Object design example: snakes and ladders game

Joan Serrat

September 16, 2014

Abstract

We'll take a simple child's game specification as a running example for the design process and design principles. We'll start by discovering the problem domain classes and assign them key responsibilities (grasp patterns) while maintaining low coupling and high cohesion. New *software* domain classes will appear to support responsibilities that can not be placed at problem domain classes. There will be also the opportunity to apply further design principles like "favor composition over inheritance" and "program to an interface, not to an implementation", in order make the design more resistant to requirement changes. We will see that responsibilities will be implemented by a large number of cooperating small methods. This example draws from the slides *Object-oriented principles* by O.Nierstrasz, at University of Bern, though some changes have been introduced in the design.

1 Problem

Everyone should know the snakes and ladders game (figure 1a). The final goal is to build a computer application with a graphical user interface such that human players and maybe also the computer can play the game. At first, however, well just build a simulator, to make a good design of the application core and test its implementation. Later, we should add a graphical user interface and a controller to mediate between it and the core classes. Anyway, a simulator may be useful on its own, for instance to calculate the probability (frequency) that a player reaches the last square after at most N turns on a board with $M = 100$ squares, as shown in figure 1b.

This simulator accepts the following input

- number, name and order of the players
- total number of squares
- ladders and snakes, specified as initial position and length

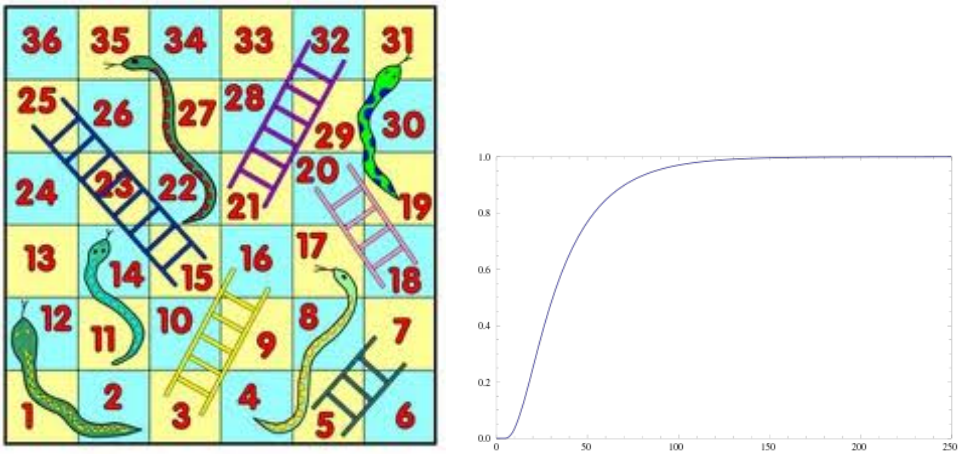
and prints to console all the details of the game: at each turn,

- the state of the game (number of the square each player is in)
- name of the current player and the rolled dice number
- the new position of the player, and if he/she falls into a ladder or snake, it tells so and the from-to square numbers
- name of the winner when any

2 Domain classes and their responsibilities

From the brief problem description, which are the domain classes ? And what should they know and do ? From these questions we will derive an initial class design, with attributes and methods. Then, we will think on how classes fulfill their responsibilities through methods and messages.

We can easily identify the following classes and responsibilities:



(a) (b)

Figure 1: a) board, b) y probability of reaching the end on a 100 squares board after at most x turns.

- **Dice**
  1. produces integer random numbers between 1 and 6
- **Player**
  1. knows his/her name
  2. knows how to move him/herself, that is, leave its current square and enter into another one, a certain number of squares ahead
  3. knows the square he/she is at
- **Square**
  1. knows its number or position
  2. knows which player is in it, if any (zero or one)
  3. therefore, can accept and remove a player from itself (that is, one can ask it to do that)
  4. knows what to do if the square already contains one player and another falls on it: send him to the first square
  5. given a number rolled by the dice, knows how to compute where the player is going to land on : the square with number equal to present square number + dice value, provided he does not surpass the last square
  6. in this later case, the computation results in staying at the present square
- **First square** : same as **Square** plus
  1. knows it can have zero, one or more players
  2. keeps (add, remove) this set of players
- **Last square** : same as **Square** plus
  1. knows it's an special square (but not why), the last one, and therefore can answer about this
- **Ladder/ Snake**
  1. knows that a player landing on it has to be moved to some square ahead/back of itself a certain amount of positions (can compute the forwarding position)

3 Software domain classes

Yet, the former list of responsibilities is not complete. For instance, who *creates* the players, the squares, ladders and snakes ? Who knows about the players order and the current player

at each time ? Who rolls dice and tells the current player to move accordingly ? Who updates the current player ? Who knows when to stop playing because there is already a winner ?

All this high-level responsibilities have to be assigned to some class, but considering the low cohesion principle none of the problem domain classes should take them. Therefore, we create a new class to take them all. It's an artificial or software-domain class in the sense of not corresponding to a real-world concept like Player, Square etc. Let's call it **Game**.

4 A first design

Let's create the classes and methods supporting the identified responsibilities (figure 2). First thing to comment is the decision to represent the different types of squares (according to the game rules, that is, its "behaviour" with regard to the players) as subclasses of a class **Square**. It represents the regular square and thus provides the default implementation for some methods. Subclasses override (redefine, but not extend) some of them. On top of it, we have defined an interface **ISquare**. This is according to the principle of "program to an interface, not to an implementation": objects using squares do not know there are different types of squares but only that they match the **ISquare** interface. Also, this is according to the "open-close" principle: we could add new types of squares conforming to that interface and nothing would change.

Each square has a position (integer number) and in addition ladders/snakes have a **transport** property, the positive/negative shift a player undergoes when he lands on them. Note that all types of square have zero or one player except first square which may have more (**players** is a list) and therefore the method **isOccupied** needs to be redefined).

The design in figure 2 also shows some associations and their multiplicities. Let's see why are they needed:

- **Main.main()** instantiates a **Game**, passing to the constructor the necessary parameters (players names, number of squares and snakes and ladders position and transport data) to create the board and the players before starting the game.
- **Game** contains the list of squares and players
- Each *Player* object has to know the square he/she is in, not only the position but the **Square** object because it will send messages to it like leave it or ask it to move the player because it is the square who knows the rules of the game on player movements: add position and dice numbers, the "go home square" rule, the need to land at the last square with the exact dice number.
- A square needs to know if it contains players or not in order to apply the "go home square" rule
- The first square needs to know its players in order
- A square needs to know the **Game** object in order to access other squares, like when asking for some square ahead where a player must land, as we will see.

How does everything start ? Let's write the code for the designed methods.

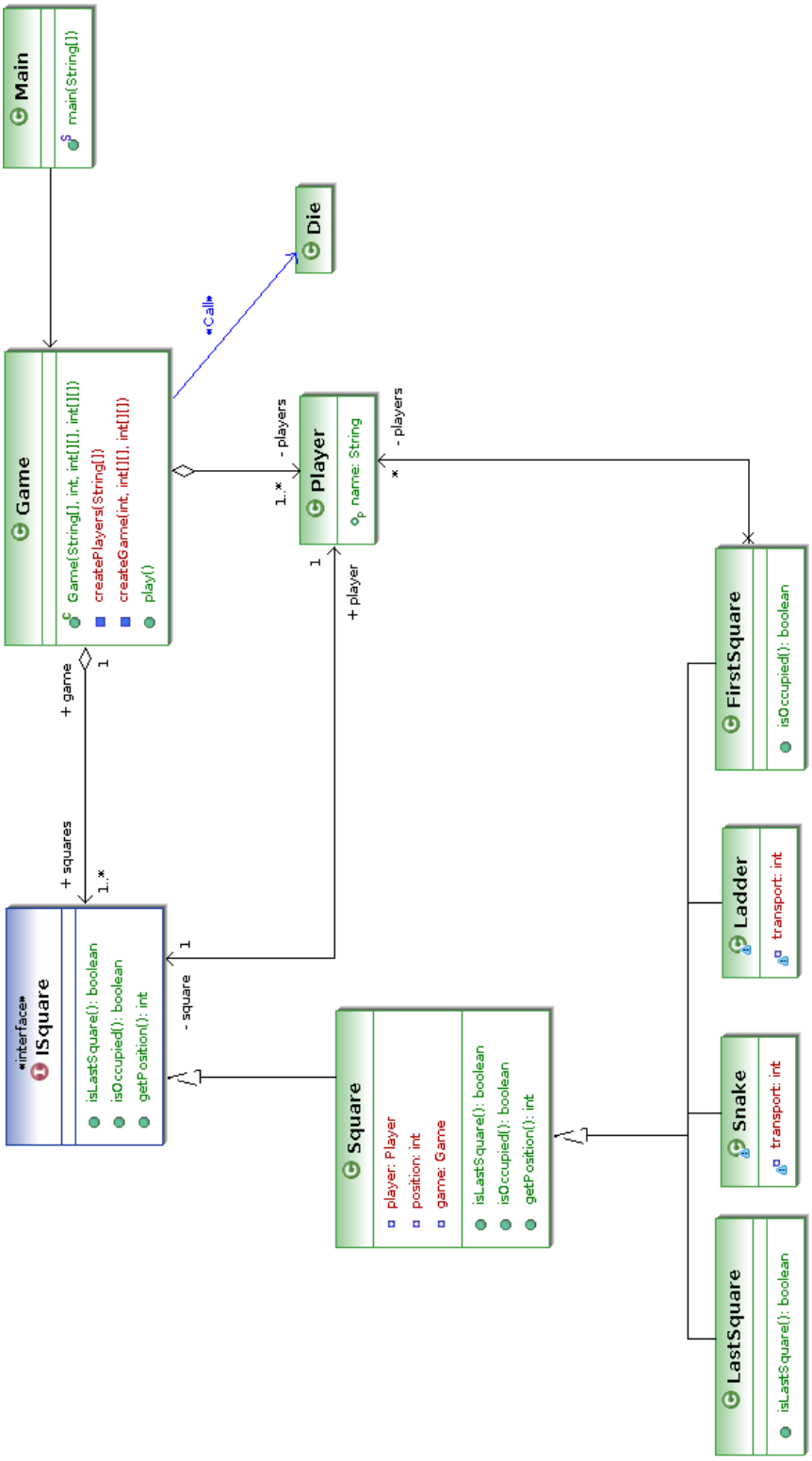


Figure 2: First design. Note that (a) **Square** is not abstract, it represents a regular square so we introduce the interface **ISquare** to "program to an interface, not an implementation". (b) The **Game** object knows its squares, but a square needs to know its **Game** in order to get the new square for a player after rolling the dice. Game has also (not shown) methods `firstSquare()` and `findSquare(int pos)` which return an **ISquare**.

```
 // replaces the user interface, that is, hardcodes it.
public static void main(String[] args) {
  String[] playerNames = {"Maria", "Abel","Nuria","Joan"};
  int numSquares = 12;
  // for the user first square is at position 1 but internally is at 0
  int[][] snakesFromTo = { {11,5} };
  int[][] laddersFromTo = { {2,6} , {7,9} };

  // no need to assign to a local variable
  new Game(playerNames, numSquares, laddersFromTo, snakesFromTo);
}
```

Listing 1: firstDesign/Main.main()

```
public Game(String[] playerNames,
    int numSquares, int[][] ladders, int[][] snakes) {
  createPlayers(playerNames);
  createGame(numSquares, ladders, snakes);
  play();
}
```

Listing 2: firstDesign/Game.Game()

```
private void createGame(int numSquares, int[][] ladders, int[][] snakes) {
    // make first, last and regular squares
  squares.add(new FirstSquare(0,this));
  for (int position=1 ; position<numSquares-1 ; position++) {
    Square square = new Square(position, this);
    squares.add(square);
  }
  squares.add(new LastSquare(numSquares-1,this));

  // make snake squares which replace some already created square
  for (int i=0; i<snakes.length ; i++) {
      assert snakes[i].length == 2;
      // snakes and ladders are defined by pairs of positions
    int fromPosition = snakes[i][0]-1;
    int toPosition = snakes[i][1]-1;
    int transport = toPosition - fromPosition;
    squares.set(fromPosition, new Snake(fromPosition,this, transport));
  }

  for (int i=0; i<ladders.length; i++) {
    assert ladders[i].length == 2;
    int fromPosition = ladders[i][0]-1;
    int toPosition = ladders[i][1]-1;
    int transport = toPosition - fromPosition;
    squares.set(fromPosition, new Ladder(fromPosition,this,transport));
  }
  // TODO: redundant code, could snakes be the same as ladders but with a
  // negative transport ?
}
```

Listing 3: firstDesign/Game.createGame()

The constructor of **Game**, after creating the squares and the players, calls **play()** to start the game:

```
    // Note: the code *tells* you what it is doing, not how does it do it
    // (the details)
  public void play() {
    Die die = new Die();

    for (Player player : players) {
      firstSquare().enter(player);
    }
    winner = null;

    System.out.println("Initial state : \n" + this);
    while (notOver()) {
      int roll = die.roll();
      System.out.println("Current player is " + currentPlayer()
          + " and rolls " + roll);
      movePlayer(roll);
      System.out.println("State : \n" + this);
    }
    System.out.println(winner + " has won.");
  }

  private boolean notOver() {
    return winner == null;
  }

  private void movePlayer(int roll) {
    Player currentPlayer = players.remove(); // from the head of the list
    currentPlayer.moveForward(roll);
    players.add(currentPlayer); // to the end of the list: it's a queue
    if (currentPlayer.wins()) {
      winner = currentPlayer;
    }
  }
```

Listing 4: firstDesign/Game.play()

We have reached the key point: how do players move themselves on the board, following the game rules ? and who enforces these rules ? As suggested by **Player.moveForward(roll)**, a player can be told to move a certain number of squares, and he/she can land into the last square, a ladder, a snake or a regular square already occupied (or not). But the player hasn't to know these rules, its the responsibility of the squares. So, within the **Player** class we have

```
  private ISquare square = null;

  public void moveForward(int moves) {
    assert moves>0 : "non-positive moves";

    // the player asks to the square  he/she is in that it moves him/her
    square.leave(this);
    square = square.moveAndLand(moves);
    square.enter(this);
  }
```

Listing 5: firstDesign/Player.moveForward()

Now, in the **Square** class we have the rules of the game related to how players move :

```
1  public ISquare moveAndLand(int moves) {
2    int lastPosition = game.findLastSquare().getPosition();
3    // the game object knows (holds a list of) all the squares, and each
4    // square has a reference to the game object in order to ask it which
5    // is the number of the last square, get the square at 'moves' positions
6    // ahead, and get the next square whatever it is ('moves' ahead or home
7    // square)
8    if (position+moves>lastPosition) {
9      System.out.println("Should go to " + (position+moves+1)
10         + " beyond last square " + (lastPosition+1)
11         + " so don't move");
12     return this;
13   } else {
14     System.out.println("move from " + (position+1) + " to "
15         + (findRelativeSquare(moves).getPosition()+1));
16     return findRelativeSquare(moves).landHereOrGoHome();
17   }
18 }
19
20 private ISquare findRelativeSquare(int moves) {
21     // this is a forwarding method to make code more readable, making the code
22     // tell ''what'', not ''how''
23     // note: it returns some type of square, that is, an ISquare
24   return game.findSquare(moves+position);
25 }
26
27 public ISquare landHereOrGoHome() {
28     // note: it returns some type of square, that is, an ISquare
29   if (isOccupied()) {
30     System.out.println("square " + (position+1) + " is occupied");
31   } else {
32     System.out.println("land at " + (position+1));
33   }
34   return isOccupied() ? findFirstSquare() : this;
35 }
36
37 private ISquare findFirstSquare() {
38     // another forwarding method to make code more readable
39   return game.firstSquare();
40 }
```

Listing 6: firstDesign/Square.moveAndLand()

While writing the code of the main methods, others have been introduced, like findFirstSquare() or notOver(). Figure 3 shows the augmented version of the first design.

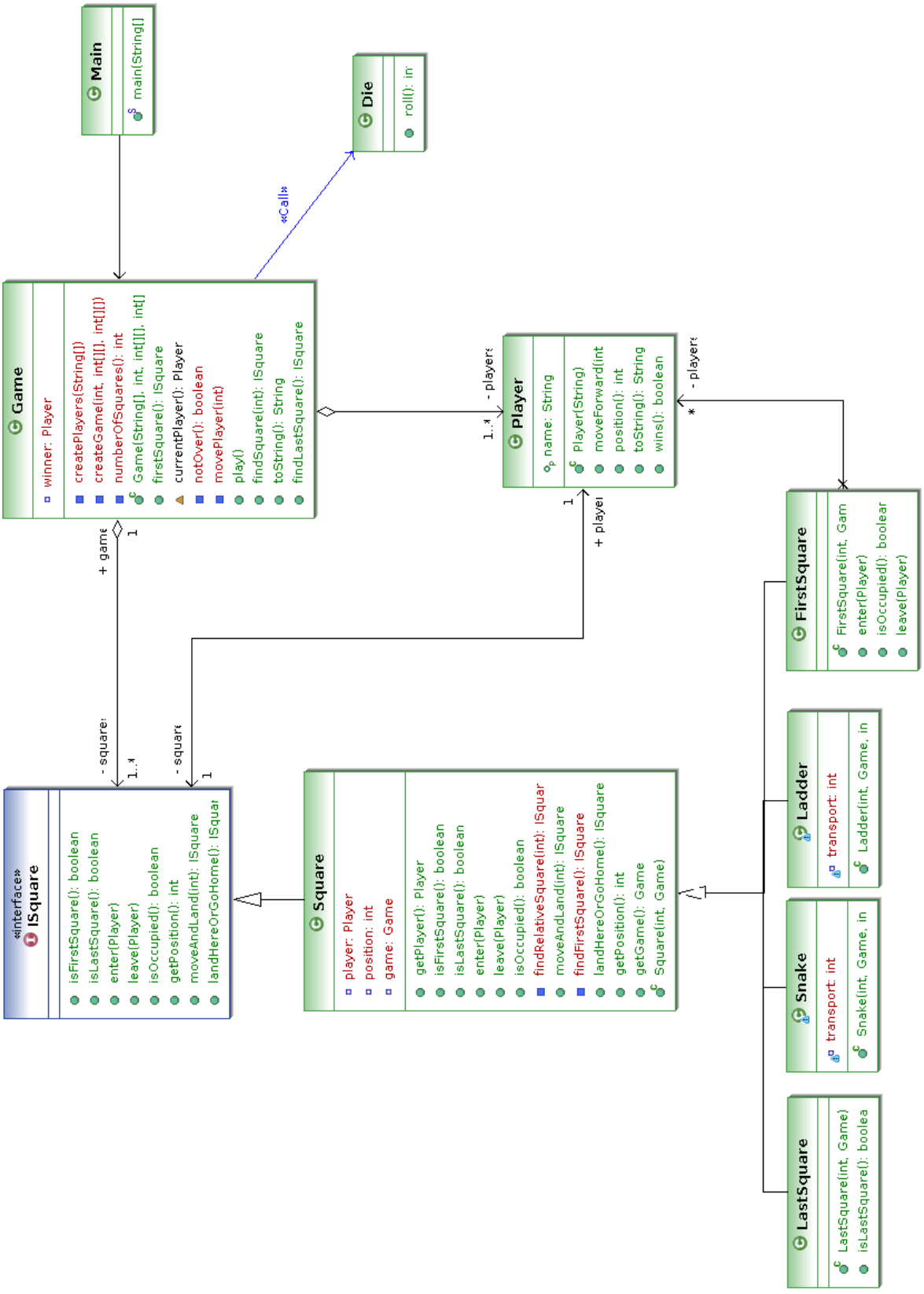


Figure 3: First design augmented.

5 Design second round

The previous design has several problems with regard to the design patterns Grasp and some others. For example, class Game is quite big, specially method createGame(). This is because it does too much. We could add a class **Board** instead of the simple list of squares in **Game**, and move several methods there to simplify the **Game** class. Second, the hierarchy derived from **ISquare** is it all right according the the Coad rules on when to create subclasses ? Are the different types of squares extending the methods of the parent class ? Or are all the same only that behave according to different rules ? Figure 4 tries to address these problems.

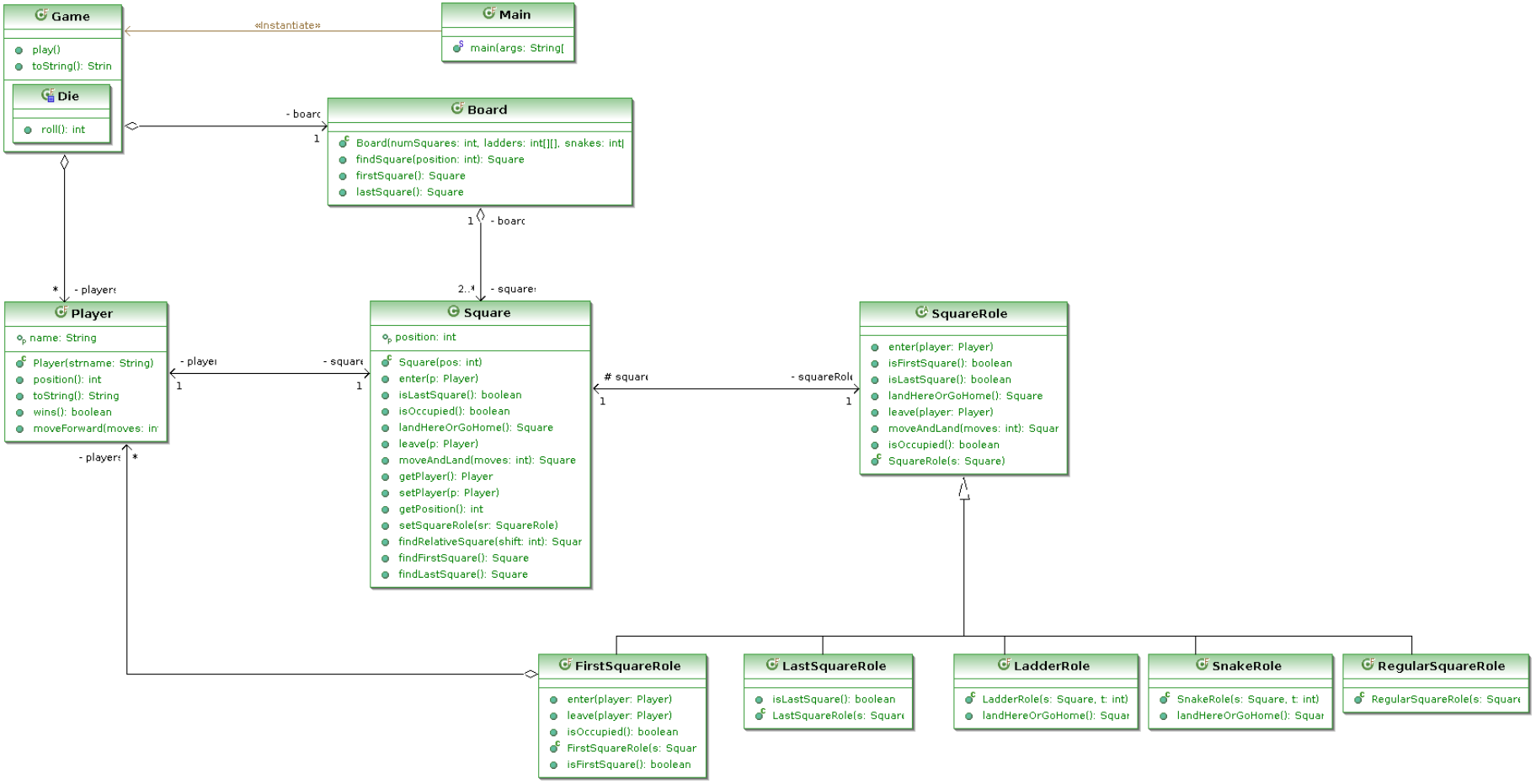


Figure 4: Second design with square roles instead of square subclasses.

6 Second design implementation

```java
package problemDomain;

public final class Main {

  public static void main(String[] args) {
    // TODO these parameters have to be read and checked for
    // validity from the console, that is, the command line.
    String[] playerNames = {"Monica", "Albert","Noemi","Jaume"};
    int numSquares = 12;
    // for the user first square is at position 1 but
        // internally is at 0
    int[][] snakesFromTo = { {11,5} };
    int[][] laddersFromTo = { {2,6} , {7,9} };

    Game game = new Game(playerNames, numSquares, snakesFromTo,
                            laddersFromTo);
    game.play();
  }
}
```

Listing 7: listings2ndDesign/Main.java

```java
package problemDomain;

import java.util.LinkedList;
import java.util.Random;

public final class Game {
  private LinkedList<Player> players = new LinkedList<Player>();
    // this is a queue: elements are removed from the beginning
    // with players.remove() and added to the end by players.add()
  private Board board = null;
  private Player winner;

  private final class Die {
    private static final int MINVALUE = 1;
    private static final int MAXVALUE = 6;

    public int roll() {
      return random(MINVALUE,MAXVALUE);
    }

    private int random(int min, int max) {
      assert min<max;
      return (int) (min + Math.round((max-min) * Math.random()));
    }
  }

  private void movePlayer(int roll) {
    Player currentPlayer = players.remove(); // from the head
    currentPlayer.moveForward(roll);
    players.add(currentPlayer); // to the tail
    if (currentPlayer.wins()) {
      winner = currentPlayer;
    }
  }

  public Game(String[] playerNames, int numSquares,
      int[][] snakes, int[][] ladders) {
    makeBoard(numSquares, ladders, snakes);
```

```
39      makePlayers(playerNames);
40    }
41
42    private void makeBoard(int numSquares, int[][] ladders,
43              int[][] snakes) {
44      board = new Board(numSquares,ladders,snakes);
45    }
46
47    private void makePlayers(String[] playerNames) {
48      assert playerNames.length>0 : "There must be some player" ;
49      System.out.println("Players are : ");
50      int i=1;
51      for (String str : playerNames) {
52        Player player = new Player(str);
53        players.add(player); // adds to the end
54        System.out.println(i + ". " + str);
55        i++;
56      }
57    }
58
59    public void play() {
60      assert !players.isEmpty() : "No players to play";
61      assert board!=null : "No scoreboard to play";
62
63      Die die = new Die();
64      startGame();
65
66      System.out.println("Initial state : \n" + this);
67      while (notOver()) {
68        int roll = die.roll();
69        System.out.println("Current player is " + currentPlayer()
70            + " and rolls " + roll);
71        movePlayer(roll);
72        System.out.println("State : \n" + this);
73      }
74      System.out.println(winner + " has won.");
75    }
76
77    private void startGame() {
78      placePlayersAtFirstSquare();
79      winner = null;
80    }
81
82    private void placePlayersAtFirstSquare() {
83      for (Player player : players) {
84        board.firstSquare().enter(player);
85      }
86    }
87
88    private boolean notOver() {
89      return winner == null;
90    }
91
92    @Override
93    public String toString() {
94      String str = new String();
95      for (Player player : players) {
96        str += player.getName() + " is at square "
97                    + (player.position()+1) + "\n";
98      }
99      return str;
100   }
```

```
101
102   Player currentPlayer() {
103     assert players.size()>0;
104     return players.peek();
105   }
106 }
```

Listing 8: listings2ndDesign/Game.java

```
1  package problemDomain;
2
3  public final class Player {
4    private Square square = null;
5    private String name;
6
7    public Square getSquare() {
8      return square;
9    }
10
11   public void setSquare(Square square) {
12     this.square = square;
13   }
14
15     public int position() {
16       return square.getPosition();
17     }
18
19   public String getName() {
20     return name;
21   }
22
23   public Player(String strname) {
24     name = strname;
25   }
26
27   @Override
28   public String toString() {
29     return name;
30   }
31
32   public boolean wins() {
33     return square.isLastSquare();
34   }
35
36   public void moveForward(int moves) {
37     assert moves>0 : "non-positive moves";
38     square.leave(this);
39     square = square.moveAndLand(moves);
40     square.enter(this);
41   }
42 }
```

Listing 9: listings2ndDesign/Player.java

```
1  package problemDomain;
2  import java.util.ArrayList;
3
4  public final class Board {
5      private ArrayList<Square> squares = new ArrayList<Square>();
6      private static int MINNUMSQUARES = 10;
7
8      public Board(int numSquares, int[][] ladders, int[][] snakes) {
```

```
9           assert numSquares > MINNUMSQUARES : "There must be at least "
10              + MINNUMSQUARES + " squares";
11          makeSquares(numSquares);
12          makeLadders(ladders);
13          makeSnakes(snakes);
14      }
15
16      private void makeSquares(int numSquares) {
17          System.out.println("There are " + numSquares + " squares");
18          for (int position=0 ; position<numSquares ; position++) {
19              Square square = new Square(position, this);
20              squares.add(square);
21              square.setSquareRole(new RegularSquareRole(square));
22          }
23          firstSquare().setSquareRole(new FirstSquareRole(firstSquare()));
24          lastSquare().setSquareRole(new LastSquareRole(lastSquare()));
25      }
26
27      public Square firstSquare() {
28          return squares.get(0);
29      }
30
31      public Square lastSquare() {
32          return squares.get(squares.size()-1);
33      }
34
35      public Square findSquare(int position) {
36          assert (position>0) && (position<numberOfSquares()) : "inexistent square";
37          return squares.get(position);
38      }
39
40      private int numberOfSquares() {
41          assert !squares.isEmpty();
42          return squares.size();
43      }
44
45      private void makeSnakes(int[][] snakes) {
46          for (int i=0; i<snakes.length ; i++) {
47              assert snakes[i].length == 2;
48
49              int fromPosition = snakes[i][0]-1;
50              int toPosition = snakes[i][1]-1;
51              int transport = toPosition - fromPosition;
52
53              assert transport<0 : "In snake, destination after origin";
54              assert (toPosition > 0) && (toPosition<numberOfSquares()-1);
55              assert (fromPosition < numberOfSquares()-1) && (fromPosition>0);
56
57              System.out.println("snake from " + (fromPosition+1)
58                      + " to " + (toPosition+1));
59
60              Square snakeSquare = squares.get(fromPosition);
61              snakeSquare.setSquareRole(new SnakeRole(snakeSquare,transport));
62          }
63      }
64
65      private void makeLadders(int[][] ladders) {
66          for (int i=0; i<ladders.length; i++) {
67              assert ladders[i].length == 2;
68
69              int fromPosition = ladders[i][0]-1;
70              int toPosition = ladders[i][1]-1;
```

```
71              int transport = toPosition - fromPosition;
72
73              assert transport>0 : "In ladder, origin after destination";
74              assert (toPosition < numberOfSquares()) && (toPosition > 0);
75              assert (fromPosition > 0) && (fromPosition < numberOfSquares());
76
77              System.out.println("ladder from " + (fromPosition+1)
78                      + " to " + (toPosition+1));
79
80              Square ladderSquare = squares.get(fromPosition);
81              ladderSquare.setSquareRole(new LadderRole(ladderSquare,transport));
82          }
83      }
84  }
```

Listing 10: listings2ndDesign/Board.java

```
1  package problemDomain;
2
3  public class Square {
4    private Board board = null;
5    private Player player = null;
6    private int position;
7    private SquareRole squareRole = null;
8
9    public Square(int pos, Board b) {
10     assert pos>=0 : "Square number must be positive or zero" ;
11     position = pos;
12     board = b;
13   }
14
15   public Player getPlayer() {
16     return player;
17   }
18
19   public void setPlayer(Player p) {
20     player = p;
21   }
22
23   public int getPosition() {
24     return position;
25   }
26
27   public void setSquareRole(SquareRole sr) {
28     assert sr!=null;
29     squareRole = sr;
30   }
31
32   public boolean isOccupied() {
33     return squareRole.isOccupied();
34   }
35
36   public boolean isLastSquare() {
37     return squareRole.isLastSquare();
38   }
39
40   public Square moveAndLand(int moves) {
41     return squareRole.moveAndLand(moves);
42   }
43
44   public Square landHereOrGoHome() {
45     return squareRole.landHereOrGoHome();
```

```
46    }
47
48    public void enter(Player p) {
49      squareRole.enter(p);
50    }
51
52    public void leave(Player p) {
53      squareRole.leave(p);
54    }
55
56    public Square findRelativeSquare(int shift) {
57      return board.findSquare(position + shift);
58    }
59
60    public Square findFirstSquare() {
61      return board.firstSquare();
62    }
63
64    public Square findLastSquare() {
65      return board.lastSquare();
66    }
67  }
```

Listing 11: listings2ndDesign/Square.java

```
1  package problemDomain;
2
3  import java.util.LinkedList;
4
5
6  public abstract class SquareRole {
7    protected Square square = null;
8
9    public SquareRole(Square s) {
10     assert s!=null : "Null square for square role";
11     square = s;
12   }
13
14   public boolean isOccupied() {
15     return square.getPlayer() != null;
16   }
17
18   public boolean isFirstSquare() {
19     return false;
20   }
21
22   public boolean isLastSquare() {
23     return false;
24   }
25
26   public Square moveAndLand(int moves) {
27     int lastPosition = square.findLastSquare().getPosition();
28     int presentPosition = square.getPosition();
29     if (presentPosition+moves>lastPosition) {
30       System.out.println("Should go to "
31                   + (presentPosition+moves+1)
32           + " beyond last square " + (lastPosition+1)
33           + " so don't move");
34       return square;
35     } else {
36       System.out.println("move from "
37                   + (square.getPosition()+1) + " to "
```

```
38            + (square.findRelativeSquare(moves).getPosition()+1));
39        return square.findRelativeSquare(moves).landHereOrGoHome();
40      }
41    }
42
43    public Square landHereOrGoHome() {
44      if (square.isOccupied()) {
45        System.out.println("square " + (square.getPosition()+1)
46                           + " is occupied");
47      } else {
48        System.out.println("land at " + (square.getPosition()+1));
49      }
50      return square.isOccupied() ? square.findFirstSquare() : square;
51    }
52
53    public void enter(Player player) {
54      square.setPlayer(player);
55      player.setSquare(square);
56    }
57
58    public void leave(Player player) {
59      square.setPlayer(null);
60    }
61  }
```

Listing 12: listings2ndDesign/SquareRole.java

```
1  package problemDomain;
2
3  public final class RegularSquareRole extends SquareRole {
4
5    public RegularSquareRole(Square s) {
6      super(s);
7    }
8  }
```

Listing 13: listings2ndDesign/RegularSquareRole.java

```
1  package problemDomain;
2
3  import java.util.ArrayList;
4
5  public final class FirstSquareRole extends SquareRole {
6
7    private ArrayList<Player> players = new ArrayList<Player>();
8
9    public FirstSquareRole(Square s) {
10     super(s);
11   }
12
13   @Override
14   public boolean isFirstSquare() {
15     return true;
16   }
17
18   @Override
19   public void enter(Player player) {
20     players.add(player);
21     player.setSquare(square);
22   }
23
24   @Override
```

```
25   public void leave(Player player) {
26     players.remove(player);
27   }
28
29   @Override
30   public boolean isOccupied() {
31     return !players.isEmpty();
32   }
33 }
```

Listing 14: listings2ndDesign/FirstSquareRole.java

```
1 package problemDomain;
2
3 public final class LastSquareRole extends SquareRole {
4
5   public LastSquareRole(Square s) {
6     super(s);
7   }
8
9   @Override
10  public boolean isLastSquare() {
11    return true;
12  }
13 }
```

Listing 15: listings2ndDesign/LastSquareRole.java

```
1 package problemDomain;
2
3 public final class LadderRole extends SquareRole {
4   private int transport;
5
6   public LadderRole(Square s, int t) {
7     super(s);
8     assert t>0 : "A ladder shift must be positive";
9     transport = t;
10  }
11
12  @Override
13  public Square landHereOrGoHome() {
14    System.out.println("ladder from " + (square.getPosition()+1)
15                + " to " + (destination().getPosition()+1));
16    return destination().landHereOrGoHome();
17  }
18
19  private Square destination() {
20    return square.findRelativeSquare(transport);
21  }
22 }
```

Listing 16: listings2ndDesign/LadderRole.java

```
1 package problemDomain;
2
3 public final class SnakeRole extends SquareRole {
4   private int transport;
5
6   public SnakeRole(Square s, int t) {
7     super(s);
8     assert t<0 : "A snake shift must be negative" ;
9     transport = t;
```

```
10    }
11
12    @Override
13    public Square landHereOrGoHome() {
14      System.out.println("snake from " + (square.getPosition()+1)
15                  + " to " + (destination().getPosition()+1));
16      return destination().landHereOrGoHome();
17    }
18
19    private Square destination() {
20      return square.findRelativeSquare(transport);
21    }
22  }
```

Listing 17: listings2ndDesign/SnakeRole.java