Technopoly

Project Report

Lars-Kristian Svenoy, James Greene, And Brendan Girvan

2019

**CSC7083 Peer Assessment: Technopoly**

This Assessment Document is intended to provide you and your assessor with an overview of each group member’s involvement delivery of the CSC7083 Project.

Each group should complete one Assessment Document and its content must be agreed by all group members. The completed form should be included as hard copy at the start of your group’s report. ***Don’t forget to fill in the Group Number*.**

There are two main parts to the Assessment Document – the Evaluation and the Declaration. Both parts must be completed – otherwise your group’s report will not be marked. Arrange a group meeting to discuss the evaluation, and see the note below!

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Evaluation** | Group Number: 3 |  | | | |
| Name | | Contribution of time and effort1 | Contribution to team-working and motivation1 | Contribution to the deliverables 1,2 | Peer Score  (Range 85 – 115) |
| Lars-Kristian Svenoy | | 3 | 5 | 5 | 115 |
| James Greene | | 3 | 2 | 2 | 93 |
| Brendan Girvan | | 3 | 2 | 2 | 92 |
|  | |  |  |  |  |
|  | |  |  |  |  |
|  | |  |  |  |  |
|  | |  |  |  |  |
|  | |  |  |  |  |
|  | |  |  |  |  |
|  | |  |  |  |  |
|  | |  |  |  |  |

1Values: 1 = Less than average; 2 = Slightly less than average; 3 = Average; 4 = Slightly more than average; 5 = More than average

2This value should consider contributions in the round – direct contributions to required deliverables, and contributions that have made the deliverables possible.

|  |  |  |
| --- | --- | --- |
| **Declaration**  “I declare that I have read the Queen's University regulations on plagiarism, and that any contribution I have made to this assignment is my own original work, except for any elements that I have clearly attributed to third parties. I understand that this assignment will be subject to an electronic test for plagiarism and will also be subject to the University’s regulations concerning late submission if it is received after the deadline.” | | |
| Name | Date | Confirmation *(use the words shown in the example below!)* |
| Lars-Kristian Svenoey | 14/03/2019 | I agree to the terms of the declaration |
| James Greene | 14/03/2019 | I agree to the terms of the declaration |
| Brendan Girvan | 15/03/2019 | I agree to the terms of the declaration |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Requirements Analysis – B.G

As part of our requirements analysis, we developed use case descriptions and diagrams to help assist in the realization of the project based on the requirements presented to us by our customer.

## Requirements

|  |  |
| --- | --- |
| Id | Requirement |
| REQ\_1 | The game has up to four players, and their names should be entered. |
| REQ\_2 | The players take turns. |
| REQ\_3 | On a player’s turn, they can throw 2 virtual dice. |
| REQ\_4 | There is a start square, where players pick up their resources. |
| REQ\_5 | There is an equivalent of a Free Parking square. |
| REQ\_6 | There are four fields, two consisting of three areas and two consisting of two areas. |
| REQ\_7 | One of the two-area fields is the most expensive field on the board to acquire and resource. |
| REQ\_8 | One of the two-area fields is the least expensive field on the board to acquire and resource. |
| REQ\_9 | Before you can develop an area within a field, you must own all areas in the field. |
| REQ\_10 | You can develop areas in a developed field no matter where your player is on the board. |
| REQ\_11 | You develop a field by building houses on it. |
| REQ\_12 | Three houses must be built on an area before a hotel may be built on it. |
| REQ\_13 | Players taking a turn are told where they have landed, and what their obligations or opportunities are. |
| REQ\_14 | Where appropriate, they may indicate their choice of action. |
| REQ\_15 | If a player’s resources have changed, the system indicates the reason for the change and announces the player’s new balance. |
| REQ\_16 | There is a cost associated with developing areas within fields. When you land on an area that someone else owns, you must pay a cost, which increases based on how developed the area is. |
| REQ\_17 | You may acquire unowned properties by giving up resources for it. Cost of properties increases the more expensive the property rent is. |
| REQ\_18 | When one player runs out of resources, the player with the most resources is declared the winner. A player may also choose to end the game, at which point the player with the most resources is declared the winner. |

## Game Board Visualisation



Based on the requirements, we made a graphical illustration of the board. This board illustrates how the player may move across the board and acquire properties in fields while doing so. The players start on the Start tile and move in a circular fashion around the board to acquire properties, while paying rent on properties they land on.

## Use Case Descriptions

|  |  |
| --- | --- |
| **Name** | **Start Game** |
| **ID** | UC\_1 |
| **Description** | Player initiates a game |
| **Pre-conditions** | System is loaded |
| **Post conditions** |  |
| **Main flow** | 1. User selects to start a new game 2. System prompts user to enter number of players (2-4) 3. System prompts user to enter names of players. 4. Last name that is entered starts game, the system displays options, players current bank balance and properties owned on screen 5. Player selects from options – roll dice, view board, manage properties or end game. 6. System displays choice and loads appropriate text-based comments/instructions |
| **Alternative flow** |  |
| **Extension points** | 1. At step 1, user selects to load a saved game. 2. At step 1, user selects to exit the game. |
| **Exceptions** | 1. At step 2, user enters a non-number of a number outside of range. 2. System prompts user to enter appropriate number 3. At step 3, duplicate names are entered. 4. System prompts user to select a different name. |

|  |  |
| --- | --- |
| **Name** | **Exit Game** |
| **ID** | UC\_2 |
| **Description** | Player exits the game |
| **Pre-conditions** | Game must be initialized |
| **Post conditions** | Player(s) with highest balance announced winner(s), end of game message shown |
| **Main flow** | 1. User selects to exit the game. 2. System announces winner(s). 3. Game is ended, and the system displayed a message to indicate it. |
| **Alternative flow** |  |
| **Extension points** |  |
| **Exceptions** |  |

|  |  |
| --- | --- |
| **Name** | **Land on square and associated options** |
| **ID** | UC\_3 |
| **Description** | Player takes turn, rolls dice and lands on a square of the board game. |
| **Pre-conditions** | Game must be in progress |
| **Post conditions** | Extension point identified, and appropriate instructions displayed |
| **Main flow** | 1. System displays name of square and available options 2. If square is not owned by another player, options to buy or auction the property are displayed 3. If square is owned by another player, system displays the rent that must be paid to the owner. |
| **Alternative flow** | 1. Player lands on a square that they own 2. Player turn ends |
| **Extension points** | 1. If a square is not already owned by another player, the system displays options to either buy or action the property. Player follows instructions following selection. (See UC\_4) 2. If following purchase, before rolling the dice, the player may select the “Manage Properties” action. (See UC\_7) 3. If the square is owned by another player, the system displays how much rent they must pay to the owner of the property. (See UC\_5) |
| **Exceptions** |  |

|  |  |
| --- | --- |
| **Name** | **Buy property – extends UC\_3** |
| **ID** | UC\_4 |
| **Description** | Player buys a property after landing on it, if it is available. |
| **Pre-conditions** | Game must be in progress, and the dice has been rolled |
| **Post conditions** | System will add the property to the player’s inventory, and the player’s balance will be updated |
| **Main flow** | 1. System displays name and price of property. 2. System displays options to buy or auction property 3. Player selects to buy 4. System displays name and price of property 5. System records player as owner of property 6. System displays how many properties player now owns and players new bank balance following transaction. |
| **Alternative flow** | At step 2 player selects to auction property:   1. System displays property for sale with a starting bid amount 2. Each player chooses to bid or not bid until property is allocated to highest bidder. |
| **Extension points** |  |
| **Exceptions** | 1. Player has insufficient funds to buy property 2. No player bids and property remains unsold |

|  |  |
| --- | --- |
| **Name** | **Pay rent on owned square – extends UC\_3** |
| **ID** | UC\_5 |
| **Description** | Player prompted to pay rent after landing on an owned square |
| **Pre-conditions** | Game in progress, dice rolled, player has moved |
| **Post conditions** | Current players bank account debited rent amount and bank balance of player that owns property is updated. System displays current players balance. |
| **Main flow** | 1. System displays that player has landed on an owned square and how much rent is due. 2. The system debits current players bank account and adds to owning players bank account. 3. System displays current players bank balance following transaction. |
| **Alternative flow** |  |
| **Extension points** |  |
| **Exceptions** | Player has insufficient funds to pay rent |

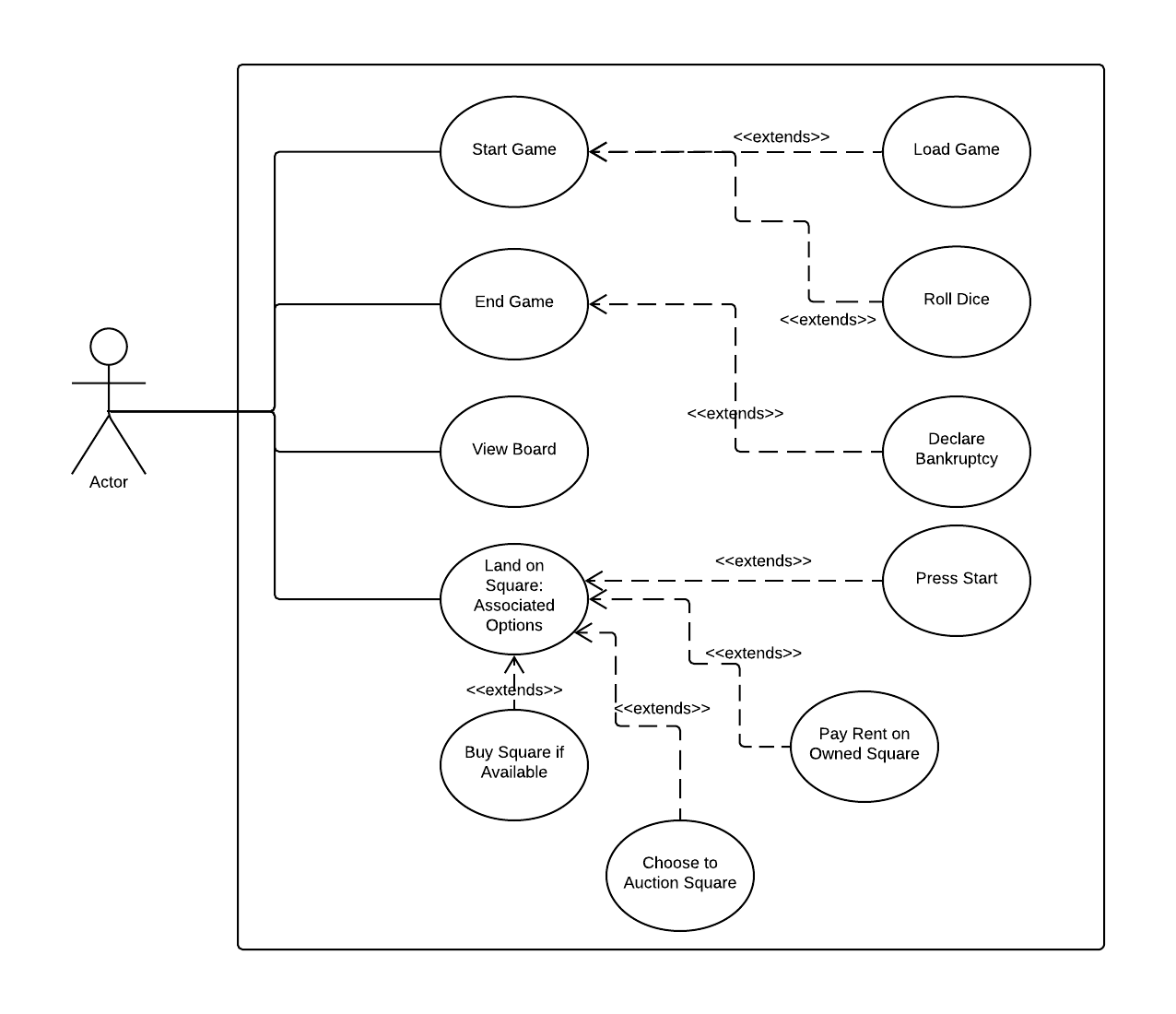
|  |  |
| --- | --- |
| **Name** | **Pass Start – extends UC\_3** |
| **ID** | UC\_6 |
| **Description** | Player passes start, and collects payment |
| **Pre-conditions** | Game in progress, dice rolled, player has moved |
| **Post conditions** | Payment is added to the players’ inventory, and the bank balance is displayed to screen |
| **Main flow** | 1. Player rolls dice 2. System displays message that player has passed start square on board 3. System adds payment to players bank balance and displays current balance on screen. |
| **Alternative flow** |  |
| **Extension points** |  |
| **Exceptions** |  |

|  |  |
| --- | --- |
| **Name** | **Manage Properties – extends UC\_3** |
| **ID** | UC\_7 |
| **Description** | When it is players turn, they can select option to manage properties, build offices and hotels if eligible |
| **Pre-conditions** | Game in progress, and it is the players turn |
| **Post conditions** | System records property to player and updates bank balance |
| **Main flow** | 1. Player selects to manage properties 2. System displays option to build offices, associated price and rent that will be charged. 3. Player selects to build office 4. System displays message and players bank balance following transaction. 5. System redisplays options to build offices and player can select until maximum of 3 offices are bought 6. System then displays option to build HQ, price and rent that can then be charged. 7. Player selects option then system displays players new bank balance. 8. System displays that property has now been upgraded to maximum. |
| **Alternative flow** | At step 2 if player does not own entire field:   1. System displays message indicating player does not own field. 2. System displays option to return to game |
| **Extension points** |  |
| **Exceptions** | Player has insufficient funds |

|  |  |
| --- | --- |
| **Name** | **View Board – extends UC\_3** |
| **ID** | UC\_8 |
| **Description** | Displays all property names on board in order, indicates which properties are owned by which players and where players are currently positioned on board. |
| **Pre-conditions** | Game in progress, and it is the players turn |
| **Post conditions** | Summary of current state of play in game with system displaying all properties, shows which properties players own and where players are currently positioned on board. |
| **Main flow** | 1. Players turn 2. Player selects option to view board 3. System displays information |
| **Alternative flow** |  |
| **Extension points** |  |
| **Exceptions** |  |

|  |  |
| --- | --- |
| **Name** | **Declare Bankruptcy – extends UC\_2** |
| **ID** | UC\_9 |
| **Description** | Player has insufficient funds left to continue and wishes to leave game |
| **Pre-conditions** | Game in progress, the player has rolled the dice and landed on a property they do not own, and can not afford to pay rent for |
| **Post conditions** | Player exits game |
| **Main flow** | 1. Player has insufficient funds to continue in game 2. Selects exit game 3. System displays player with highest bank balance as winner and a congratulations message. |
| **Alternative flow** |  |
| **Extension points** |  |
| **Exceptions** |  |

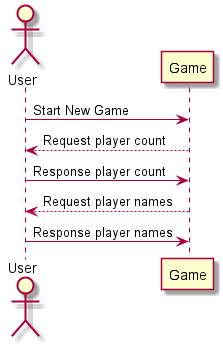
## Use Case Diagram



# Realisation – J.G

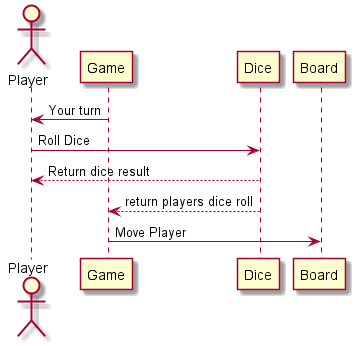
For the realization section of our report we made several sequence UML’s to correspond to our use cases highlighted in the requirements analysis. This involved planning regarding the method calls we would likely need to implement in the design stage of the system when components and the user communicate.

## Start Game



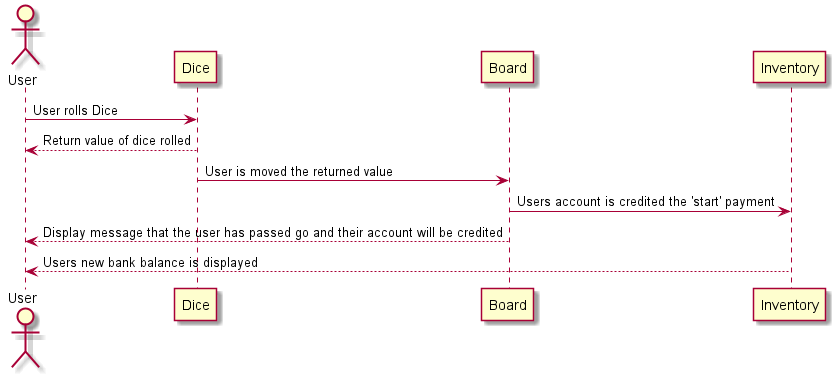
The first sequence UML we chose to represent was the start game feature. As you can see in the diagram a user would select to start a new game then the game would request a player count in response. A user would be prompted for the number of players then this interchange would also occur for each of the player names.

## New Turn



As you can see from the diagram above the game would indicate to a player that it is their turn. The user starts their turn by pressing a button to roll a dice interacting with the dice component. The dice result would be returned to the user in the form of a String variable to display how much they rolled. The result would also be returned to the game so that the player is moved the correct number of tiles along the board.

## Player passes start



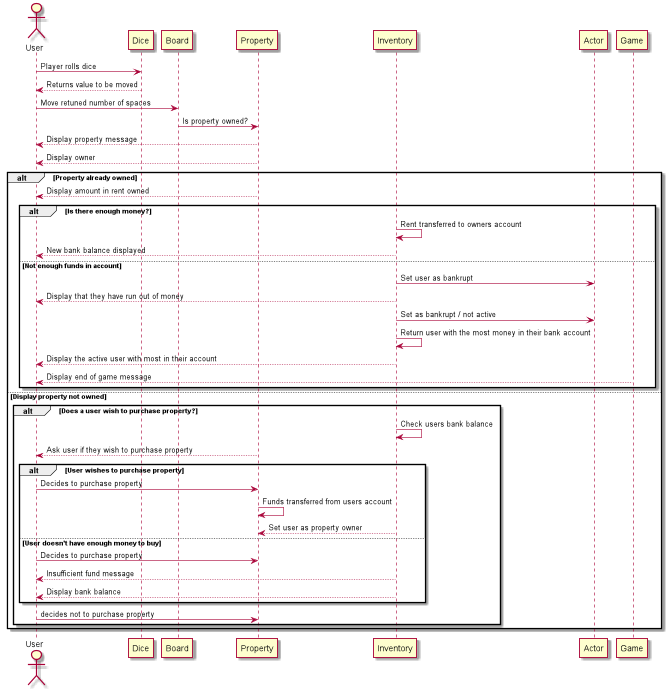
This UML is to show the sequence of method calls the system makes when a player passes go. In here the player would have had to roll a dice and the dice would have to be of a value that the user passes the ‘Start’ tile. When this happens, the board would make a call to the inventory to credit the users balance by the amount specified for the game. The user would then be displayed with a message indicating that they had passed to start tile and a new display of their bank balance.

## Player lands on square

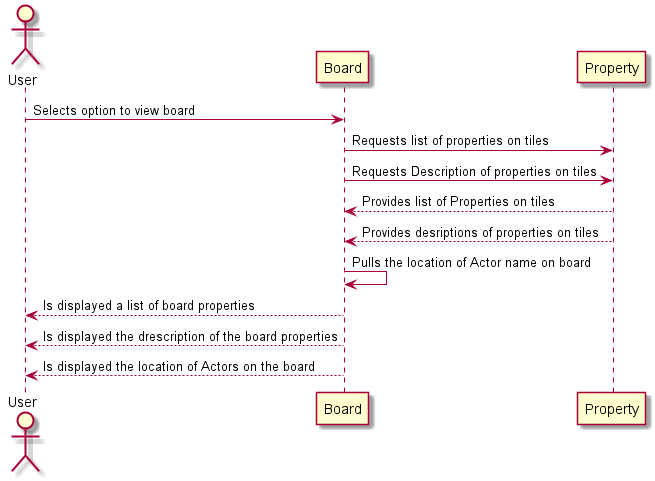
This was a pretty general scenario to represent as a sequence UML but provided us with a opportunity to plan out a lot of the actions / options that entailed from a simple action of a player moving to a different tile. Firstly, as displayed in a previous UML a dice is thrown and the player is moved several spaces. From here a user would be displayed with a description of the property they landed on and it if is currently owned or not.

The first scenario is that the square is already owned and that a user must pay rent. The rent would be returned along with the previous tile messages to the user and then the inventory would make a call to itself to credit the rent to the owners account with the specified amount. The diagram also displays an alternative flow whereby we illustrate the scenario where a user may not have enough money and the Inventory must make a call to the actor component to set the user as bankrupt. From here the system would display to the user that they have run out of money and that they are bankrupt and send a call to the actor component to set them as bankrupt. The Inventory would also select the active user with the most money in their account and return a string to the user to display the winning player. After this the game would output a method to signify that the game was finished.

Another sequence of alternate flows would be if the property isn’t owned. Firstly, there would be the flow of a user choosing to buy the property as displayed with the messages between the User, Property and Inventory components. A check would be done on whether the user has required funds and if it returns that they can afford the property then the Property component would make a call to the Inventory to set the tile to owned and against the players name. An alternative flow again is if the player can’t afford the property an error message is displayed.

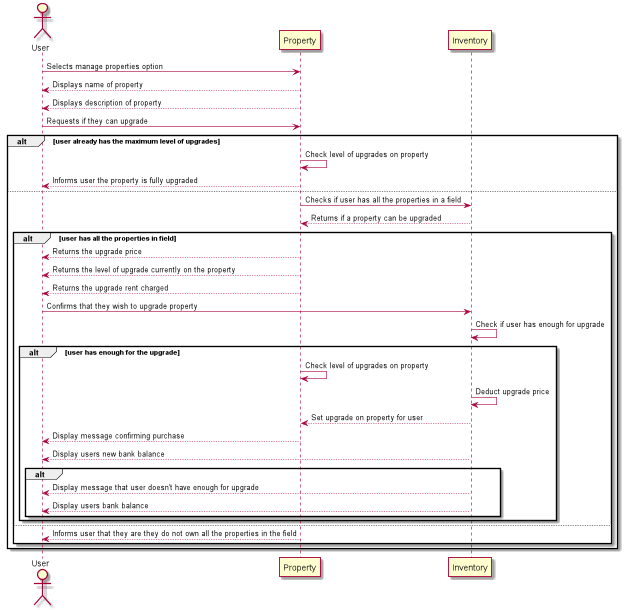


## View board / Display Board State

****

This is an option within a game for a player to view a board with a list of properties and the position of another player. As you can see illustrated on the diagram the user initially sends a request to view a board state to the board component. From here the Board and Property components would call and return property list and descriptions. The board would also pull the location of the actors on the board. After this the user would be displayed with a list of the properties and their descriptions followed by the current location of their Actor as well as the other players.

## Build Properties

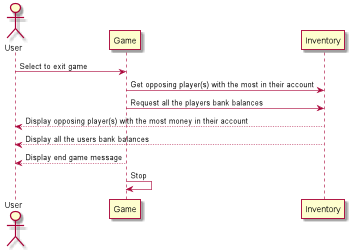
****

This feature is part of the core requirements in that it incorporates one of the main features of the ‘Technopoly’ game when a user can upgrade their properties when they own a section. In the diagram a user prompts to select a manage properties option, from here the property component returns the name and description displays. When a user requests to upgrade the property, several different flows can occur depending on the board circumstances.

There is a case where a property is already fully upgraded in which the Property component returns to the user that the property is already fully upgraded.

Another flow is a check to ensure that a user has all the properties in a field. After the initial request to upgrade. The price, level of upgrade currently on a property, rent to the upgrade will charge other players are displayed to the User from the Property component. The system performs a check on the inventory regarding if the player owns all a field that the property, they are currently on belongs to. The user is then prompted to confirm the purchase of an upgrade after which the inventory balance for the player is deducted the specified value followed by the Inventory setting the Property component to reflect that it is owned by the player. From here Property displays a message confirming purchase to the User and the Inventory displays the Users new bank balance.

## Exit Game

****

Finally, for the exit game feature which is in the core requirements. A user has the option of exiting if they no longer want to play where they select exit game which then sends a request to the Game component to end the game. The Game then request the opposing player with the most money from the Inventory and the other players names and bank balances. This is then sent to the user to be displayed before the Game component displays and end of game message before stopping the game.

# Design – L.S

## The COnfiguration System

For the game design, we looked at the requirements and use-cases and identified and made architectural decisions to support extending, maintaining and testing the game. One of the major components of the design is the configuration system we put in place.

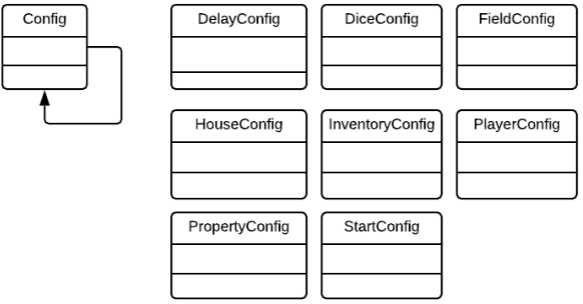


Figure 1. The configuration system

The configuration system is a set of objects that define variables for use in configuring the game. The objects are JSON serializable, so editing the game can be done externally without the need to re-compile the application. An added benefit to this approach is that we are following the single responsibility principle; by removing the configuration from its objects, we can more easily test the system and re-configure it on the fly. The configuration objects are injected into their objects, which follows the dependency inversion principle.

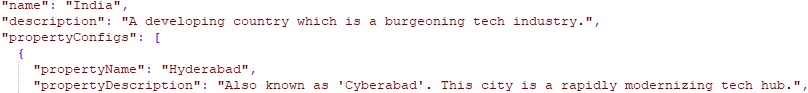


Figure 2. Example configuration JSON

The configuration system will detect an external config.json placed next to the executable, if no config.json file is present, the game will load its default configuration, which is stored in the java executable.

## The Action System

The game is a turn-based multi-choice game driven by player input. The input supplied by the user, except for specifying names during game initialization, is just a series of numbers that correspond to an action presented to the user during play.

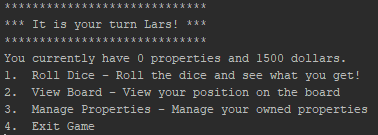


Figure 3. Example for how a user is prompted for input during play

### Actions

Based on this pattern, we developed the action system. The fundamental part of the action system is the action interface. The action interface specifies what an action must contain, in a consistent way.

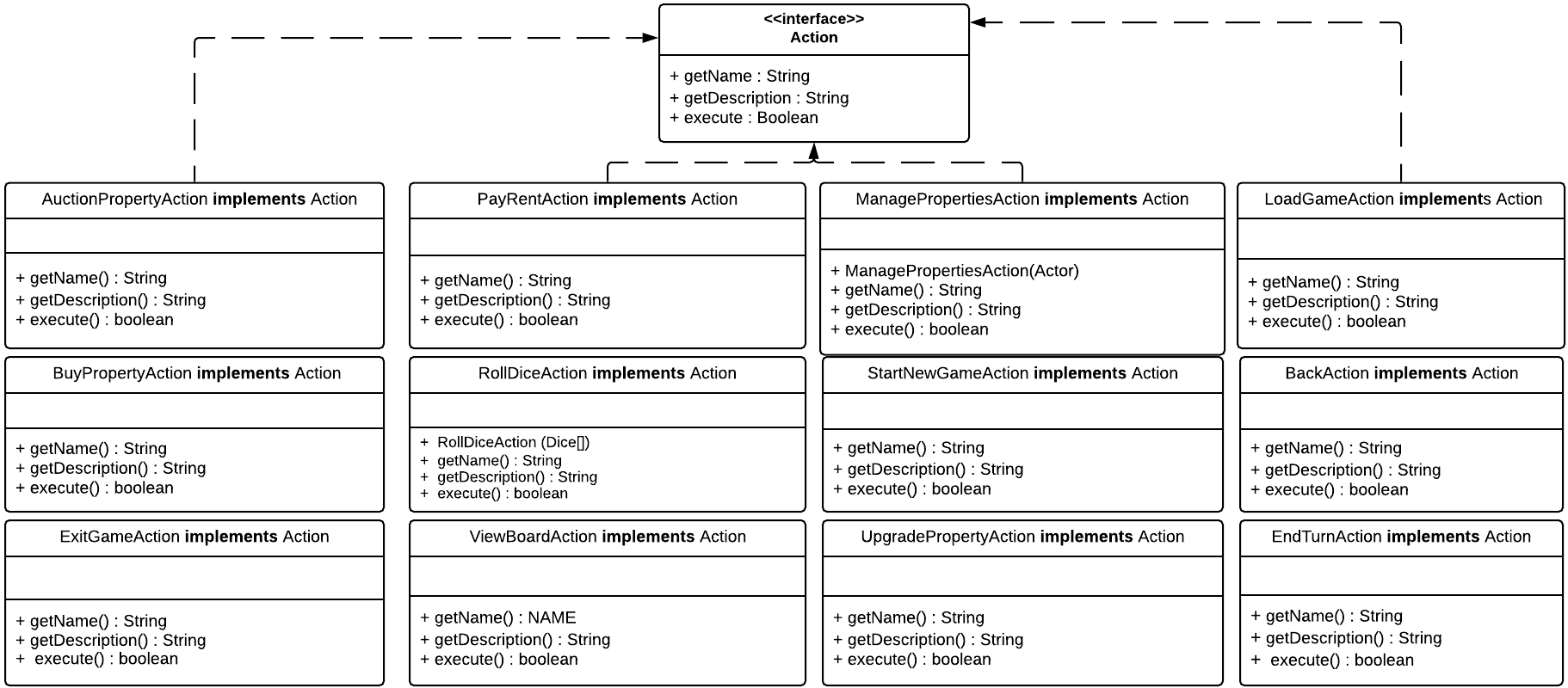


Figure 4. The action system

Specific actions define their dependencies in their constructors, and the getName() and getDescription() methods return human readable information about them. Finally, the execute() method is called to apply the action. If an action succeeds, execute() returns true. If not, it returns false. Additionally, an action may output textual information to the screen, and is the fundamental building block for changing state throughout the game.

The action system fits well based on how the game plays. The game presents choices to the player, which is essentially an array of actions for the user to select. Once the player inputs a number, that number gets indexed into the action array, and the game attempts to execute that action. Depending on the result from execution (true or false), the game may prompt the user to select a different action. The action system is a custom implementation of the common Action design pattern, as defined by the gang of four.

### Action Groups

To enable the logical grouping and orchestration of actions, the action system features a grouping system. As can be seen in **Figure 3**, three choices appear on the screen for the user to select from. The specific actions displayed on that screen are part of the NewTurnActionGroup. This logical grouping construct makes it easy to extend the game. By defining new action groups, one may introduce new concepts and further develop the game without touching existing code. Additionally, this logical grouping promotes maintainability by making testing simple.

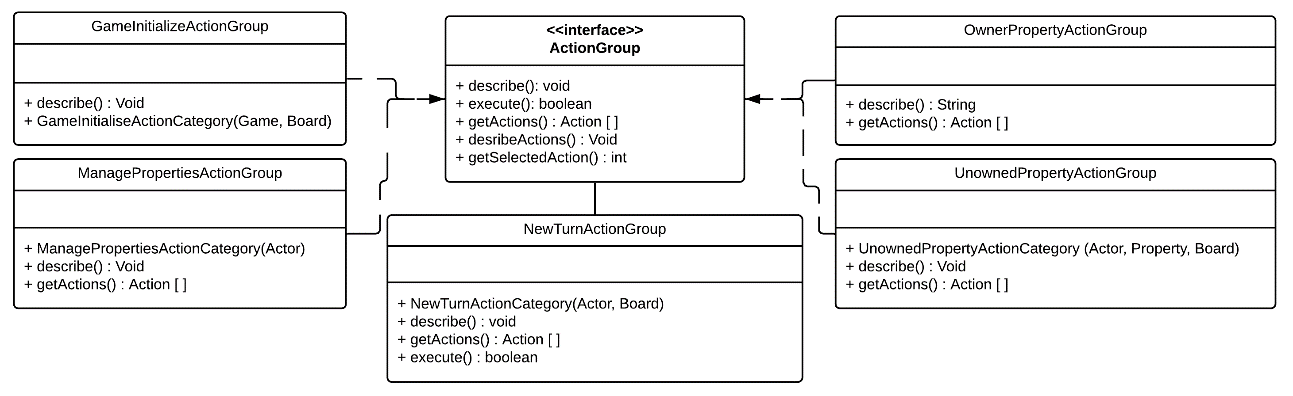


Figure 5. The action group system

As seen in **Figure 3**, the output action descriptions generated is a result of calling the describe() method on the corresponding ActionGroup. For convenience, the interface contains default implementations of these methods. When execute() is called, the ActionGroup abstracts away the work needed for a specific action to be invoked, by managing fetching user input, validation and executing the selected action.

## The Actor System

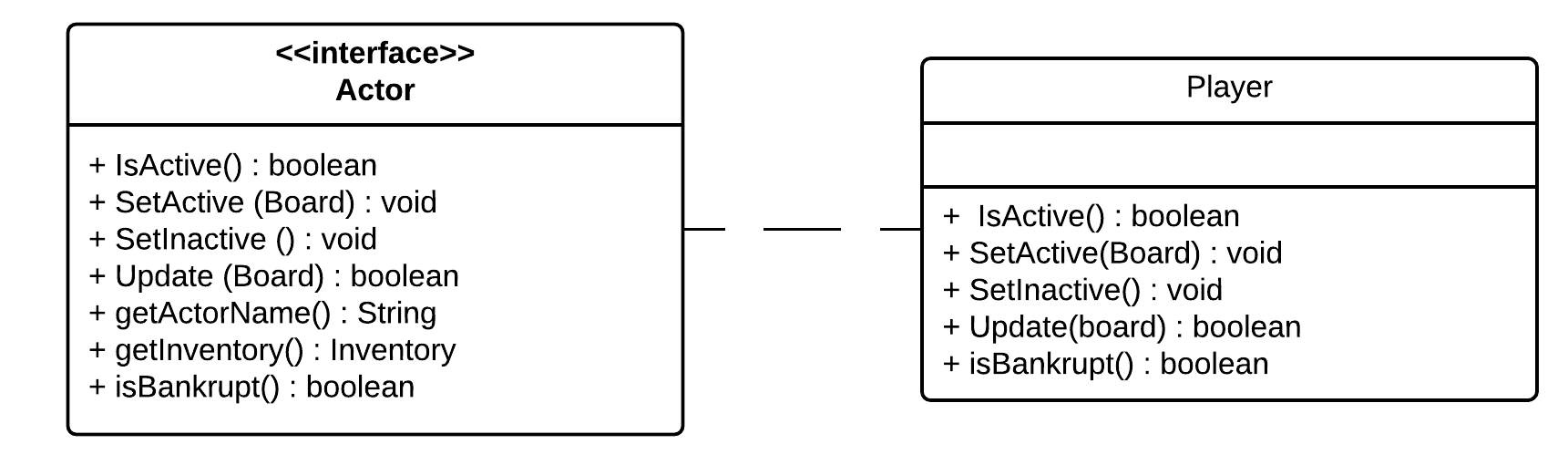
The actor system is a simple but flexible system that allows the game to be easily modified in the future. 

Figure 6. The actor system

A Player implements the Actor interface, where a Player is a human-controlled virtual character that can perform actions throughout the game. By defining this interface, one could extend the game further in the future, if desirable. For example, one could define a ComputerPlayer class and implement Actor, and implement Human vs. AI gameplay. The interface contains various methods that get called during play. Update() is the fundamental method that the game calls. Update() returns true when the Actor has ended its turn, and false when there is still more to do.

The IsActive(), SetActive() and SetInactive() methods are methods for checking and signaling that it is a specific Actors turn. The getInventory() method is a convenience method for accessing an Actors inventory, which contain its properties and balance. The setBankrupt() method is a convenience method that currently, based on the requirements, executes an EndGameAction that announces a winner, and ends the game.

## The Dice System

The dice system is a configurable system containing of an interface, configuration, and concrete implementations. The Dice interface exposes a roll() method for rolling a dice, which will return a value between getMinRoll() and getMaxRoll() in the DiceConfig, which is a JSON serializable configuration object. The configuration step allows configuration of the game to be more flexible, while the Dice interface allows us to implement custom dice.

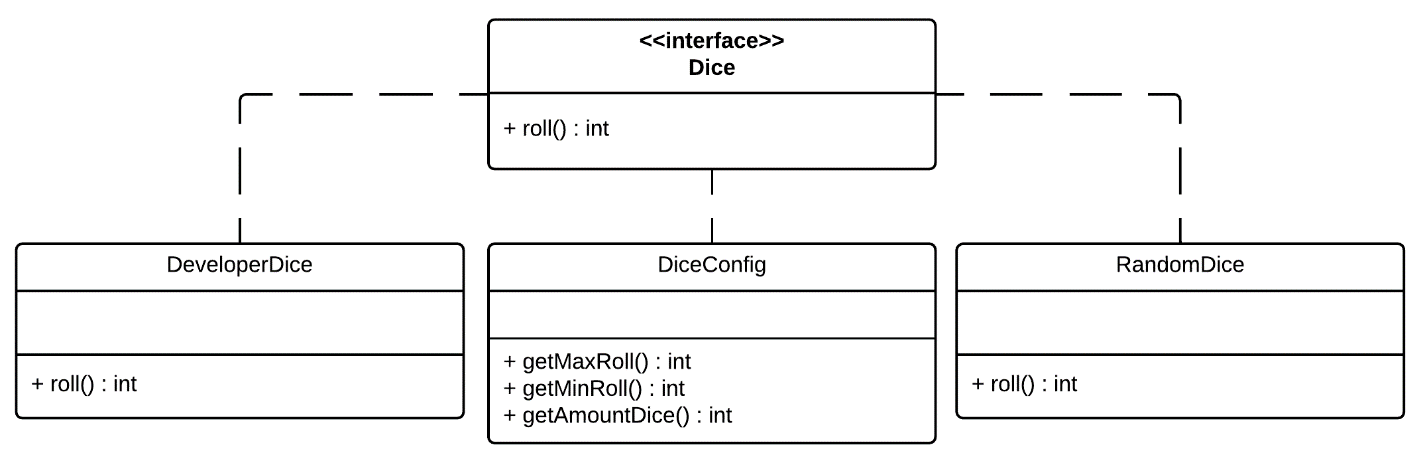


Figure 7. The dice system

For the game, we developed the DeveloperDice and the RandomDice; The first being a developer testing utility to allow us to manually specify a dice roll, and the latter being a random dice roll. However, this interface could potentially be extended in the future to add extra features to the game; perhaps landing on a specific tile grants you special dice?

## The Board System

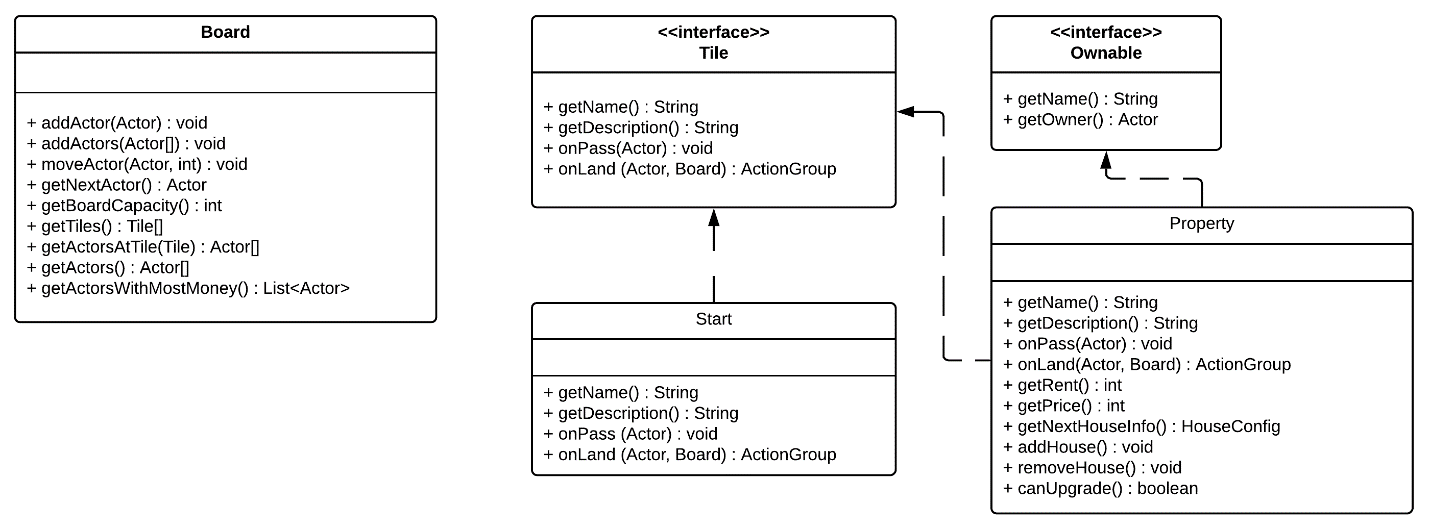
The board system is the core piece of functionality that combine the pieces of the systems into a functional game. The Board class contains Tile and Actor instances and encapsulates the functionality necessary for the Actors to move around the board. 

Figure 8. The board and tile system

The Board is consistent with a traditional controller, it is the primary interface utilized by the root Game class for updating the state of the game.

### The Tile System

The tile system is a subset of the board system; It defines the interfaces and concrete implementations of tiles that can exist on the board. The primary interface is the Tile interface, which defines simple getters and utilizes an event-driven approach to executing Actions. The getName() and getDescription() methods will return human-readable information about the tile, while the onPass(Actor) and onLand(Actor) are event-driven methods called by the Board class in the moveActor(Actor) method.

The reason the Board class does not implement an interface is because it should not be modified, nor extended. It is a core piece of functionality that sits at the heart of the system. Instead, the other systems, including the Tile system, are meant to be extended to add additional functionality. One could easily extend the game by adding new tiles and providing functionality in the onPass(Actor) and onLand(Actor) event methods and execute customs actions from here.

The Ownable interface was also defined for entities that are counted as being possible to own. While currently only currency and properties can be owned, this opens for the possibility of owning items that are not currently in the game. Perhaps a future developer would like to implement the popular trend of owning hats, or perhaps one can own different vehicles that give you higher dice rolls?

By defining the systems in such a manner, we are adhering to the open/closed principle, which states that software entities should be open for extension but closed for modification. We have defined a simple interface for extending the game, without the need for modifying the existing source code.

## The I/O System

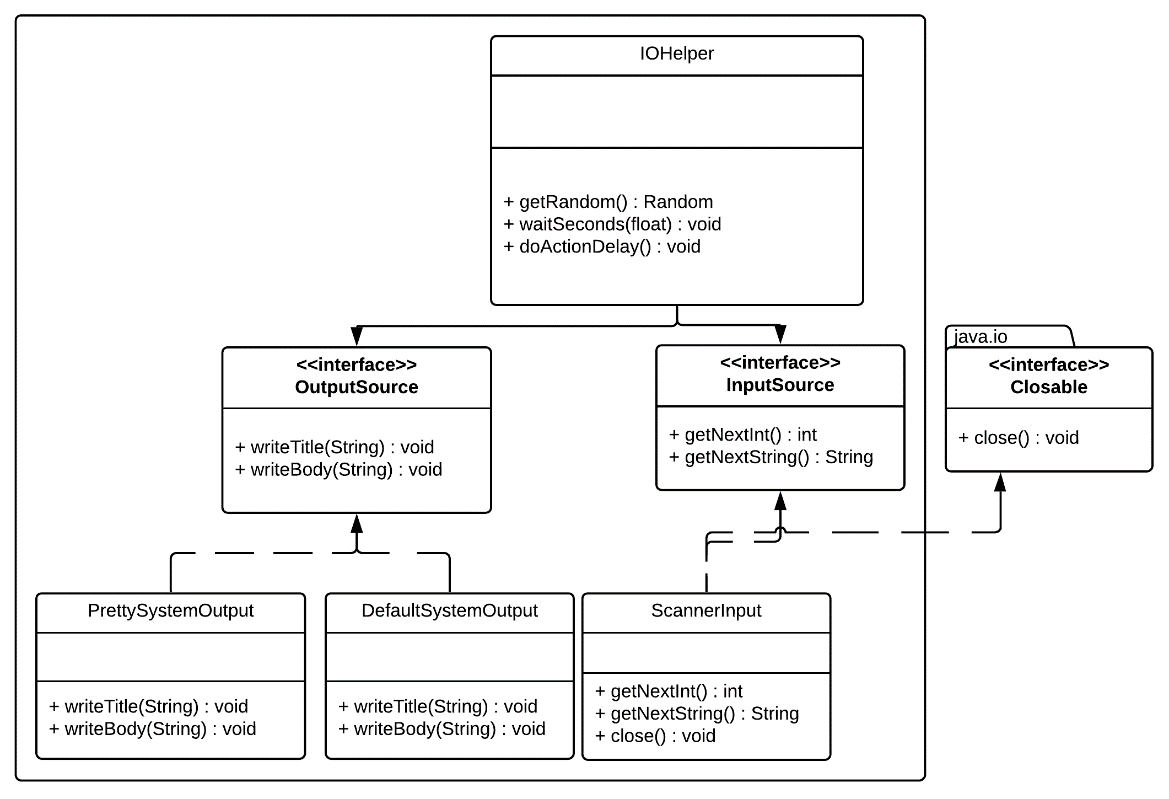
I/O Functionality can be fundamentally difficult to test, as mocking System defined behaviors is tricky. To combat this, we defined our own custom wrappers around common I/O functionality accessed throughout the game, such as writing to console and accepting user input.

Figure 9. The I/O system

The core interfaces in the I/O system is the OutputSource and InputSource interfaces. The output source interface defines two simple methods, writeTitle(String) for writing a title, and writeBody(String) for writing a body paragraph to the console. The primary use-case of defining this interface is for mocking purposes; By defining this interface, we can mock OutputSource under test to assert expected output.

An added benefit is that we can define our own custom text writers; For our current implementation, we have two concrete implementations for this, the DefaultSystemOutput class which writes simple console output, and the PrettySystemOutput class which writes formatted output to the console. This allows the customer to implement a custom OutputSource in the future, without having to change code in multiple places.

The InputSource interface is essential for mocking under test. By defining this interface and using it, we abstract away System.io scanner implementation, which allows us to define our own inputs and assert expected behavior by the system. This interface is used heavily throughout the application for JUnit testing for testing input validation.

For ease of use throughout the system, the IOHelper class provides a convenient way of accessing these interfaces without the need to define them over and over. This is especially important for the InputSource interface, as System.in (which is used by the Scanner used in the concrete ScannerInput implementation) is essentially a singleton, and the underlying Scanner is an unmanaged resource that implements the Closeable interface. By creating an instance singleton of the InputSource in the IOHelper, we prevent a memory leak by only keeping one instance of it.

Some other convenience methods also exist. The first method, getRandom(), is for ensuring there is only one instance of Random being used throughout the game. (Multiple Random instances lead to irregular behavior), and the waitSeconds(float) and doActionDelay() are convenience flavor methods for halting execution to allow a user to read text paragraph by paragraph, which can be enabled in configuration if desired by the customer.

## General System Design

The system has been designed in adherence with the SOLID design principles. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995).

* Single Responsibility Principle
* Open/Closed Principle
* Liskov Subtitution Principle
* Interface Segregation Principle
* Dependency Inversion Principle

We have discussed single responsibility, open/closed and dependency inversion throughout this design section. Additionally, the architecture follows the principle of substitution: This principle states that if an object is a subtype of another, then any other subtype may be substituted without altering any of the desirable properties of the program. This is demonstrated in all the interfaces in the application; No side-effects are introduced by swapping out implementations for another. The interface segregation principle states that no client should be forced to implement methods it does not need. As is demonstrated throughout this software application, interfaces are highly specific and specialized, and purely contain methods that apply to the type of object it is.

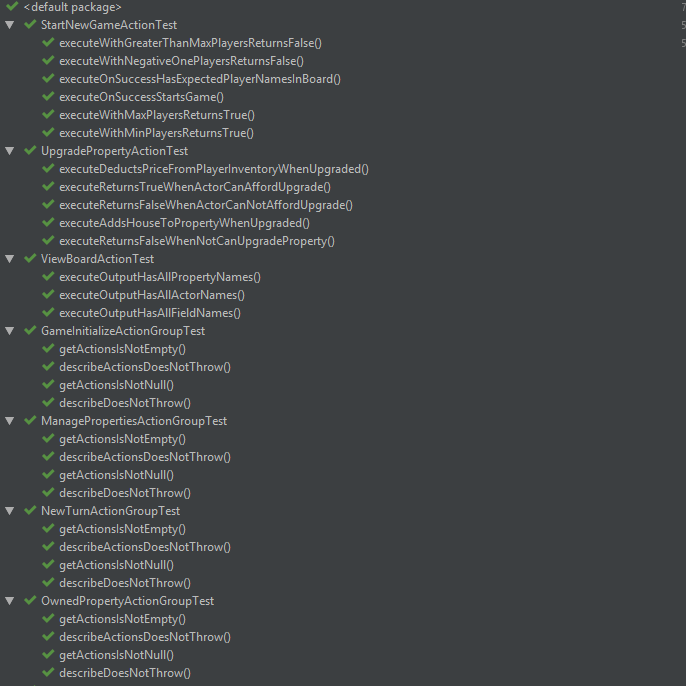
Other principles have been followed as well; throughout this programming exercise we have used Lombok , which is an annotation processor that automatically generates common functionality, such as getters and setters. This is a good example of the DRY (Don’t repeat yourself) principle in action. Project Lombok. (n.d.).

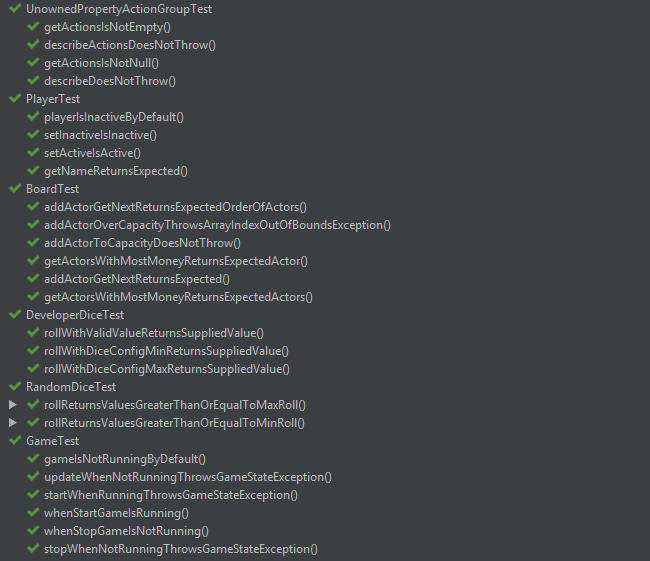
# Appendix

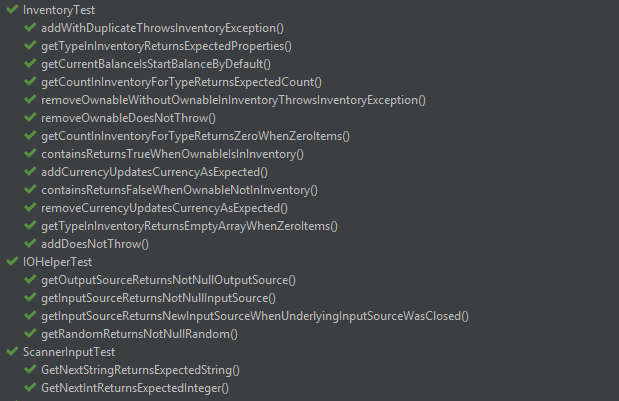
## Appendix 1. Test Plan

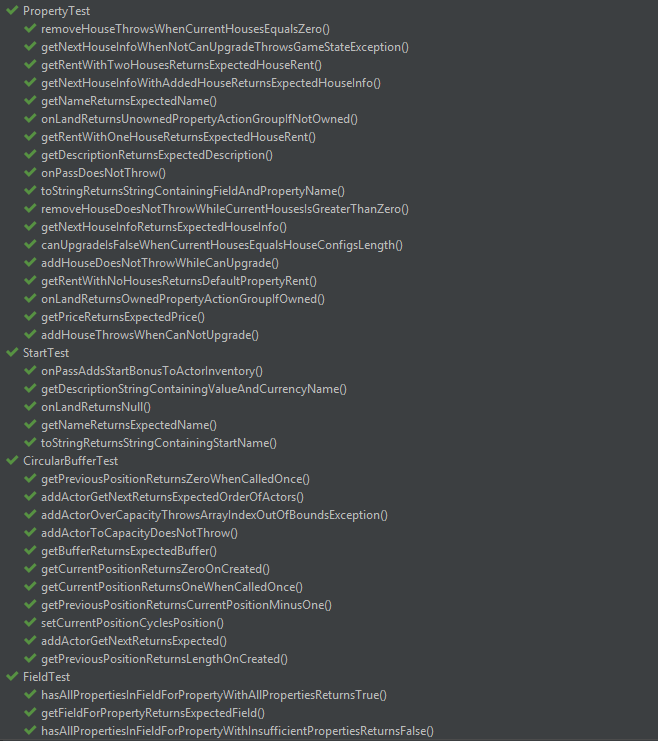
|  |  |  |
| --- | --- | --- |
| **SECTION** | **TEST NUMBER** | **DESCRIPTION** |
| Action |  |  |
| Start new game | 1 | Ensure restriction on allowed number of players |
|  | 2 | Ensure game loads when all players are entered |
|  | 3 | Ensure entered names are displayed when game starts |
| Upgrade new property | 1 | Check player is eligible to upgrade |
|  | 2 | Ensure sufficient funds before upgrading property |
|  | 3 | Ensure players bank account debited following transaction |
|  | 4 | Ensure house/office is added to property when upgraded |
| View board | 1 | Ensure all field names are displayed when called |
|  | 2 | Ensure all property names are displayed when called |
|  | 3 | Ensure all players names are displayed on board |
| Actor | 1 | Ensure when name is set player is active |
|  | 2 | Ensure correct name is displayed |
| Board | 1 | Ensure number of players are restricted to allowed number |
|  | 2 | Ensure names appear in relevant order |
| Dice | 1 | Ensure developer dice is functioning |
|  | 2 | Ensure player dice is restricted to number range |
| Inventory | 1 | Ensure inventory item allocated only once |
|  | 2 | Ensure bank account balances are updating as expected |
|  | 3 | Ensure inventory displaying correctly |
| Tile | 1 | Ensure selected property name displayed |
|  | 2 | Ensure relevant associated description is displayed |
|  | 3 | Ensure system displays if property owned/available |
|  | 4 | Ensure correct rent amount is displayed |
|  | 5 | Ensure correct property price is displayed |
|  | 6 | Ensure houses/offices can be added when eligible |
|  | 7 | Ensure houses/offices can be added when not eligible |
|  | 8 | Ensure system displays correctly following an upgrade |
|  | 9 | Ensure correct name is displayed |
|  | 10 | Ensure players account updated when passing start |
| Util | 1 | Ensure restriction on number of players |
|  | 2 | Ensure correct position of player is displayed |
|  | 3 | Ensure properties are associated with correct fields |
|  | 4 | Ensure game can be initialized and stopped |
| IO | 1 | Output source is not null |
|  | 2 | Input source is not null |
|  | 3 | Input source returns new instance when closed |
|  | 4 | Get random is not null |
|  | 5 | Get string from input source is expected string |
|  | 6 | Get next int from input source is expected int |
|  |  |  |

### JUnit Overview









## Appendix 2. Weekly Team Minutes

### Minutes for Group 3 Week commencing 28/01/2019\_ Date of this minute \_28/01/19

The following team members were present

|  |  |
| --- | --- |
| Name (printed/typed) | Signature |
| James Greene |  |
| Lars-Kristian Svenoey |  |
| Brendan Girvan |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Task Reporting (Briefly list the progress for each team member in the last week.\*)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

* N/A – Initial team meeting

Name & Role (2): James Greene Developer

* N/A – Initial team meeting

Name & Role (3): Brendan Girvan Developer

* N/A – Initial team meeting

Actions Planned (Briefly list the actions required of each team member for the next week.)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

* Decide on software and tools for project
* Prepare task allocations for developers

Name & Role (2): James Greene Developer

* Consider board layout and overall theme.
* Requirements analysis – identify core functionality components
* Review java programming language

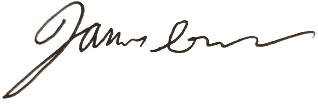
Name & Role (3): Brendan Girvan Developer

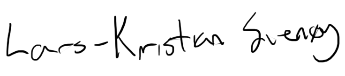
* Consider board layout and overall theme.
* Requirements analysis – identify core functionality components
* Review java programming language

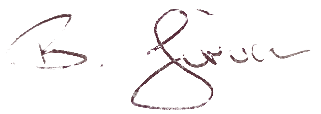
### Minutes for Group 3 Week commencing 04/02/2019\_ Date of this minute \_04/02/19

The following team members were present

Name (printed/typed) Signature

James Greene 

Lars-Kristian Svenoey 

Brendan Girvan 

Task Reporting (Briefly list the progress for each team member in the last week.\*)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

• Various software tools identified and discussed with team (Lombok, Maven)

• Role allocations for developers proposed.

Name & Role (2): James Greene Developer

• Board layout and theme proposed, discussed and agreed

• Reported on high level requirements for core functionality

Name & Role (3): Brendan Girvan Developer

• Board layout and theme proposed, discussed and agreed

• Reported on high level requirements for core functionality

Actions Planned (Briefly list the actions required of each team member for the next week.)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

• Set up project on gitlab utilising sourcetree and intellij

• Online meeting with developers to demo use of above software

Name & Role (2): James Greene Developer

• Collaborate with developer BG to develop use case descriptions

• Design initial use case diagrams

• Attend online meeting with project manager

Name & Role (3): Brendan Girvan Developer

• Collaborate with developer JG to develop use case descriptions

• Design initial use case diagrams

• Attend online meeting with project manager

### Minutes for Group 3 Week commencing 11/02/2019\_ Date of this minute \_11/02/19

The following team members were present

|  |  |
| --- | --- |
| Name (printed/typed) | Signature |
| James Greene |  |
| Lars-Kristian Svenoey |  |
| Brendan Girvan |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Task Reporting (Briefly list the progress for each team member in the last week.\*)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

* Project set up complete
* Developers able to access
* Developers provided with demo session
* JSON file created in IDE

Name & Role (2): James Greene Developer

* Use case descriptions and initial diagrams complete

Name & Role (3): Brendan Girvan Developer

* Use case descriptions and initial diagrams complete

Actions Planned (Briefly list the actions required of each team member for the next week.)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

* Develop Uml diagrams
* Write initial code for core game functionality

Name & Role (2): James Greene Developer

* Develop UML class diagrams from use case descriptions
* Write initial code Json configuration code
* Project manager to approve code and push to master file

Name & Role (3): Brendan Girvan Developer

* Develop UML class diagrams from use case descriptions
* Familiarise with gitlab
* Write initial code and upload to gitlab
* Project manager to approve code and push to master file

### Minutes for Group 3 Week commencing 18/02/2019\_ Date of this minute \_18/02/19

The following team members were present

|  |  |
| --- | --- |
| Name (printed/typed) | Signature |
| James Greene |  |
| Lars-Kristian Svenoey |  |
| Brendan Girvan |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Task Reporting (Briefly list the progress for each team member in the last week.\*)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

* Initial Uml class diagrams completed
* Code written for core interfaces

Name & Role (2): James Greene Developer

* Initial UML class diagrams completed
* Code for some classes developed
* More competent using gitlab

Name & Role (3): Brendan Girvan Developer

* Initial UML class diagrams completed
* Basic pseudo-code written for some classes, implemented

Actions Planned (Briefly list the actions required of each team member for the next week.)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

* Continue with code for game functionality

Name & Role (2): James Greene Developer

* Develop use case sequence diagrams
* Continue with code review for allocated classes

Name & Role (3): Brendan Girvan Developer

* Enter details in JSON file for game theme
* Continue reviewing code and testing for allocated classes

### Minutes for Group 3 Week commencing 25/02/2019\_ Date of this minute \_25/02/19

The following team members were present

|  |  |
| --- | --- |
| Name (printed/typed) | Signature |
| James Greene |  |
| Lars-Kristian Svenoey |  |
| Brendan Girvan |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Task Reporting (Briefly list the progress for each team member in the last week.\*)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

* Further code and documentation created

Name & Role (2): James Greene Developer

* Initial use case sequence diagrams completed
* Code for allocated classes written

Name & Role (3): Brendan Girvan Developer

* JSON file updated with board layout/theme
* Continuation of code development for classes

Actions Planned (Briefly list the actions required of each team member for the next week.)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

* Game functional – write code for added functionality
* Test plan
* Write Junit tests

Name & Role (2): James Greene Developer

* Initial test plan consideration
* Write Junit tests plans
* Identify and report any bugs in system

Name & Role (3): Brendan Girvan Developer

* Initial test plan consideration
* Implement junit tests
* Identify and report any bugs in system

### Minutes for Group 3 Week commencing 04/03/2019\_ Date of this minute \_04/03/19

The following team members were present

|  |  |
| --- | --- |
| Name (printed/typed) | Signature |
| James Greene |  |
| Lars-Kristian Svenoey |  |
| Brendan Girvan |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Task Reporting (Briefly list the progress for each team member in the last week.\*)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

* Code for added functionality created
* Tests run and errors identified

Name & Role (2): James Greene Developer

* Junit tests implemented and errors located

Name & Role (3): Brendan Girvan Developer

* Junit tests implemented and errors located

Actions Planned (Briefly list the actions required of each team member for the next week.)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

* Continue with code for added functionality
* Resolve coding errors

Name & Role (2): James Greene Developer

* Review code and game functionality
* Report on coding errors
* Ensure new functionality is incorporated into UML diagrams

Name & Role (3): Brendan Girvan Developer

* Review code
* Game testing
* Report bugs
* Review documentation

### Minutes for Group 3 Week commencing 11/03/2019\_ Date of this minute \_14/03/19

The following team members were present

|  |  |
| --- | --- |
| Name (printed/typed) | Signature |
| James Greene |  |
| Lars-Kristian Svenoey |  |
| Brendan Girvan |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Task Reporting (Briefly list the progress for each team member in the last week.\*)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

* Further functionality added to game
* Testing and error resolving ongoing
* Documentation reviewed and amended as necessary

Name & Role (2): James Greene Developer

* UML diagrams updated
* Code review and further error resolution
* Documentation up to date

Name & Role (3): Brendan Girvan Developer

* Added functionality tested
* Documentation reviewed and up to date
* Test plan reviewed

Actions Planned (Briefly list the actions required of each team member for the next week.)

Name & Role (1): Lars-Kristian Svenoey Project manager/Developer

* Collaboration with team to ensure full functionality of game
* Testing to be completed
* All documentation completed

Name & Role (2): James Greene Developer

* Ensure all code submitted
* All documentation per final report completed
* Game demo video to be completed

Name & Role (3): Brendan Girvan Developer

* Ensure all code submitted
* All documentation per final report completed
* Game demo video to be completed

## Process

The process used while developing the system was the waterfall model. The requirements were delivered upfront, and were not changed throughout the project, as is common in modern agile software engineering practices. We followed a rigid pattern, flowing from requirements, design, implementation and verification, and have now entered the maintenance stage in the life cycle of the project.

# Bibliography/References

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley. *Reading, MA*, 1995.

Project Lombok. (n.d.). Project Lombok. Retrieved March 14, 2019, from <https://projectlombok.org/>

PlantUML. (n.d.). Open-source tool that uses simple textual descriptions to draw UML diagrams.. Retrieved March 14, 2019, from http://plantuml.com/