Sprint 1 Documentation

Summary Data

- **Team Number:** 13
- **Team Lead:** Chris
- **Sprint Start:** 03/02/2020
- **Sprint End:** 10/02/2020

Individual Key Contributions

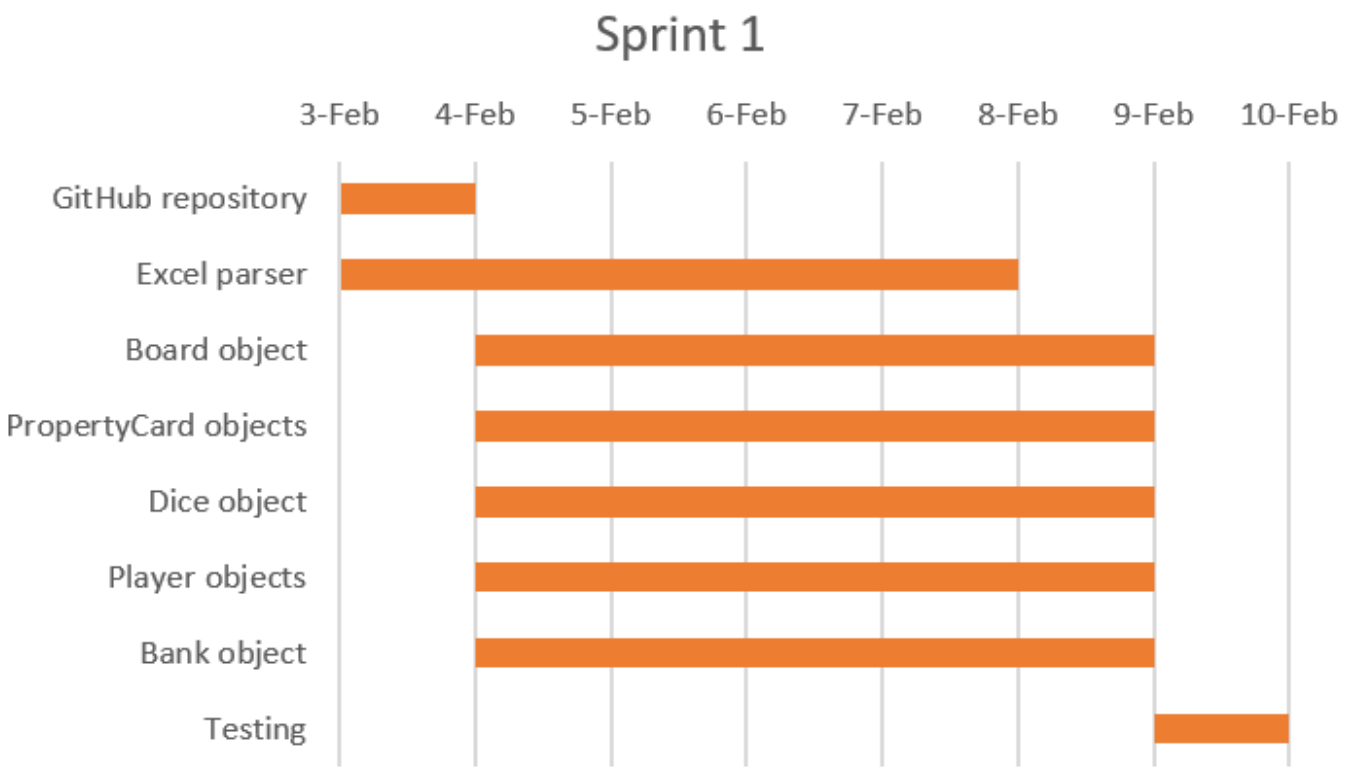
Team Member	Key Contributions
Aiden	Documentation & Implementation
Ankeet	Implementation
Chris	Organisation & Implementation
Duarte	Implementation

Task Cards

- Establish GitHub repository
- Create property cards
- Develop Excel parser to input information about property cards
- Create individual players
- Create board with property cards from parser
- Create dice that a player can roll
- Create bank that will hold money and properties

The image below shows the tasks set out on Trello during our weekly meeting

Gantt Chart



Requirements Analysis

Functional Requirements

- F1
  - The software should be able to automatically parse an Excel file to create the board tiles. The each board tile shall include the name, the associated group, any actions to be performed, the price of the property, the rent with and without houses and each individual house cost. An additional parameter should be added to indicate if the board tile is an ownable property by a player. The parser shall return a list of board tiles in the order in the Excel document.
- F2
  - The software shall have players that play the game. The amount of players in a single game shall be between 2 - 6 players. Each player should have a name, a balance, a list of ownable properties, their location on the board, whether the player is in jail and the amount of doubles rolled during a turn. A player shall start with £1,500 in their balance. A player shall be able to roll dice to update their location on the board. The sum of all rolls is summed together and player moves that amount of spaces forward. If the player rolls the dice and both dice have the same number, the player shall roll again. If this continues after rolling 3 consecutive doubles, the player is moved to jail. The player should be able to see what actions they can perform at each instant during their turn. The player shall buy a property when they land on that property. The player shall also sell properties from any location during their turn. When a player is buying a property, their balance shall be deducted by the price of the property and the property shall be added to their list of owned properties. The selling of a property shall deposit the price of the property into the player's balance and the property is removed from their list of owned properties.
- F3
  - The software shall construct a board built upon the board tiles created by the parser. The board will hold all the board tiles associated in the game.
- F4
  - The software shall have dice that should be rolled. Each die is 6-sided; therefore, a number between 1 - 6 shall be returned when rolled.
- F5
  - The software shall have a bank that initially owns all of the properties that can be owned and a sum of money. At the start of the game, the bank shall start with £50,000 in its balance and all ownable properties. If the bank cannot withdraw enough funds to give to a player, the bank shall automatically replenish its funds by adding an additional £50,000. The bank shall add and remove properites from its list of ownable properties.

Non-Functional Requirements

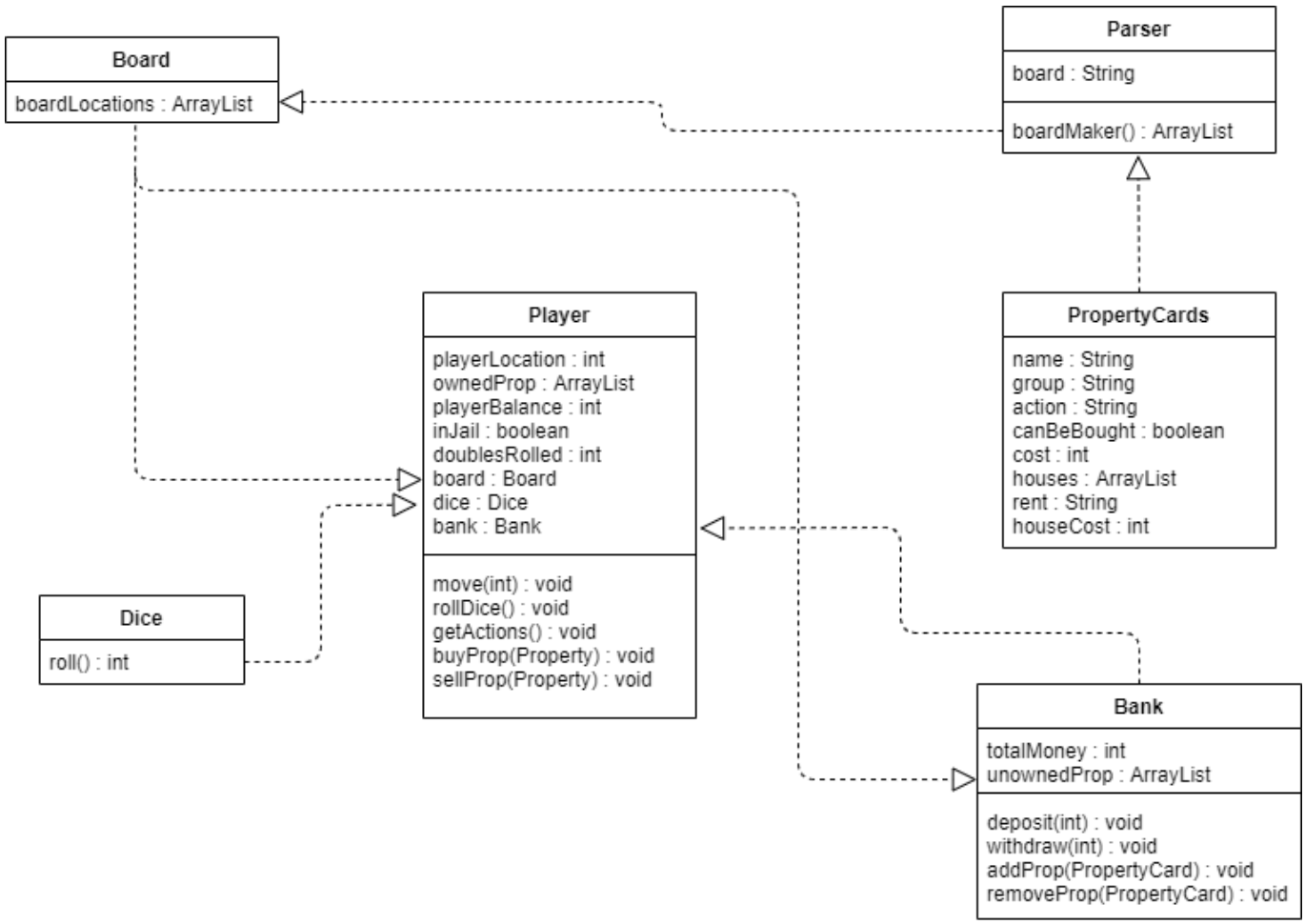
- NF1
  - The language the software shall be written in Java.
- NF2
  - Maven shall be used to manage the project including but not limited to managing dependencies and building the parser.
- NF3
  - The implementation of the project will vary between the IDEs of NetBeans and Intellij.

Domain Requirements

- D1
    - The version control of the project should be stored on GitHub, iteration by iteration. Need more information on how to edit and update repository.

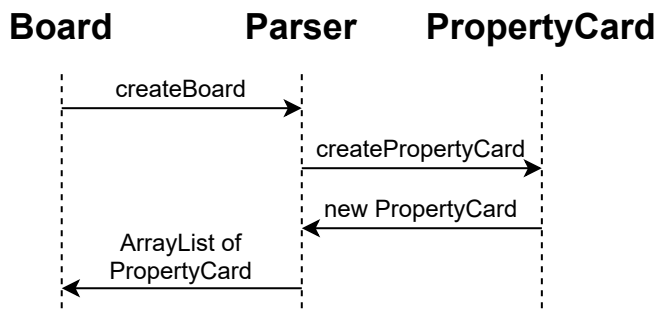
Design

UML Diagram

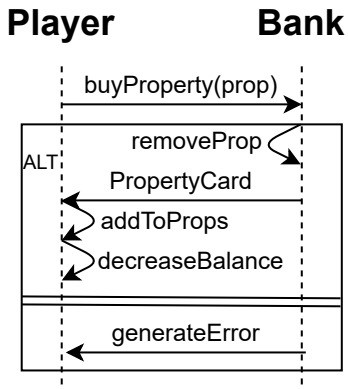


Sequence Diagrams

Creating the board



Buying a property



Test Plan

With this sprint being the first, we chose to not have system-wide testing due to the face that there was no 'main' class that would control these objects. Apart from that, all objects had a JUnit test with multiple test methods to ensure the object was behaving propertly, correctly and error-free. This will apply to every sprint until a function GUI is implemented.

With the `Player` test class, we have created tests that determine whether:

1. rolling the dice advances the player
2. ensures moving completely around the board does not result in an `IndexOutOfBoundsException`
3. buying a property results in the specific property being added to the player's owned properties

```
/**
 * Test that rolling the dice will cause the player to move to that location that it is not on;
 */
@Test
public void testRollDice() throws IOException, InvalidFormatException {
    ArrayList<Integer> a = new ArrayList<Integer>();
    a.add(1);
    PropertyCards propertyCards = new PropertyCards("test", "green", "", true, 1000, "10", a, 100);
    ArrayList<PropertyCards> properties = new ArrayList<PropertyCards>();
    properties.add(propertyCards);
    Bank bank = new Bank(properties);
    Dice dice = new Dice();
    Board board = new Board();
    Player player = new Player(board, dice, bank);

    player.rollDice();
    Assertions.assertTrue(player.getPlayerLocation() != 0);
}
```

With the `PropertyCard` test class, the tests that are implemented show that:

1. ensure the accessors return the correct values

```
/**
 * Test of getName method, of class PropertyCards.
 */
@Test
public void testGetName() {
    System.out.println("getName");
    String expResult = "Sainsburys";
    String result = pc.getName();
    assertEquals(expResult, result);
}

/**
 * Test of getGroup method, of class PropertyCards.
 */
@Test
public void testGetGroup() {
    System.out.println("getGroup");
    String expResult = "Indigo";
    String result = pc.getGroup();
    assertEquals(expResult, result);
}
```

With the `Parser` test class, the sole responsibility of this test class to ensure:

1. the correct number of PropertyCards were being created
2. the list of PropertyCards were in the correct order

```
/**
 * Test of boardMaker method, of class Parser.
```

```
     *
     * @throws java.lang.Exception
     */
    @Test
    public void testBoardMaker() throws Exception {
        System.out.println("boardMaker");
        ArrayList<String> expResult = new ArrayList<>
        expResult.add("Go");
        expResult.add("Crapper Street");
        expResult.add("Pot Luck");
        expResult.add("Gangsters Paradise");
        expResult.add("Income Tax");
        expResult.add("Brighton Station");
        expResult.add("Weeping Angel");
        expResult.add("Opportunity Knocks");
        expResult.add("Potts Avenue");
        expResult.add("Nardole Drive");
        expResult.add("Jail/Just visiting");
        expResult.add("Skywalker Drive");
        expResult.add("Tesla Power Co");
        expResult.add("Wookie Hole");
        expResult.add("Rey Lane");
        expResult.add("Hove Station");
        expResult.add("Cooper Drive");
        expResult.add("Pot Luck");
        expResult.add("Wolowitz Street");
        expResult.add("Penny Lane");
        expResult.add("Free Parking");
        expResult.add("Yue Fei Square");
        expResult.add("Opportunity Knocks");
        expResult.add("Mulan Rouge");
        expResult.add("Han Xin Gardens");
        expResult.add("Falmer Station");
        expResult.add("Kirk Close");
        expResult.add("Picard Avenue");
        expResult.add("Edison Water");
        expResult.add("Crusher Creek");
        expResult.add("Go to Jail");
        expResult.add("Sirat Mews");
        expResult.add("Ghengis Crescent");
        expResult.add("Pot Luck");
        expResult.add("Ibis Close");
        expResult.add("Lewes Station");
        expResult.add("Opportunity Knocks");
        expResult.add("Hawking Way");
        expResult.add("Super Tax");
        expResult.add("Turing Heights");
        ArrayList<String> result = new ArrayList<>();
        for (PropertyCards pc : board) {
            result.add(pc.getName());
        }
```

```
    assertEquals(expResult, result);
}
```

With the `Board` test class, the only test method needed was that the Parser passed on the PropertyCards to the Board object.

```
@Test
public void getBoardLocations() {
    assertEquals("Go", board.getBoardLocations().get(0).getName());
}
```

With the `Bank` test class, our team ensured that:

1. the initial balance of the bank is £50,000
2. depositing a certain amount of money resulted in adding said money to the bank's balance
3. withdrawing a certain amount of moeny resulted in subtracting said money from the bank's balance
4. if the bank senses that a transaction will make its balance becomes negative, the bank replenishes itself to £50,000
5. a property can be removed the bank's list of properties
6. a property can be added back to the bank's list of properties

```
/**
 * Deposit test
 */
@Test
public void testDeposit() {
    b.deposit(500);
    assertEquals(b.getMoney(), 50500);
}

/**
 * Withdraw test
 */
@Test
public void testWithdraw() {
    int cash = b.withdraw(500);
    System.out.print(b.getMoney());
    assertEquals(cash, 500);
    assertEquals(b.getMoney(), 49500);
    int cash2 = b.withdraw(49500);
    assertEquals(b.getMoney(), 0);
    int cash3 = b.withdraw(500);
    assertEquals(b.getMoney(), 49500);
}
```

With the `Dice` test class, the sole method ensured that the amount shown on the dice was between 1 and 6.

```
@Test
public void testDiceRoll() {
    Dice dice = new Dice();
    Assertions.assertTrue(dice.roll() <= 6 && dice.roll() >= 1);

}
```

Summary of Sprint

The team started strong. We designed and set up the procedures for how we would work as a team including how to use git effectively as well as the ideas of an agile methodology. We began to design the parser system and got a very basic version working.

The strengths of this sprint was the willingness of the team and the motivation to get a sprint done enabling us to complete this sprint quicker then we originally planned. Additionally, we had a good initial build of the game with clear and concise goals that were easily met.

A downside to having done this sprint quickly was that we did not set enough work for us; we felt underworked. Furthermore, we should have spent more time designing the game and the structure of its componenents in the long term so the goals for future sprints would have been more clear.