Preface

This document defines the requirements for the Monopoly Game, a command-line-based simulation of the classic board game designed for COMP3211 Software Engineering. This SRS is intended for both developers and testers.

Introduction

The Monopoly Game application simulates a simple version of the traditional Monopoly Board Game which includes basic components like players, squares on the board, turn taking, rolling a die, etc. The game accommodates 2 to 6 players, who take turns rolling a dice, moving across a board, purchasing properties, and paying rent. The game's objective is to outlast opponents financially, either by holding the most money after 100 rounds or being the last player after all the other players have lost their money.

Glossary

Square	A position on the board where players can land.
PropertySquare	A type of square representing a property that can be bought, owned, and on which rent can be collected.
Player	A participant in the game with attributes like money, properties, and positions on the board.
Game	Manages the overall game mechanics, such as player turns and board state.
Go	A type of square where players receive a salary when they pass.
Change	A type of square that causes players to gain or lose money.
Jail	A type of square where players may be held temporarily.

User requirements definition

UR1	Users should be able to start a new game game with a predefined board
UR2	Users should be able to name players using either input or randomly generated names
UR3	Users should be able to display the status of specific players or all players
UR4	Users should be able to show the overall game status, including player positions and board configurations
UR5	Users should be able to see the next player to take a turn
UR6	Users should be able to save the current game state to a file
UR7	Users should be able to load a saved game state and continue from where it left off
UR8	Users should be able to customize a game by adding, modifying, or organizing different squares
UR9	Users should be able to load and modify and existing game board
UR10	Users should be able to save custom game board configuration

System requirements

Functional Requirement

FR1	System shall initialize the game board with 20 predefined squares, each of a specified type
FR2	System shall allow players to roll two dice, advancing their tokens based on a dice outcome
FR3	System shall enable players to purchase properties when they land on unowned properties, provided they have enough funds
FR4	System shall charge rent to players landing on properties owned by others, with rent amount depending on the property’s location
FR5	System shall reward players with a salary when they pass the Go square
FR6	System shall implement the “Go to Jail” square, restricting player movement until they a fine or roll doubles within three turns.
FR7	System shall end the game if only one player remains or after 100 rounds, determining the winner by the highest money balance.

Non Functional Requirement

Performance	The game should handle user inputs and process game events with minimal delay
Usability	The command-line interface should prompt users for inputs clearly and display game state updates in a user-friendly format
Reliability	The game should handle erroneous inputs gracefully and maintain game consistency, including player balances, property ownership, and turn sequence
Extensibility	The design should allow for additional features without major refactoring

System models

The design will use an object-oriented model with classes representing key game components:

- Game: Manages overall game state, player turns, and win conditions;
- Player: Tracks individual player attributes such as balances, properties, and board properties;
- Square (abstract class): Base class for board squares, providing an interface for specific square types;
- PropertySquare (subclass of square): Manages property-specific attributes like price, rent, and ownership;

System evolution

Assumptions

The game rules and board configuration will remain fixed in the current version. However, the system should allow easy modification for future changes, such as adjusting board size, introducing new square types, or updating gameplay rules.

Future changes

The design should support future expansions, such as adding different property types or incorporating community chest and chance cards with more complex outcomes.

Appendices

Environment Requirements

The game should run on a standard Java runtime environment with a command-line interface for input and output.