click on the top-right corner cross to close it.

Interface	Title deed 
What is tested?	Mortgaging / Unmortgaging a property
Inputs (requested, given by the program)	Player clicks on the cell of one of its property. No hotel on it.
Expected result	The title deed appears, and the player has 2 choices: . if the property is not mortgaged, a Mortgage button is on. By clicking it the title deed closes itself, the corresponding amount of money is collected, and the “M” icon is placed on the cell on the game board. . if the property is mortgaged, an Unmortgage button is on the card. By

	pressing it the title deed closes itself, the amount of money is debited, and the "M" icon is removed from the cell on the game board. . A OK button to close the card and do nothing.
Effective result	As expected.

Interface	Title deed 
What is tested?	Trading a property.
Inputs (requested, given by the program)	Player clicks on the cell owned by another player.
Expected result	The title deed appears, and the player has 2 choices: . a trade button to trade this property. If the player clicks on it, then the trading card appears (see below, trading cards). . A OK button to close the card and do nothing.
Effective result	As expected.

Interface	Title deed 
What is tested?	Buying and selling hotels

Inputs (requested, given by the program)	Player clicks on the cell of one of its properties. Player owns all the properties of the district.
Expected result	The title deed appears, and the player has several choices . If enough money, a “buy hotel” button. By pressing it, the money is debited from the bank account, the title deed disappears, and a hotel icon is placed on the cell on the game board. . Not enough money and no hotel on the property: mortgage and ok button only. . If hotels on a property, a “sell button” is enabled. By clicking on it, the money is collected from the bank, the title deed disappears, and a hotel icon is removed on the cell on the game board. . A mortgage button if no hotel on the cell. . A OK button to close the card and do nothing. . If hotels on any property of the district, it should not be possible to mortgage a property without hotels. The player should sell to the bank all the hotels first. It should not be possible to build several hotels on a property if the other properties contain no hotel.
Effective result	As expected BUT It is possible to mortgage a property while the district still contains hotels. It is possible to buy several hotels on a property, even if the other properties of the district contain no hotel.

Interface	Title deed 
What is tested?	Paying the rent
Inputs (requested, given by the program)	Player lands on a property owned by another player, and no mortgaged. The corresponding card appears, and a button “Pay Rent” allows to pay the rent.
Expected result	The player clicks on “Pay Rent”, the money is debited from its account, and collected to the owner’s.

	The card is closed.
Effective result	As expected BUT The name of the button “Pay Rent” has been changed for “Pay xx” where xx represents the rent to pay.

5.4.1.4 Metro / Utility card

Same comportment as the title deed cards, except the fact that it is not possible to build and sell hotels on it.

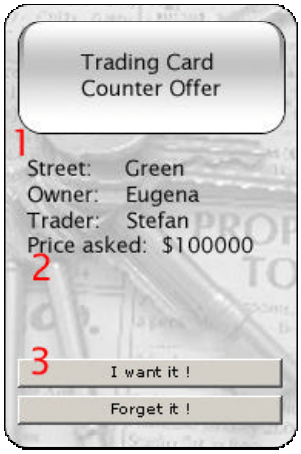
5.4.1.5 Trading cards

Interface	Trading card 
What is tested?	Making a proposal to buy a property
Inputs (requested, given by the program)	Player clicks on the desired cell. The cell must not contain hotels. The title deed appears, and the player presses on “Trade”. The player then indicates an amount of money for the property.
Expected result	The player proposes an amount of money for the property. By clicking on “I want it”, the proposal is sent to the owner of the desired street. The trading card disappears, the other players has the hand. If the amount of money is not valid, an error message should appear. By pressing “Forget it”, the proposal is cancelled.
Effective result	As expected. BUT If the amount of money is invalid (not filled, or incorrectly filled), the program crashes.

Interface	Trading card

	
What is tested?	Accepting or refusing a trading.
Inputs (requested, given by the program)	A player has sent a proposal through the Trading Card. The receiver is the owner of the property. The player can give an amount of money.
Expected result	The receiver has several choice: . Clicking on “Done Deal Buddy”: in that case, the trading is made. Bank accounts are debited and credited, the trading card disappears, and the token of the new owner is placed on the cell. The hand gets back to the initial trader. . Indicating a new amount of money in the adequate text box and click “I want more”. Then the hand goes back to the initial trader and the counter offer card appears. The amount of money must be valid, if not, an error message should appear. . “Forget it”, to close the window and go on playing.
Effective result	As expected. BUT Nothing happens when clicking on “I want more” except that the amount of money is changed on the card.

Interface	Trading card – When counter-offering.

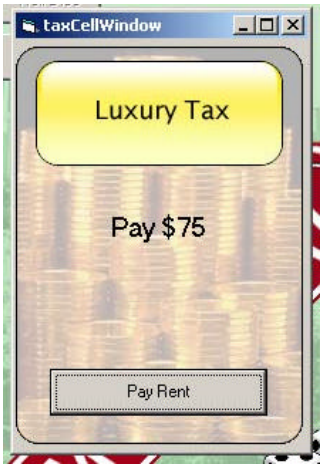
	
What is tested?	Accepting or refusing a counter-offer.
Inputs (requested, given by the program)	The owner of the desired cell has given an amount of money and has pressed on “I want more”
Expected result	The hand gets back to the initial player. The counter-offering card appears. If the player presses on “Forget it”, the trade is cancelled, the card is closed and the game goes on. If the player clicks on “I want it”, then the trading is made. Bank accounts are debited and credited, the counter-offering card disappears, and the token of the new owner is placed on the cell.
Effective result	Functionality not implemented / not working.

5.4.1.6 JFL (Just For Laughs) Cards

Interface	Just For Laughs 
What is tested?	The viewing of a JFL card.
Inputs (requested, given by the program)	The player lands on a JFL cell.

Expected result	A JFL card appears. The player clicks on the “OK” button to close it and the action described on the JFL is executed. If the card is a “Get out of jail card”, when the player will go to jail, the button “Use your Get Out Of Jail card” will be available. Is the card gives money, the money is collected to the bank account. If the card asks for money, the money is debited from the bank account. If the card is “Do nothing”, nothing is done. If the card is “Go to Jail”, the player goes to jail. If the card tells to move to a certain place or to a certain number of cells, the token is moved to the good location.
Effective result	As expected. The card is automatically closed after 1 second.

5.4.1.7 Luxury tax / Income tax

Interface	Luxury tax / Income tax 
What is tested?	Paying a fine.
Inputs (requested, given by the program)	The players lands on an Income Tax or Luxury Tax.
Expected result	The corresponding card appears. The player then clicks on “Pay Rent”, the money is taken from its account and the card is closed.
Effective result	As expected.

5.4.1.8 Winner interface

Interface	Winner interface

	
What is tested?	End of the game
Inputs (requested, given by the program)	All users except one have declared bankruptcy.
Expected result	The winner interface is loaded. The name and the token of the winner appear in the Montreal Metro front page.
Effective result	As expected.

Interface	Winner interface 
What is tested?	Starting a new game or quitting the game.
Inputs (requested, given by the program)	The winner clicks on “Start a new game” The winner clicks on “Exit Game”
Expected result	By pressing “Start a new game”, the winner interface is closed, and a new start panel (reset) appears. By pressing “Exit”, the game is closed.
Effective result	As expected.

5.4.1.9 File menu

Interface	Board game  
What is tested?	Menu bar
Inputs (requested, given by the program)	Player clicks on the menu. Player moves the mouse pointer to an option an click on it to select it.
Expected result	The file menu was designed in this way: File -> New Game (to start a new game) Exit (to quit the program) About (information concerning the team)
Effective result	The file menu is like that: File -> Sound -> Mute : to mute the sounds of the application Enable : to enable the sounds Voice -> Mute: to mute the voice (not tested) Enable: to enable the voice (not tested) Exit Help -> About: information concerning the team

5.5 Performance Profiling

In order to test the performance of the application, we will test and evaluate the response time of the game. Due to the nature of the application (game), this is mostly concerned with the response time of the token movements and the AI. Therefore, we will be testing two dimensions.

First, the speed of the token movements should be measured and analyzed to ensure that it is slow-enough for the user to know what is going on, but fast enough so that the game does not become unexciting.

Second, the response of the decisions made by AI should be measured and analyzed to ensure that if a human player is playing against one or several computer players, the response is slow-enough for the user to know what the computer player is doing, but fast enough so that the game does not become dull unexciting.

5.5.1 Token Movements

Test Case	Move 2 steps
Test Case Description	To ensure that if the token is moved by 2 steps, the movement is noticeable
Input (Steps to produce test)	6. Start a game with 2-8 players 7. Player several turns until the value on the dice roll is 1-1 (value is 2) 8. Measure the time it takes for the token to move.
Output (Expected Results)	• The token movement time is in the neighborhood of 1 second.

Test Case	Move 6 steps
Test Case Description	To ensure that if the token is moved by 6 steps, the movement is noticeable
Input (Steps to produce test)	1. Start a game with 2-8 players 2. Player several turns until the value on the dice roll is 6. 3. Measure the time it takes for the token to move.
Output (Expected Results)	• The token movement time is in the neighborhood of 2 seconds.

Test Case	Move 12 steps
Test Case Description	To ensure that if the token is moved by 12 steps, the movement is noticeable
Input (Steps to produce test)	1. Start a game with 2-8 players 2. Player several turns until the value on the dice roll is 12. 3. Measure the time it takes for the token to move.
Output (Expected Results)	1. The token movement time is in the neighborhood of 3 seconds.

5.5.2 AI Response

Test Case	Computer player lands on owned-property
Test Case Description	To ensure that if a computer player lands on a property owned by another player, the time taken by the computer to make a decision is within acceptable range.
Input (Steps to produce test)	1. Start a game with 2-8 players, and at least 1 computer player. 2. Player several turns until the computer player lands on a property owned by another player. 3. Measure the time it takes for the computer player to make a decision and pay the rent.
Output (Expected Results)	2. The response time is in the neighborhood of 1 second.

Test Case	Computer player lands on un-owned property
Test Case Description	To ensure that if a computer player lands on an un-owned property, the time taken by the computer to make a decision is within acceptable range.
Input (Steps to produce test)	1. Start a game with 2-8 players, and at least 1 computer player. 2. Player several turns until the computer player lands on an un-owned property. 3. Measure the time it takes for the computer player to make a decision and buy the property.
Output (Expected Results)	3. The response time is in the neighborhood of 1 second.

Test Case	AI does a trade with an AI player
Test Case Description	To ensure that if a compute player decides to trade with another computer player, the time to make a decision is within an acceptable range.
Input (Steps to produce test)	1. Start a game with 2-8 players, and at least 2 computer players. 2. Play several turns until a computer player decides to make a trade offer with another computer player. 3. Measure the time it takes for the computer player to make a decision and accept or reject the trade.
Output (Expected Results)	4. The response time is in the neighborhood of 2 seconds.

5.6 Load Testing

Load testing is normally concerned with testing the system beyond the limits it was designed for. However, due to restrictions we have placed on the number of players (8 players) that can participate in one game, we cannot, for instance, test to see if the game works with 9 players. Therefore, we will be testing the game as close as possible to the limits it was designed for. In fact, we will attempt to simulate a fully loaded board, where all the properties are owned, and each property has the maximum number of hotels built on it. In this scenario, we will re-evaluate the game's functionalities and response times.

Test Case	Fully Loaded Board – Functionality
Test Case Description	To ensure that if the board is fully loaded, the game is still functioning correctly.
Input (Steps to produce test)	1. Start a game with 8 players, and at least 1 computer player. 2. Play many turns, forcing players to buy the properties they land on. 3. Do not build any hotels until all the properties are owned. 4. Start trading properties until every player owns a district. 5. Start building hotels until all the properties have 4 hotels build on them. 6. Play several turns, until all Functions mentioned in section 5.3 (Function Testing) have been properly tested.
Output (Expected Results)	• The functions tested work as expected (see section 5.3).

Test Case	Fully Loaded Board – AI
Test Case Description	To ensure that if the board is fully loaded, the AI response time is acceptable.
Input (Steps to produce test)	1. Start a game with 8 players, and at least 1 computer player. 2. Play many turns, forcing players to buy the properties they land on. 3. Do not build any hotels until all the properties are owned. 4. Start trading properties until every player owns a district. 5. Start building hotels until all the properties have 4 hotels build on them. 6. Play several turns, and measure the time it takes for the computer player to play his turn.
Output (Expected Results)	• The response time is in the range of 2-3 seconds.

5.7 Configuration Testing

Configuration testing is concerned with testing the application under different environment configurations the users may have. For the Montrealopoly game, we will be focusing on testing the game under different versions of the Microsoft Windows ™ operating system. As per the requirements document, this includes Windows 95, Windows 98, Windows Me, Windows 2K and Windows XP, but excludes Windows NT.

In order to simulate these different client environments, we will be using awell-known software emulation software called Virtual PC ™. This software (similar to the VMWare product series) emulates the hardware of a personal computer and allows you to install and test different operating systems simultaneously. For example, your main operating system may be Windows XP, but using Virtual PC will allow you to run several other operating systems as "children" of your main OS.

In order to test the Montrealopoly game under several different operating systems, we will be using Virtual PC to emulate these environments and then test the game under them.

Test Case	Windows 95
Test Case Description	To ensure that the game runs properly under Windows 95.
Input (Steps to produce test)	1. Install Virtual PC 2. Create a new PC called Windows 95 3. Install Windows 95 on that virtual PC 4. Copy the Montrealopoly game and all files needed to execute it. 5. Re-test the function test cases that were detailed in section 5.3 (Function Testing) 6. Re-test the user interface test cases that were detailed in section 5.4 (User Interface Testing) 7. Re-test the installation test cases that were detailed in section 5.8 (Installation testing)
Output (Expected Results)	• The test cases' expected results are as described in section 5.3.

Test Case	Windows 98
Test Case Description	To ensure that the game runs properly under Windows 98.
Input (Steps to produce test)	1. Install Virtual PC 2. Create a new PC called Windows 98 3. Install Windows 98 on that virtual PC 4. Copy the Montrealopoly game and all files needed to execute it. 5. Re-test the function test cases that were detailed in section 5.3 (Function Testing) 6. Re-test the user interface test cases that were detailed in section 5.4 (User Interface Testing) 7. Re-test the installation test cases that were detailed in section 5.8 (Installation testing)
Output (Expected Results)	• The test cases' expected results are as described in section 5.3.

Test Case	Windows Me
Test Case Description	To ensure that the game runs properly under Windows Me.
Input (Steps to produce test)	1. Install Virtual PC 2. Create a new PC called Windows Me 3. Install Windows Me on that virtual PC 4. Copy the Montrealopoly game and all files needed to execute it. 5. Re-test the function test cases that were detailed in section 5.3 (Function

	Testing) 6. Re-test the user interface test cases that were detailed in section 5.4 (User Interface Testing) 7. Re-test the installation test cases that were detailed in section 5.8 (Installation testing)
Output (Expected Results)	• The test cases' expected results are as described in section 5.3.

Test Case	Windows 2K
Test Case Description	To ensure that the game runs properly under Windows 2K.
Input (Steps to produce test)	1. Install Virtual PC 2. Create a new PC called Windows 2K 3. Install Windows 2K on that virtual PC 4. Copy the Montrealopoly game and all files needed to execute it. 5. Re-test the function test cases that were detailed in section 5.3 (Function Testing) 6. Re-test the user interface test cases that were detailed in section 5.4 (User Interface Testing) 7. Re-test the installation test cases that were detailed in section 5.8 (Installation testing)
Output (Expected Results)	• The test cases' expected results are as described in section 5.3.

Test Case	Windows XP
Test Case Description	To ensure that the game runs properly under Windows XP.
Input (Steps to produce test)	8. Install Virtual PC 9. Create a new PC called Windows XP 10. Install Windows XP on that virtual PC 11. Copy the Montrealopoly game and all files needed to execute it. 12. Re-test the function test cases that were detailed in section 5.3 (Function Testing) 13. Re-test the user interface test cases that were detailed in section 5.4 (User Interface Testing) 14. Re-test the installation test cases that were detailed in section 5.8 (Installation testing)
Output (Expected Results)	• The test cases' expected results are as described in section 5.3.

5.8 Installation Testing

After having completed the implementation, the application will be packaged by the well-known InstallShield software. This generated package is an application by itself. When this application is executed, it installs the Montrealopoly game into a location that can be specified by the user. In this section, we will focus on testing this installation package. Most importantly, the package should be compatible with the different operating systems it is required to support and should copy/provide the necessary .dll (dynamic link libraries) files for the game to work properly.

Test Case	Install – Normal
Test Case Description	To ensure that the game installs properly under normal conditions (the game has never been installed before, there is enough disk space and the user has enough privileges to install).
Input (Steps to produce test)	1. Run the installer (setup.exe). 2. Install the application into the default directory. 3. Start a game with 2-8 players. 4. Verify the location of the installed application. 5. Test the game music. 6. Test the sound effects. 7. Test the text-to-speech effects.
Output (Expected Results)	• The game music, sound effect and test-to-speech are working correctly. This is an indication that the .dll files have been installed properly. • The game's executable and other necessary files are installed in the default directory.

Test Case	Install – Normal – Override Directory
Test Case Description	To ensure that the game installs properly under normal conditions (the game has never been installed before, there is enough disk space and the user has enough privileges to install), but the user overrides the default directory.
Input (Steps to produce test)	1. Run the installer (setup.exe). 2. Install the application, but override the default directory. 3. Start a game with 2-8 players. 4. Verify the location of the installed application. 5. Test the game music. 6. Test the sound effects. 7. Test the text-to-speech effects.
Output (Expected Results)	• The game music, sound effect and test-to-speech are working correctly. This is an indication that the .dll files have been installed properly. • The game's executable and other necessary files are installed in the directory that was specified by the user.

Test Case	Install – Insufficient Disk Space
Test Case Description	To ensure that the installer detects and handles an insufficient disk space problem correctly.
Input (Steps to produce test)	1. Create a large temporary file to use up the disk space of the hard drive. 2. Leave a little bit of space, but not enough to install the game. 3. Run the installer (setup.exe). 4. Try to install the application.
Output (Expected Results)	• The installer displays an error message indicating that there is insufficient disk space.

Test Case	Install – Already Installed
Test Case Description	To ensure that the installer works properly if the application is already installed.
Input (Steps to produce test)	1. Install the game. 2. Delete some of the files from the installation directory. 3. Install the game again.
Output (Expected Results)	• The installer replaces the deleted files.

Test Case	Install – Already Installed – Game Running
Test Case Description	To ensure that the installer works properly if the application is already installed and the game is running.
Input (Steps to produce test)	1. Install the game. 2. Start the game with 2-8 players. 3. Try to install the game again.
Output (Expected Results)	• The installer detects that the game is already running and displays an error message indicating that.

Test Case	Install – Not Enough Privileges
Test Case Description	To ensure that the installer does not allow a user with low privileges to install the application.
Input (Steps to produce test)	1. Create a user with low privileges (this can be the Guest user in XP) 2. Try to install the game.
Output (Expected Results)	• The installer detects that the user doesn't have enough privileges and displays and error message indicating that.

Test Case	Un-Install – Normal
Test Case Description	To ensure that once the game is installed, it can be un-installed.
Input (Steps to produce test)	1. Install the game. 2. Un-install the game using the Add/Remove Programs option in the Control Panel.
Output (Expected Results)	• All files in the installation directory are removed.

Test Case	Un-Install – Game Running
Test Case Description	To ensure that if the game is running, it can't be un-installed.
Input (Steps to produce test)	1. Install the game. 2. Start the game with 2-8 players. 3. Try to un-install the game using the Add/Remove Programs option in the Control Panel.
Output (Expected Results)	• The un-installer detects that the game is running and displays an error message indicating that.

6. Testing Workflow

In this section, we will describe, in detail, the procedures and guidelines that are to be followed during the testing effort. This will outline the flow of the testing activities implicated, and allow us to easily manage the bugs that are found, resulting in a smooth testing phase.

6.1 Workflow Overview

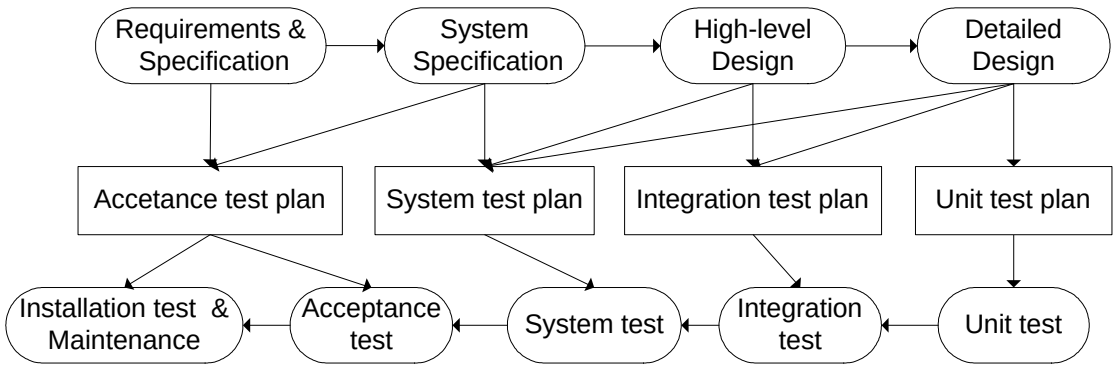
Some of the goals of this testing phase are to test for: correctness of algorithms, correctness of implementation, good GUI and proper performance level.

6.1.1 Test Plan & Software Engineering process

First, let us make clear the relationship between the test plan and the software engineering process of our project. As shown in the next diagram, the relationships are:

- We use the detailed design document to produce the unit-testing plan.
- We use the detailed design document to produce the integration-testing plan.
- After the system is integrated, we test the system's features by using the requirements document.
- Finally, we use the systems specifications to ensure that the implemented system follows them.

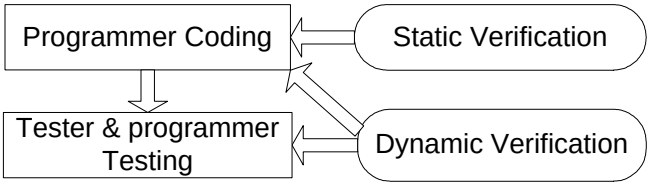
Test plan & SE process



6.1.2 Static and Dynamic Verification

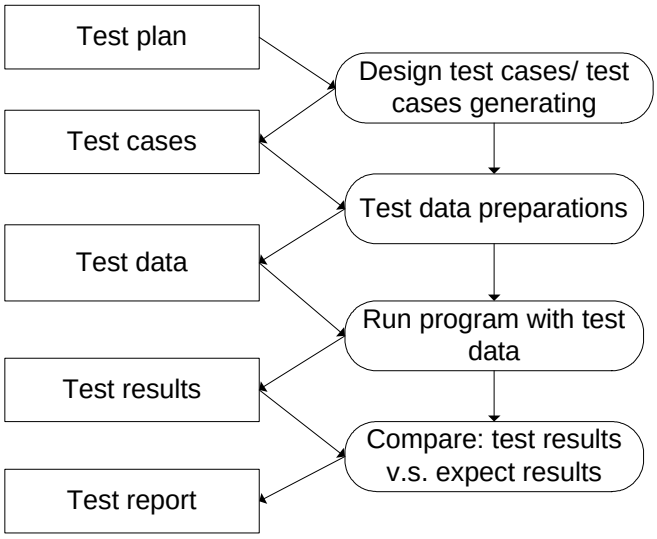
Static and dynamic testing was performed during the implementation phase of the project. The implementation team did the static verification by doing desk checking on their code. Dynamic testing was done by the programmers and the testers to find bugs when the system was executing.

Static & Dynamic Verification



6.1.3 Work Flow of a Test

Work Flow of a Test



As the diagram shows, for each target test item, there are tests. For each test, the workflow will be:

- Design test cases and generate test cases. The output of this step is the test cases.
  Note: optimization is required to minimize the number the test cases that are required.
- Test data preparation is the second step. The test data can be prepared by the detailed test documentation. The output of this step is the test data.
- Run the program with the test data is the next step. The output of this step is the test results.
- The last step is to compare the test results with the expected results. The output of this step will be the bug reports, modification, suggestions, etc.

This diagram applies to most of the tests, specifically the unit tests, integration tests, and system tests.

6.2 Incident Logs and Change Requests

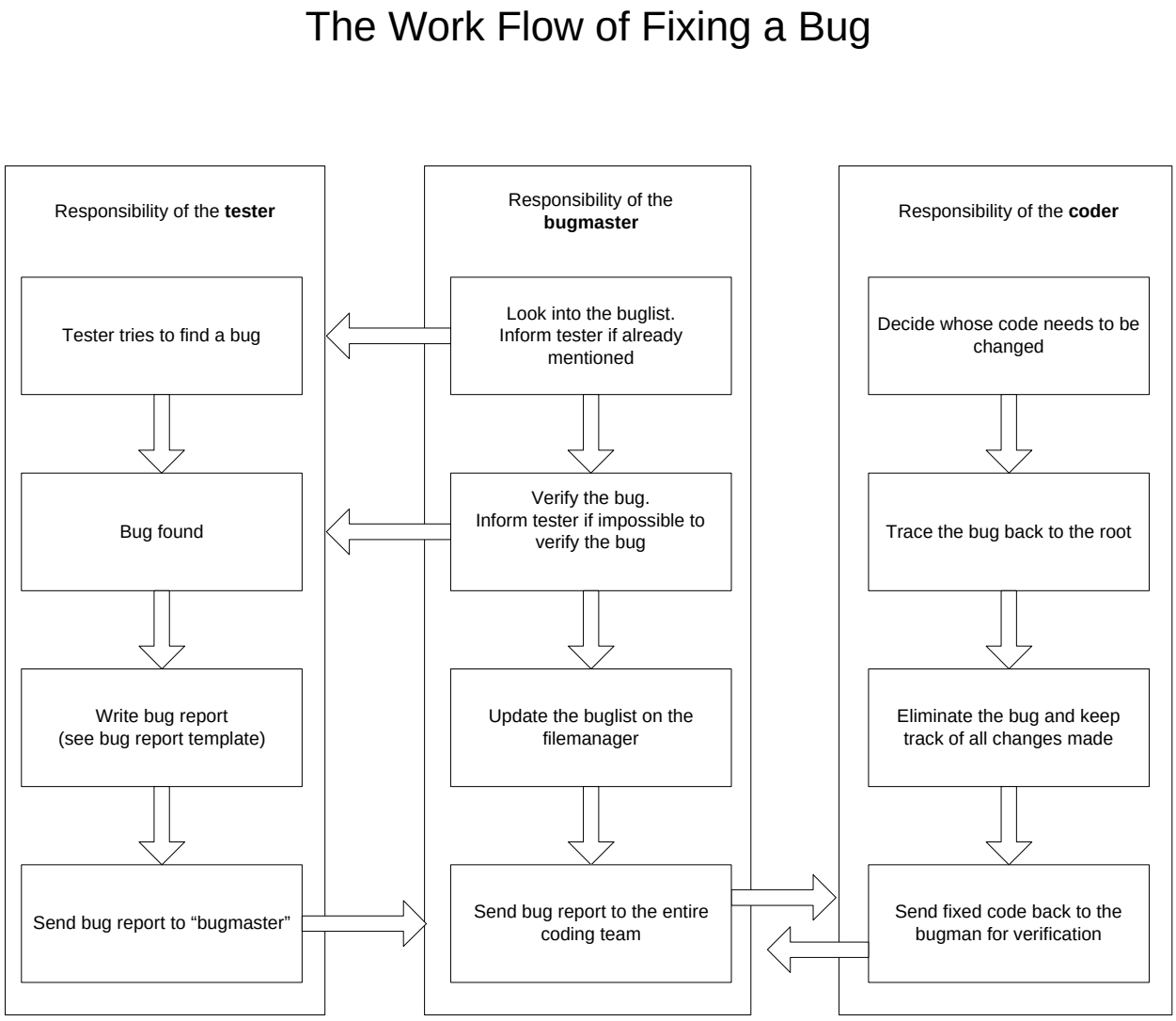
To manage changes in the testing process, several templates for bug management, unit testing and integration testing were created. These templates help improve the traceability of the testing. An Internet based file manager was setup to store all the files and templates that the members of Team Redmond could use to share files and view the bug list. A mailing list was also setup in order to facilitate communications between group members.

6.2.1 Managing changes: the file manager and group e-mail list

Managing change is very important in all phases of software engineering processes. To facilitate this, a file manager was setup in order to allow team members to upload their files and access other team member's files. Everything that the group required was stored on the file manager, from templates and documentations to source code and executables of the game. To coordinate communications between team members, a mailing list was setup in order to keep all group members up to date on the current events of the project.

6.2.2 Bug Workflow

As the diagram shows, there is a standard workflow for fixing bugs and there are interactions between the tester, bug master and coder. Each of them has a clearly defined responsibility when it comes to bug management. The bug master keeps a master bug list of all the bugs that have been submitted and updates their status. The tester is responsible for testing the game and filling out the bug template when a bug is found and sending it to the bug master. The coder is responsible for fixing the bugs and notifying the bug master when they have been fixed. The tester will then retest the game to ensure that the bug has been properly fixed and do appropriate regression testing to ensure that no other bugs have been created as a result of this bug fix.



6.2.3 Bug Report Template

This is the bug report template that is used by the testers to report a bug.

Bug #		Title:							
Bug Date:		Submitted by:		Assigned to:		Status:	Open	Status Date:	
Description									

This bug was caused by…

Screen shot (optional)

How to use it:

The Bug Master keeps this template. When a bug is found, the tester fills out the bug form and e-mails it to the Bug Master at bugs@maverick.to. The Bug Master will then verify the bug, add it to the master bug list and e-mail the implementation team about it. Whenever a bug is assigned to a member of the implementation team, the member will e-mail the Bug Master, who will then update the master bug list with who is working on the bug. Once the bug has been fixed, the member will e-mail the Bug Master again with the new status of the bug. The Bug Master will then notify the tester that the bug has been fixed and the tester can then test it again.

The bug list fields must be filled out in the following way:

- Bug #: Leave blank, filled in by the Bug Master.
- Title: Title of the bug.
- Bug Date: Date bug discovered.
- Submitted By: Person who submitted the bug
- Assigned To: Leave blank, filled in by the member who is working on the bug.
- Status:
  - Open: All bugs have this initial status.
  - Confirmed: Bug has been assigned to someone.
  - Closed: Bug has been fixed.
- Status Date: Date the status was changed.

6.2.4 Master Bug List

The bug master is in charge of maintaining the master bug list. It contains all the bugs that have been submitted and the bug master updates the statuses of the bug whenever one is fixed. Below is a sample master bug list.

#	Date	Status	Submitted By	Bug description
1	Dec. 1	Open	Alex	on the Income and Luxury tax cards, maybe change pay rent for pay tax
2	Dec. 1	Open	Alex	the games crashes when you don't close a title deed and you go on playing

Master Bug List

6.2.5 Responsibilities of the tester, bug master and coder

6.2.5.1 Responsibilities of the tester

The procedure starts with the testing effort, which means that each tester of the testing team tries to find bugs. If a bug is found he fills out the bug report template and sends it to the bug master.

6.2.5.2 Responsibilities of the bug master

The bug master is responsible for tracking all the bugs that have been submitted and is the link between the coders and testers. Once a bug is received, the bug master tries to reproduce the bug and if it is reproducible and is not a duplicate bug, it is added to the master bug list. The bug master assembles all the bugs into a master bug list and keeps track of their status. The master bug list is on the file manager, which the programmers can view to see what bugs need to be fixed.

6.2.5.3 Responsibilities of the coder

The leader of the implementation team decides which programmer is to fix the bug and notifies him. The programmer then looks at the bug list and attempts to fix the bug. Once the bug has been fixed, the programmer then notifies the bug master that the bug has been fixed and that the bug's status can now be changed.

6.2.6 Black box testing template

Unit testing is the activity that verifies each module in isolation. For each test items of unit testing, both black box and white box testing was performed. Several templates for black box testing were made including some for boundary value analysis and equivalence partitioning.

6.2.6.1 One variable boundary value analysis method

The first template is for one variable black box testing which can be used to test a class or a method of a class.

Black box testing template for 1 variable (BVA)

Tester name				Test date			
Class name		Method name		File name			
Variable name		Lower bound		Upper bound:			
less than lower bound	**Value:**						
on lower bound	**Value:**						
between the bounds	**Value:**						
on the upper bound	**Value:**						
greater than upper bound	**Value:**						
Test Case	less than lower bound	on lower bound	between the bounds	on the upper bound	greater than upper bound		
Expected output							
Actual output							
Bug found?							

6.2.6.2 Two variables boundary value analysis method

The two-variable template considers all the possible combinations of two variables. This table can be expanded to multiple variables template.

Black box testing template for 2 variables (BVA)

Name of tester				Testdate	
Class name		Name of method		Filename	
v: (1st variable name)		v: Lower bound		v: Upper bound	
w: (2nd variable name)		w: Lower bound		w: Upper bound	

v1: 1st variable less than lower bound	Value v1:
v2: 1st variable on lower bound	Value v2:
v3: 1st variable between the bounds	Value v3:
v4: 1st variable on the upper bound	Value v4:
v5: 1st variable greater than upper bound	Value v5:
w1: 2nd variable less than lower bound	Value w1:
w2: 2nd variable on lower bound	Value w2:
w3: 2nd variable between the bounds	Value w3:
w4: 2nd variable on the upper bound	Value w4:
w5: 2nd variable greater than upper bound	Value w5:

Variable w1					
Testcase	v1 ~ w1	v2 ~ w1	V3 ~ w1	v4 ~ w1	v5 ~ w1
Expected output					
Actual output					
Bug found?					
Variable w2					
Testcase	v1 ~ w2	v2 ~ w2	V3 ~ w2	v4 ~ w2	v5 ~ w2
Expected output					
Actual output					
Bug found?					
Variable w3					
Testcase	v1 ~ w3	v2 ~ w3	V3 ~ w3	v4 ~ w3	v5 ~ w3
Expected output					
Actual output					
Bug found?					
Variable w4					
Testcase	v1 ~ w4	v2 ~ w4	v3 ~ w4	v4 ~ w4	v5 ~ w4
Expected output					
Actual output					
Bug found?					
Variable w5					
Testcase	v1 ~ w5	v2 ~ w5	v3 ~ w5	v4 ~ w5	v5 ~ w5
Expected output					

Actual output					
Bug found?					

6.2.6.3 Equivalence Partition Testing

Different from boundary analysis method template, equivalence-partitioning template just considers the valid classes and invalid classes of variables.

Black box testing template for one/multiple variable (EP)

Tester name				Test date			
Class name		Method name		File name			
Variable name		Lower bound		Upper bound:			
Variable name							
Valid class 1							
Valid class 2							
Invalid class 1							
Invalid class 2							
Testcase	1	2	3	4	5		
Expected output							
Actual output							
Bug found?							
Note: this template user equivalent partition method. It is similar to do the multiple variables testing by adding the variables and valid/invalid classes' cells.							

6.2.7 White box testing template

White box testing template

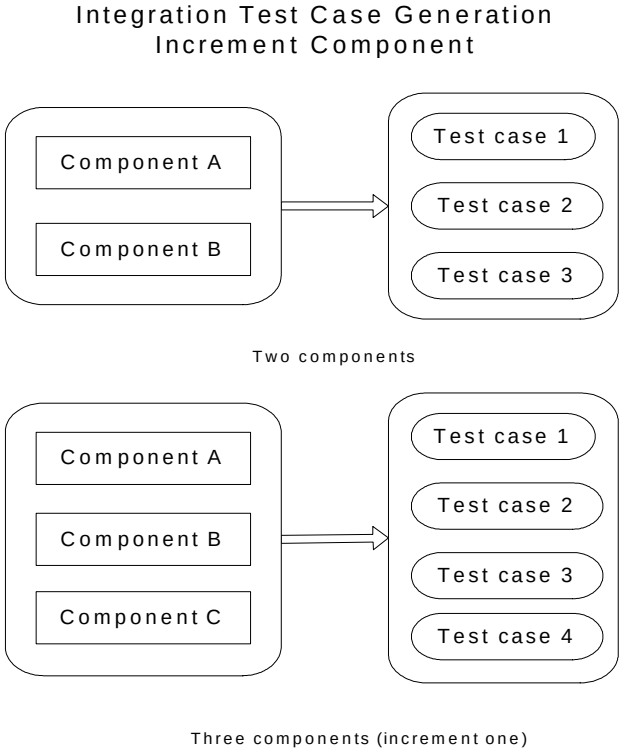
Test ID		Tester name:		Test date	
Class name:		Method name:		VB File name	
Code segment that is marked with step number					
Path diagram					
Test method:		Number of test case			
Test case ID	1	2			
Test cases name					
Content to test					
Expect result					
Test result					
Find bug					
Path 1	1-2-10-16-17-18-20-22-24				
Variables					
Expected result					
Bug description:					
Note: 0 suggest that each table for one code segment of one test method. 1 test method includes path test, branch test, condition testing, loop testing. 2 you may have many test case, just add them. 3 if you want to test more than one method, just add them. 4 if a bug is fixed, this table can also be used for testing the bug fixing result. 5 the size of the table can be changed. 6 the basis path test method can also be used.					

6.2.8 Integration test

For integration testing, testing consists of testing groups of components incrementally together. Defining the order of integration is of prime importance.

6.2.8.1 Method

The make sure that each component is tested once, we use a method as the diagram shows. First, we test two components that produce several test cases. Then more components are added to the system. This will generate new test cases.



6.2.8.2 Integration test template

Integration testing template

Test ID		Tester name:		Test date	
Test name					
Relevant information	Requirement:				
	Specification				
	Scenario:				
	VB File names				
Relevant test components	Classes involved				
	Unit test status (y/n)				
	Other test components involved				
	Unit test status (y/n)				
Test method		Sandwich integration			
Item description					
Test cases number					
Test case ID					
Test cases name					
Test case description (why this test case designed?)					
Expect result					
Content to test (what to test exactly?)					
Testing procedure (How to test)					
Bugs Found					
Bug ID		Bug processor		Status	
Bug description:					
Test result/suggestion					
Verifier name		Verifying Date			

Note:
0 each test item is suggested has a table.
1 test method includes Sandwich integration only.
2 you may have many test case, just add them.
3 if a bug is fixed, this table can also be used for testing the bug fixing result.
4 the size of the table can be changed.

7. Iteration Milestones

The following are the milestones that were set in this iteration.

			November									December										
Milestone	Days	Who	5	13	15	20	22	27	28	29	30	1	2	3	4	5						
General																						
Test Plan Template Creation	1d	ALL			■																	
Test Plan Document	19d	ALL			■	■	■	■	■	■	■	■	■	■	■							
Phase 3 deliverables																■						
Testing																						
Testing & QA	6d	ALL							■	■	■	■	■	■	■							
Implementation																						
Code	30d	Programmers	■	■	■	■	■	■	■	■	■	■	■	■	■							
Build 1	5d	Programmers		■	■																	
Build 2	6d	Programmers				■	■															
Build 3	6d	Programmers						■	■	■												
Build 4	4d	Programmers									■	■	■									
Release Candidate	2d	Programmers												■	■							
Final Montrealopoly Game	1d	Programmers														■						

8. Team Members Log Sheets

8.1 Stefan Thibeault

Date	Task	Duration
28/10/2003	Meeting - initial kick-off meeting	2 hours
15/11/2003	Meeting - to discuss document template, what to include/exclude	6 hours
22/11/2003	Individual – worked on document	4 hours
27/11/2003	Individual – worked on document	5 hours
01/12/2003	Tested the game	2 hours
02/12/2003	Meeting – worked on bugs, document	6 hours
03/12/2003	Individual – worked on Document / tested game	8 hours
04/12/2003	Individual – Finalized Document	6 hours
	Total:	45 Hours

8.2 Robert Hanna

Date	Task	Duration
28/10/2003	Meeting - initial kick-off meeting	2 hours
15/11/2003	Meeting - to discuss document template, what to include/exclude	6 hours
18/11/2003	Individual – Product Functions	3 hours
30/11/2003	Individual – Game Testing	2 hours
02/12/2003	Individual – Product Functions (continued…)	5 hours
02/12/2003	Meeting – worked on bugs, document	6 hours
04/12/2003	Individual – Section 5.4 – 5.7	4 hours
04/12/2003	Individual – help with document finalization	5 hours
	Total:	33 hours

8.3 Simon Lacasse

Date	Task	Duration
28/10/2003	Meeting - initial kick-off meeting	2 hours
03/11/2003	Prototype 1	8 hours
05/11/2003	Prototype 1 Implementation	6 hours
12/11/2003	Prototype 1 Validation (Team Meeting)	2 hours
20/11/2003	Prototype 2	8 hours
22/11/2003	Prototype 2 Implemenation	12 hours
23/11/2003	Prototype 2 Validation	12 hours
25/11/2003	Visual Design Implementation	12 hours
26/11/2003	Sound Implementation	12 hours
27/11/2003	Visual and Module Integration	6 hours
29/11/2003	Debugging	10 hours
02/12/2003	Meeting – worked on bugs, document	6 hours
03/12/2003	Final debugging	10 hours
04/12/2003	Voice Implementation	1 hours
	Total:	103 hours

8.4 Alexandre Bosserelle

Date	Task	Duration
28/10/2003	Meeting - initial kick-off meeting	2 hours
18/11/2003	Individual – User interface testing	8 hours
02/12/2003	Individual – Web Page Design for the Montrealopoly website	1 hour
02/12/2003	Individual – Target Test Items (Section 3)	0.5 hours
02/12/2003	Meeting – worked on bugs, document	6 hours
04/12/2003	Individual – Updated section 5.4	2 hours
	Total:	19.5 hours

8.5 Eugena Zolorova

Date	Task	Duration
01/12/2003	Individual – Section 1	5 hours
	Total:	5 hours

8.6 Zhi Zhang

Date	Task	Duration
28/10/2003	Meeting - initial kick-off meeting	2 hours
15/11/2003	Meeting - to discuss document template, what to include/exclude	6 hours
22/11/203	Individual – worked on document	8 Hours
28/11/2003	Individual – worked on document	8 Hours
02/12/2003	Meeting – worked on bugs, document	6 hours
03/12/2003	Completed integration testing	8 hours
04/12/2003	Completed section 6	8 hours
	Total:	46 hours

8.7 Xin Xi

Date	Task	Duration
	Did not participate in this phase	0
	Total:	0

8.8 Patrice Michaud

Date	Task	Duration
28/10/2003	Meeting - initial kick-off meeting	2 hours
05/11/2003	Board Class and Board initialisation	6 hours
07/11/2003	Street, Utility, Metro	1 hours
13/11/2003	Other Cell	1 hours
14/11/2003	Basic Player Class	3 hours
16/11/2003	CellInfoWindow and advance player class	3 hours
18/11/2003	Jail	9 hours
19/11/2003	Trading between human and trading card	3 hours
21/11/2003	JFLCard JFLDeck JFLCardWindow	3 hours
22/11/2003	Tax card with calculate assets	3 hours
25/11/2003	AI basic move	2 hours
28/11/2003	AI advance (automakemoney, autotrade)	4 hours
30/11/2003	Various enhancements	3 hours
01/12/2003	Misc. bug fixes	12 hours
02/12/2003	Meeting – worked on bugs, document	6 hours
	Total:	61 hours

8.9 Hu Shan Liu

Date	Task	Duration
28/10/2003	Meeting - initial kick-off meeting	2 hours
12/111/2003	Implemented JFL Queue	6 hours
	Total:	8 Hours

8.10 Jens Witkowski

Date	Task	Duration
28/10/2003	Meeting - initial kick-off meeting	2 hours
15/11/2003	Meeting - to discuss document template, what to include/exclude	2 hours
24/11/2003	Individual – section 6 (Testing Workflow)	3 hours
03/11/2003	Individual – updated section 6 (Testing Workflow)	3 hours
	Total:	10 hours