

Projet de programmation : Zen l'Initié

Cahier de conception



Table des matières

1) Diagramme de classe : analyse	3
2) Diagramme de classe : conception.....	5
3) Diagramme de séquence boîte noir.....	7
4) Spécification des fichiers	10
5) Squelette du projet	11
a) Package model	11
1. GameConfig.....	11
2. Game	12
3. Player.....	17
4. Human	18
5. Computer	19
6. Square	20
7. Pawn.....	22
8. Color	24
9. Difficulty	24
10. Direction	25
11. Mode.....	25
b) Package control	26
1. Prompt.....	26
c) Package util	27
1. Save.....	27
d) Launcher	28
6) Test unitaires (JUnit)	29
a) Test du package model	29
1. TestGameConfig.....	29
2. TestGame	30
3. TestHuman	34
4. TestComputer.....	35
5. TestSquare	37
6. TestPawn	39
b) Test du package util	41
1. TestSave	41
7) ANT	42

```

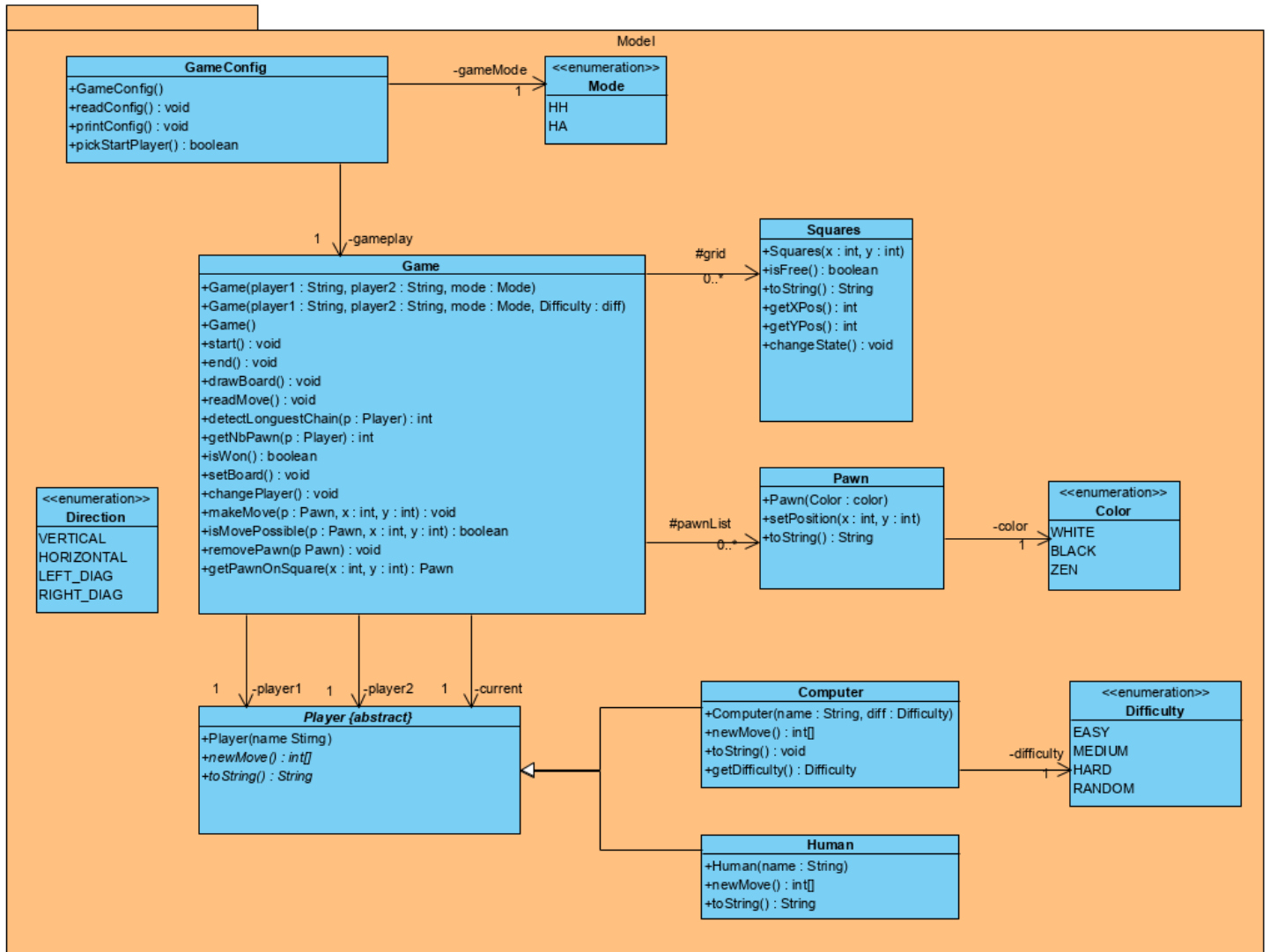
classDiagram
    class Launcher {
        +main(String[]) void
    }
    class Prompt {
        +inputMode() Mode
        +inputName() String
        +inputCoordinates() int[]
        +askForQuit() char
    }
    class Save {
        +writeSave(fileName: String)
        +readSave(fileName: String) Game
    }
    class GameConfig {
        +GameConfig()
        +readConfig() void
        +printConfig() void
        +pickStartPlayer() boolean
    }
    class Game {
        +Game(player1: String, player2: String, mode: Mode)
        +Game(player1: String, player2: String, mode: Mode, Difficulty: diff)
        +Game()
        +start() void
        +end() void
        +drawBoard() void
        +readMove() void
        +detectLongestChain(p: Player) int
        +getNbPawn(p: Player) int
        +isWon() boolean
        +setBoard() void
        +changePlayer() void
        +makeMove(p: Pawn, x: int, y: int) void
        +isMovePossible(p: Pawn, x: int, y: int) boolean
        +removePawn(p: Pawn) void
        +getPawnOnSquare(x: int, y: int) Pawn
    }
    class Squares {
        +Squares(x: int, y: int)
        +isFree(x: int, y: int) boolean
        +toString() String
        +getxPos() int
        +getyPos() int
        +changeState() void
    }
    class Pawn {
        +Pawn(Color: color)
        +setPosition(x: int, y: int)
        +toString() String
    }
    class Player {
        <<abstract>>
        +Player(name: String)
        +newMove() int[]
        +toString() String
    }
    class Computer {
        +Computer(name: String, diff: Difficulty)
        +newMove() int[]
        +toString() void
        +getDifficulty() Difficulty
    }
    class Human {
        +Human(name: String)
        +newMove() int[]
        +toString() String
    }
    class Direction {
        <<enumeration>>
        VERTICAL
        HORIZONTAL
        LEFT_DIAG
        RIGHT_DIAG
    }
    class Color {
        <<enumeration>>
        WHITE
        BLACK
        ZEN
    }
    class Difficulty {
        <<enumeration>>
        EASY
        MEDIUM
        HARD
        RANDOM
    }

    Launcher --> Game
    Prompt --> Game
    Save --> Game
    GameConfig --> Game : -gameMode 1
    Game --> Squares : #grid 0..*
    Game --> Pawn : #pawnList 0..*
    Game --> Player : -player1 1, -player2 1, -current 1
    Player <|-- Computer
    Player <|-- Human
    Computer --> Difficulty : -difficulty 1
    
```

```
graph TD
    Game[Game] --> Control[Control]
    Game --> Launcher[Launcher]
    Game --> Util[Util]
    Control --> Prompt[Prompt]
    Launcher --> LauncherClass[Launcher]
    Util --> Save[Save]
```

Game

- Control**
 - Prompt**
 - +inputMode() : Mode
 - +inputName() : String
 - +inputCoordinates() : int[]
 - +askForQuit() : char
- Launcher**
 - +main(String[]) : void
- Util**
 - Save**
 - +writeSave(fileName : String)
 - +readSave(fileName : String) : Game



Package Model

Ce diagramme de classe, dans sa version d'analyse, permet de comprendre rapidement le fonctionnement général de l'application future.

2) Diagramme de classe : conception

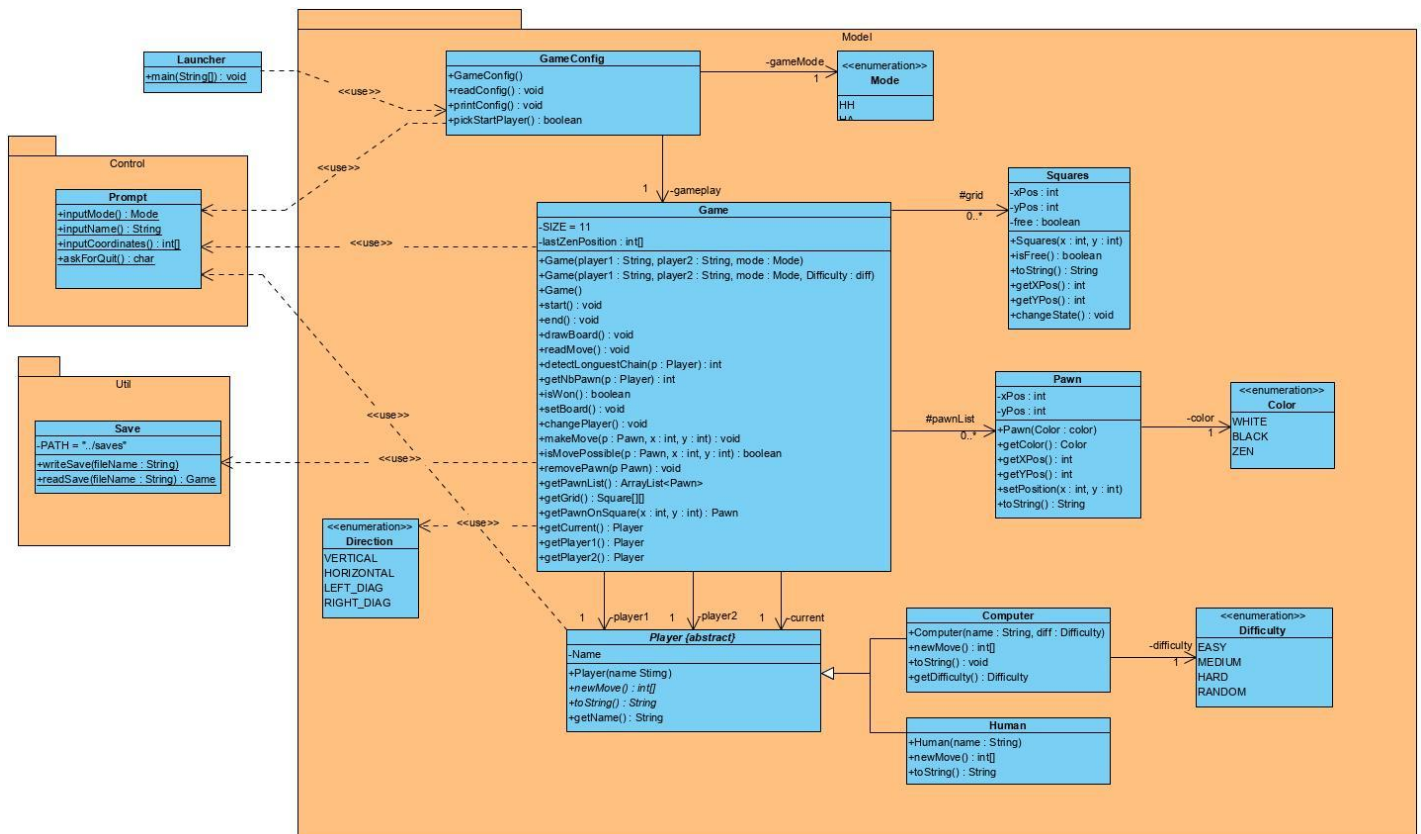
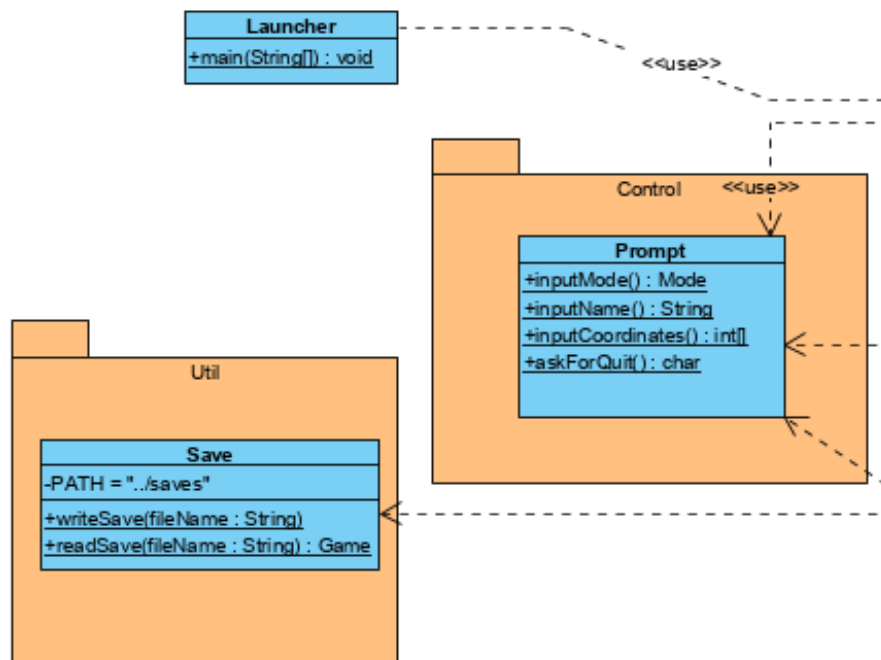
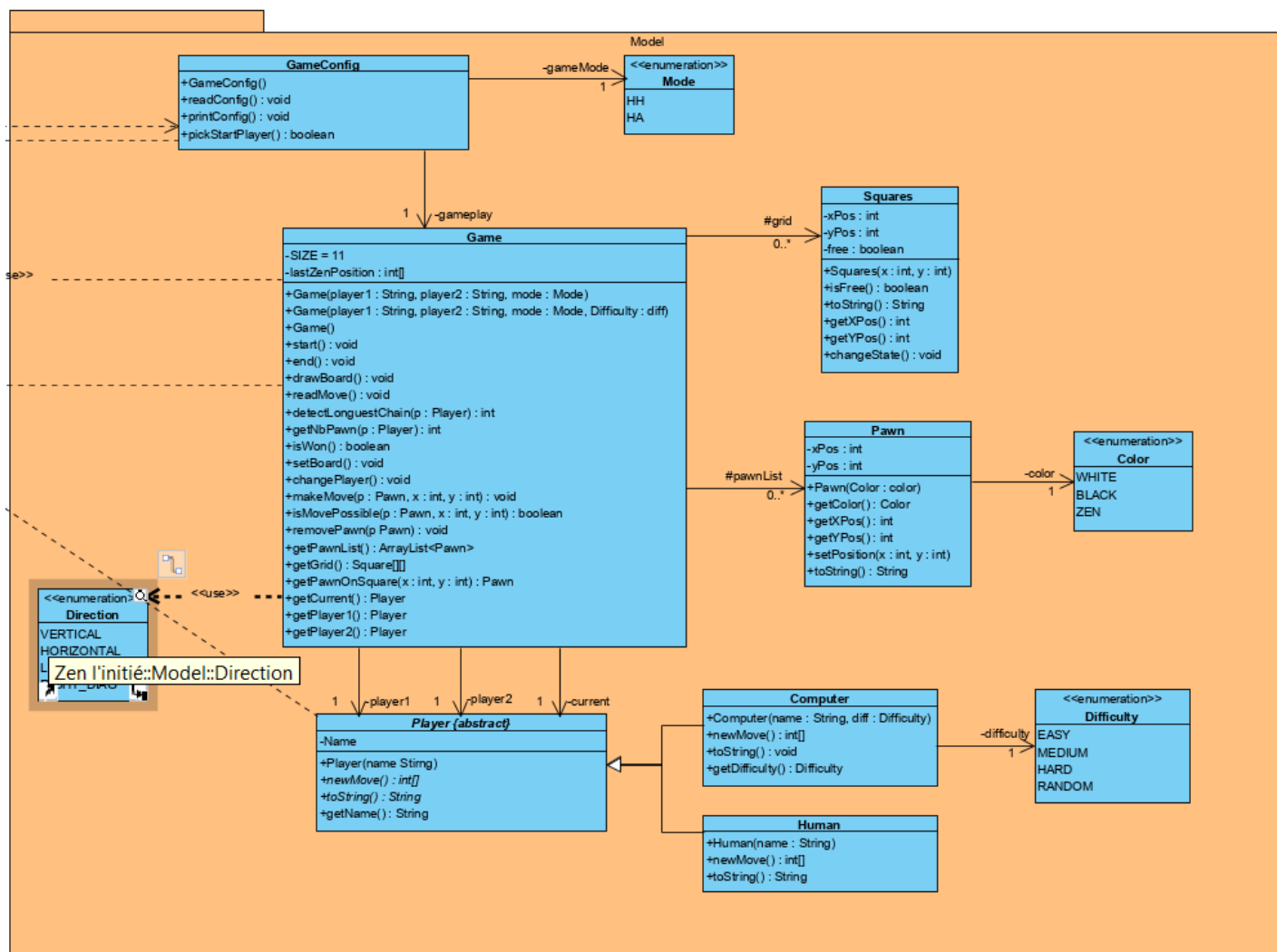


Diagramme de conception complet



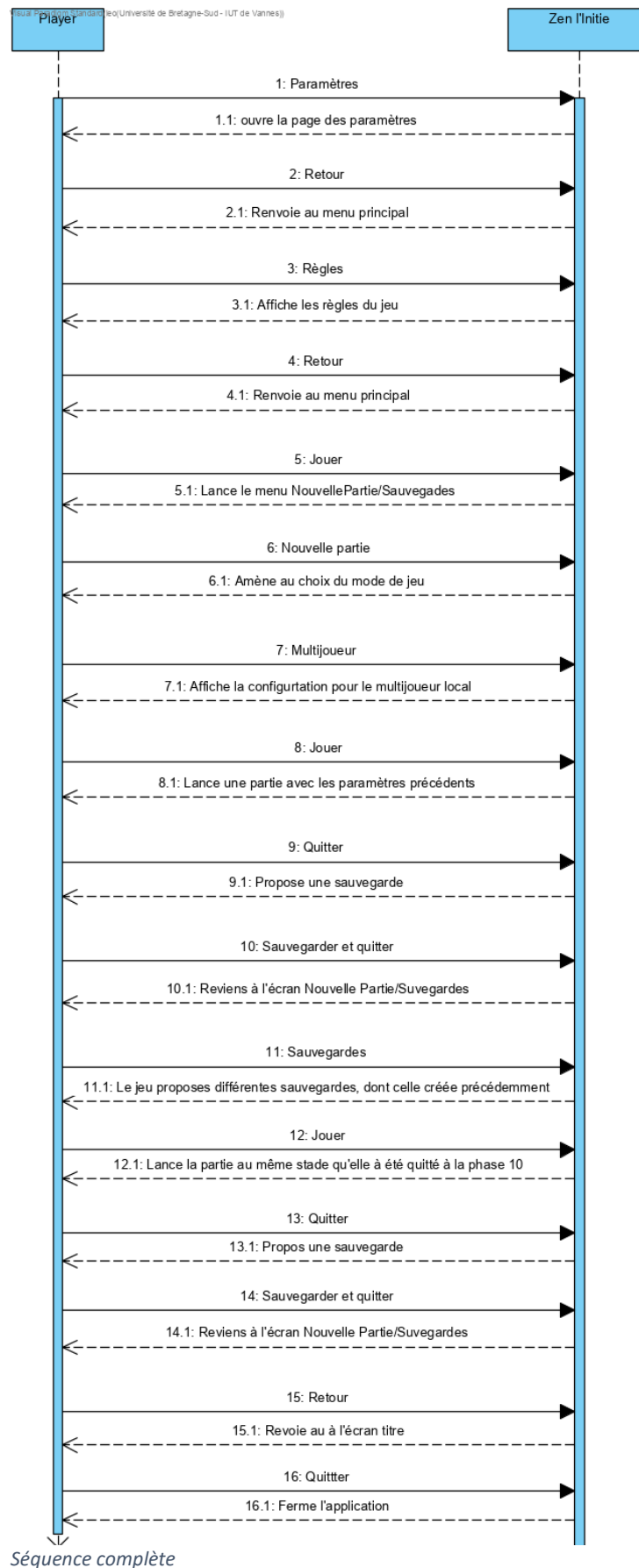
Launcher, Package Control, Package Util

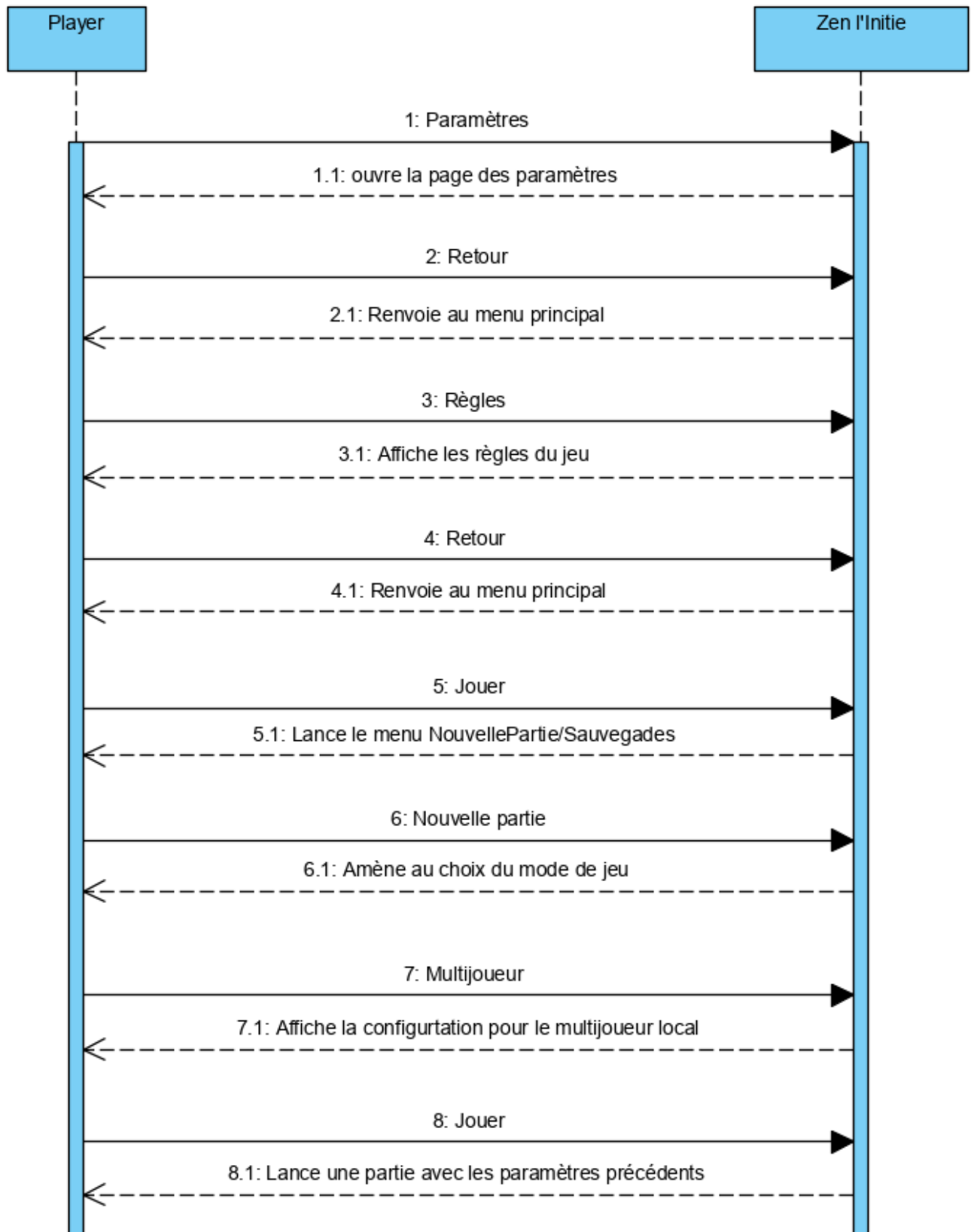


Package Model

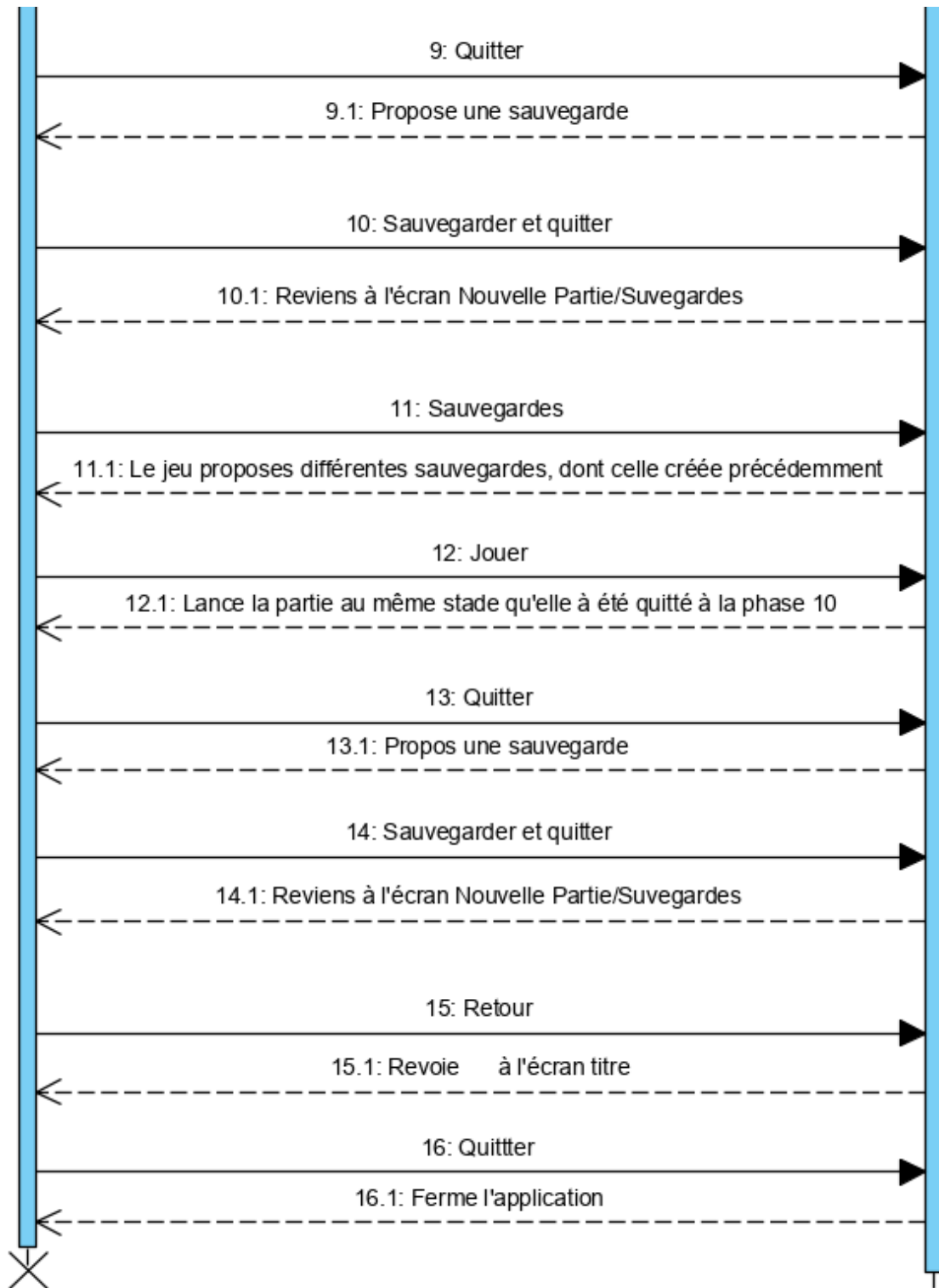
Ce diagramme de classe, dans sa version de conception, permet de comprendre le fonctionnement précis de l'application future. On y voit dessiné tous les attributs, méthodes et dépendances.

3) Diagramme de séquence boîte noir





Séquence partie 1



Séquence partie 2

Ce diagramme de séquence présente un exemple de test effectué en boîte noire. C'est-à-dire que l'utilisateur ne connaît pas le fonctionnement interne de l'application. Il doit donc s'assurer qu'il comprend son utilisation, et qu'aucune fonctionnalité ne présente de bug.

4) Spécification des fichiers

Etant donné la structure complexe des objets à sauvegarder (des instances de la classe Game), on se propose d'utiliser la sérialisation. Ce procédé va nous permettre de sauvegarder l'état de l'objet à un instant T, et de le recharger à l'identique lors d'une autre partie (même après redémarrage de la machine). Il faudra pour cela que la classe Game implémente l'interface Serializable (ainsi que toutes les classes dont Game possède une instance).

On utilisera une classe externe (dans le package util), car il est fortement déconseillé d'utiliser une classe qui peut effectuer une sérialisation d'elle-même. Cette classe possèdera deux méthodes, une méthode save (de sauvegarde) et une méthode load (pour charger une partie).

Lors de la configuration dans gameConfig, il sera demandé au joueur s'il désire créer une nouvelle partie (auquel cas il devra entrer une série de paramètres), ou bien charger une partie sauvegardée (auquel cas on lui demandera de choisir le fichier). Le fichier sera ensuite utilisé pour lire l'objet, et le réinsérer dans le jeu.

5) Squelette du projet

Voici les squelettes des classes.

a) Package model

1. GameConfig

```
1  package model;
2
3
4  /**
5   * Class used to configure a game before it starts
6   *
7   * @author Léo DESMONTS - IUT VANNES - 2020
8   * @version 1.0
9   */
10 public class GameConfig {
11
12
13     /**
14      * The class constructor launches the readConfig sequence,
15      * and creates a new game with the read parameters
16      */
17     public GameConfig() {
18
19     }
20
21
22     /**
23      * readConfig is a prompt sequence, to read the game parameters from the user.
24      */
25     public void readConfig() {
26
27     }
28
29
30     /**
31      * picks randomly the player who starts
32      * Original playing order is the order player names where entered.
33      * This method randomly returns a boolean
34      *
35      * @return -true : order is kept | -false : order is inverted
36      */
37     public boolean pickStartPlayer() {
38         return true;
39     }
40
41
42     /**
43      * printConfig prints the configuration used for the game (gamemode, playernames,
44      * etc)
45      */
46     public void printConfig() {
47
48     }
49 }
```

2. Game

```
1  package model;
2
3  import java.util.ArrayList;
4
5  /**
6   * Main class that handles the game.
7   * Handles the board, the main loop, the end conditions
8   */
9  public class Game {
10
11     private int SIZE = 11;
12     private int[] lastZenPosition;
13     protected Square[][] grid;
14     protected ArrayList<Pawn> pawnList;
15     private Player player1;
16     private Player player2;
17     private Player current;
18
19
20
21     /**
22      * class constructor in case both player are human
23      *
24      * @param player1 name of the first player
25      * @param player2 name of the second player
26      * @param gameMode gameMode (for debugging purposes)
27      */
28     public Game(String player1, String player2, Mode gameMode) {
29
30     }
31
32
33
34     /**
35      * class constructor in case the user is playing against the machine
36      *
37      * @param player1 name of the first player
38      * @param player2 name of the second player
39      * @param gameMode gameMode (for debugging purposes)
40      * @param dif Difficulty of the Automated player
41      */
42     public Game(String player1, String player2, Mode gameMode, Difficulty dif) {
43
44     }
45
46
47
48     /**
49      * class constructor in case the game is loaded
50      */
51     public Game() {
52
53     }
54
55
56
57     /**
58      * method used to start the game
59      */
60     public void start() {
```

```
61
62     }
63
64
65
66     /**
67      * method used to end the game when someone won
68      */
69     public void end() {
70
71     }
72
73
74
75     /**
76      * returns the arrayList of Pawns (for testing purposes)
77      *
78      * @return pawnList attribut
79      */
80     public ArrayList<Pawn> getPawnList() {
81         return new ArrayList<Pawn>();
82     }
83
84
85
86     /**
87      * returns the grid (for testing purposes)
88      *
89      * @return game board
90      */
91     public Square[][] getGrid() {
92         return new Square[0][0];
93     }
94
95
96
97     /**
98      * Gets the pawn on the given square
99      *
100     * @param x x coordinate of the square
101     * @param y y coordinate of the square
102     * @return a pawn, or null if the square is empty
103     */
104     public Pawn getPawnOnSquare(int x, int y) {
105         return new Pawn(Color.ZEN);
106     }
107
108
109
110     /**
111      * Method used to draw th board with the pawns in the current stat in the console
112      */
113     public void drawBoard() {
114
115     }
116
117
118
119     /**
120     * readMove reads the players next move (asks for pawn to move, and for the next
121     * coordinates).
122     * Reapats until move is right, or the player saved the game to quit.
```

```
123     */
124     public void readMove() {
125
126     }
127
128
129
130     /**
131      * Moves a pawn on the grid
132      *
133      * @param p pawn to move
134      * @param x x Coordinate of where to move the pawn
135      * @param y y Coordinate of where to move the pawn
136      */
137     public void makeMove(Pawn p, int x, int y) {
138
139     }
140
141
142
143     /**
144      * Removes a Pawn from pawnList, if this one is taken by the opponent, and is
145      * therefor no longer in game
146      *
147      * @param p Pawn to remove
148      */
149     public void removePawn(Pawn p) {
150
151     }
152
153
154
155     /**
156      * Detects the longest chain of a player. Is used to detect if there is a winner.
157      *
158      * @param p Player to detect the Longest chain of
159      * @return the Longest chain length
160      */
161     public int detectLongestChain(Player p) {
162         return 0;
163     }
164
165
166
167     /**
168      * Gets the number of pawn remaining possessed by the player (counting the ZEN
169      * pawn)
170      * Used to compare to the Longest chain
171      */
172     public int getNbPawn(Player p) {
173         return 0;
174     }
175
176
177
178     /**
179      * isWon is called to verify if the game was won by one of the players
180      * @return true if there is a winner or a tie, else otherwise
181      */
182     public boolean isWon() {
183         return true;
184     }
```

```
185
186
187
188     /**
189     * setBoard is called once by the constructor, to initialize the board and the
190     * pawns
191     */
192     public void setBoard() {
193
194     }
195
196
197
198     /**
199     * changes the current player to the other player
200     */
201     public void changePlayer() {
202
203     }
204
205
206
207     /**
208     * returns the current player (for testing purposes)
209     *
210     * @return the current player
211     */
212     public Player getCurrent() {
213         return new Human("player");
214     }
215
216
217
218     /**
219     * returns the player 1 (for testing purposes)
220     *
221     * @return the player 2
222     */
223     public Player getPlayer1() {
224         return new Human("player");
225     }
226
227
228
229     /**
230     * returns the player 2 (for testing purposes)
231     *
232     * @return the player 2
233     */
234     public Player getPlayer2() {
235         return new Human("player");
236     }
237
238
239
240     /**
241     * Checks if the entered move is possible :
242     * 1) Evaluates the Direction by using relatives positions of the pawn and the
243     * entered move
244     * 2) Checks if there are some enemy pawn on the way
245     *
246     * @param p Pawn to move
```

```
247      * @param x x coordinate to move to
248      * @param y y coordinate to move to
249      * @return true if the move is possible, false otherwise
250      */
251      public boolean isMovePossible(Pawn p, int x, int y) {
252          return true;
253      }
254  }
```


3. Player

```
1  package model;
2
3  /**
4   * abstract class modeling a player, and common attributs/method to human and automated
5   * players.
6   *
7   * @author Léo DESMONTS - IUT VANNES - 2020
8   * @version 1.0
9   */
10 public abstract class Player {
11
12     private String name;
13
14
15
16     /**
17      * creates the player
18      *
19      * @param name the player's name
20      */
21     public Player(String name) {
22
23     }
24
25
26
27     /**
28      * Getter : gets the players name
29      */
30     public String getName() {
31         return "";
32     }
33
34
35
36     /**
37      * Method to be reimplemented by humman and automated players.
38      * Lauches the procedure of a new move
39      */
40     public abstract void newMove();
41
42
43
44     /**
45      * Method to be reimplemented bu human and automated players.
46      *
47      * @return a string containing formatted information about the player
48      */
49     public abstract String toString();
50 }
```

4. Human

```
1  package model;
2
3  /**
4   * class that models a automated player
5   *
6   * @author Léo DESMONTS - IUT VANNES - 2020
7   * @version 1.0
8   */
9  public class Human extends Player {
10
11     /**
12      * class constructor, that calls upon the Player(String) constructor
13      *
14      * @param name name given to the human player
15      */
16     public Human(String name) {
17         super(name);
18     }
19
20
21
22     /**
23      * Allow the player to move a pawn on his turn
24      */
25     public void newMove() {
26
27     }
28
29
30
31     /**
32      * returns a String with formated information about the player
33      * @return formated String
34      */
35     public String toString() {
36         return "";
37     }
38 }
```

5. Computer

```
1  package model;
2
3  /**
4   * class that models a automated player
5   *
6   * @author Léo DESMONTS - IUT VANNES - 2020
7   * @version 1.0
8   */
9  public class Computer extends Player {
10
11      /**
12       * class constructor, that calls upon the Player(String) constructor
13       *
14       * @param name name given to the automated player
15       * @param diff Automated player difficulty
16       */
17      public Computer(String name, Difficulty diff) {
18          super(name);
19      }
20
21
22
23      /**
24       * Allow the automated player to move a pawn on ti's turn
25       */
26      public void newMove() {
27
28      }
29
30
31
32      /**
33       * getter that gets the player difficulty
34       *
35       * @return the difficulty
36       */
37      public Difficulty getDifficulty() {
38          return Difficulty.RANDOM;
39      }
40
41
42
43      /**
44       * returns a String with formatted information about the player
45       * @return formatted String
46       */
47      public String toString() {
48          return "";
49      }
50  }
```

6. Square

```
1 package model;
2
3
4 /**
5  * class that models a square, meant to be part of the playing grid
6  *
7  * @author Léo DESMONTS - IUT VANNES - 2020
8  * @version 1.0
9  */
10 public class Square {
11
12     private int xPos;
13     private int yPos;
14     private boolean free;
15
16
17
18     /**
19      * creates the square, with his position on the grid
20      *
21      * @param x horizontal position
22      * @param y vertical position
23      */
24     public Square(int x, int y) {
25
26     }
27
28
29
30     /**
31      * Checks if the square is free (no pawn is on it)
32      *
33      * @return true if it's free, false otherwise
34      */
35     public boolean isFree() {
36         return true;
37     }
38
39
40
41     /**
42      * Chages the state of the square (free/not free)
43      */
44     public void changeState() {
45
46     }
47
48
49
50     /**
51      * Returns a string with formatted information about the square
52      *
53      * @return the formatted string
54      */
55     public String toString() {
56         return "";
57     }
58
59
60
61     /**
62      * Horizontal poistion on the grid getter
```

```
63      *
64      * @return int with the x position
65      */
66      public int getXPos() {
67          return 0;
68      }
69
70
71
72      /**
73       * Vertical position on the grid getter
74       *
75       * @return in with y position
76       */
77      public int getYPos() {
78          return 0;
79      }
80  }
```

7. Pawn

```
1  package model;
2
3  /**
4   * class that models a pawn
5   *
6   * @author Léo DESMONTS - IUT VANNES - 2020
7   * @version 1.0
8   */
9  public class Pawn {
10
11     private Color color;
12     private int xPos;
13     private int yPos;
14
15
16
17     /**
18      * Class construtor, creates the pawn
19      *
20      * @param color Color of the pawn (can be WHITE, BLACK, ZEN)
21      */
22     public Pawn(Color color) {
23
24     }
25
26
27
28     /**
29      * getter that returns the color of the pawn
30      *
31      * @return a Color
32      */
33     public Color getColor() {
34         return Color.ZEN;
35     }
36
37
38
39     /**
40      * getter that returns the horizontal position of the pawn on the grid
41      *
42      * @return int with the x position
43      */
44     public int getXPos() {
45         return 0;
46     }
47
48
49
50     /**
51      * getter that returns the vertical position of the pawn on the grid
52      *
53      * @return int with the y position
54      */
55     public int getYPos() {
56         return 0;
57     }
58
59
60
61     /**
62      * setter that sets the position of the pawn on the grid
```

```
63      *
64      * @param x horizontal coordinate
65      * @param y vertical coordinate
66      */
67      public void setPosition(int x, int y) {
68
69      }
70
71
72
73      /**
74       * returns a formatted information about the pawn
75       *
76       * @return a string with the information
77       */
78      public String toString() {
79          return "";
80      }
81
82  }
```

S

8. Color

```
1  package model;
2
3  /**
4   * Enumeration
5   * Possible values for pawn color
6   *
7   * @author Léo DESMONTS - INFO VANNES - 2020
8   * @version 1.0
9   */
10 public enum Color {
11     WHITE,
12     BLACK,
13     ZEN
14 }
```

9. Difficulty

```
1  package model;
2
3  /**
4   * Enumeration
5   * Possible difficulty levels for Automated player
6   *
7   * @author Léo DESMONTS - INFO VANNES - 2020
8   * @version 1.0
9   */
10 public enum Difficulty {
11     EASY,
12     MEDIUM,
13     HARD,
14     RANDOM
15 }
```


10. Direction

```
1  package model;
2
3  /**
4   * Enumeration
5   * Possible directions :
6   * VERTICAL
7   * HORIZONTAL
8   * LEFT_DIAG - diagonal heading left over the horizontal line passing through the
9   * tested pawn
10  * RIGHT_DIAG - diagonal heading right over the horizontal line passing through the
11  * tested pawn
12  *
13  * @author Léo DESMONTS - INFO VANNES - 2020
14  * @version 1.0
15  */
16  public enum Direction {
17      VERTICAL,
18      HORIZONTAL,
19      LEFT_DIAG,
20      RIGHT_DIAG
21  }
```

11. Mode

```
1  package model;
2
3  /**
4   * Enumeration
5   * Possible game modes :
6   * H - Human
7   * A - Automated
8   *
9   * @author Léo DESMONTS - INFO VANNES - 2020
10  * @version 1.0
11  */
12  public enum Mode {
13      HH,
14      HA
15  }
```

b) Package control

1. Prompt

```
1 package control;
2
3 import model.Mode;
4
5 /**
6  * Class that offers different prompt types : mode prompt, name prompt, coordinates
7  * prompt and quit/save prompt.
8  * Every method is only designed for the value recovery. Every display fonctionnalité
9  * (visual communication with the user) is handled by the calling method.
10 * This way, the methods are more likely to be called by various methods.
11 *
12 * @author Léo DESMONTS - IUT VANNES - 2020
13 * @version 1.0
14 */
15 public class Prompt {
16
17     /**
18      * inputMode asks the user for an input, to select the gameMode :
19      *     - HH (Human - Human)
20      *     - HA (Human - Automated)
21      *
22      * @return Mode gameMode chosen by the user
23      */
24     public static Mode inputMode() {
25         return Mode.HH;
26     }
27
28
29
30     /**
31      * inputName asks the user for an input, to select a name for a player.
32      *
33      * @return String name of the player
34      */
35     public static String inputName() {
36         return "";
37     }
38
39
40
41     /**
42      * inputCoordinates asks the user for two Int, to form a coordinate (for a pawn to
43      * select of a move to make).
44      *
45      * @return int[] a tab of int of length 2
46      */
47     public static int[] inputCoordinates() {
48         int[] ret = {0,0};
49         return ret;
50     }
51
52
53
54     /**
55      * askForQuit asks the user for y/n to maybe launch the save/quit procedure
56      *
57      * @return a char with the user's answer
58      */
59     public static char askForQuit() {
60         return 'y';
61     }
62 }
```

c) Package util

1. Save

```
1 package util;
2
3 import model.Game;
4
5 /**
6  * This class offers two static method, to save and load a game to/from a file
7  * (Serialized).
8  *
9  * @author Léo DESMONTS - IUT VANNES - 2020
10 * @version 1.0
11 */
12 public class Save {
13
14
15
16     /**
17      * writeSave allows you to save an ongoing game, to continue later
18      *
19      * @param fileName name of the save. The path is fix, and defined in the method's
20      * code.
21      * @param game Game object to save
22      */
23     public static void writeSave (String fileName, Game game) {
24
25     }
26
27
28
29     /**
30      * readSave allows you to read a saved game to load and play.
31      *
32      * @param fileName name of the save.
33      * @return a Game object that will be loaded by the app.
34      */
35     public static Game readSave(String fileName) {
36         return new Game();
37     }
38
39 }
```

Les classes présentées ci-dessus sont les classes qui seront complété durant la phase de codage.

d) Launcher

```
1  import model.GameConfig;
2
3  /**
4   * Laucher class of the ZenApp
5   *
6   * @author Léo DESMONTS - IUT Vannes - 2020
7   * @version 1.0
8   */
9  public class Launcher {
10
11      /**
12       * Lauches the game
13       *
14       * @param args String[] but isn't needed here.
15       */
16      public static void main (String[] args) {
17
18      }
19  }
```

6) Test unitaires (JUnit)

Toute méthode, toute classe doit être testée. On suit ici un modèle de TDD (Test Driven development), développement piloté par les tests en français ; on écrit les tests avant de commencer la phase de code, afin de voir au fur et à mesure si nos méthodes passent tous les tests auxquels on les soumet. Bien sur il est difficile d'être exhaustif, il est donc tout à fait possible de rajouter des test durant/après la phase de développement.

On tente, dans la mesure du possible de faire une classe de test, par classe de l'application. Les enumerations, ainsi que les méthodes demandant une entrée de donnée de la part de l'utilisateur ne seront pas testées.

a) Test du package model

1. TestGameConfig

```
1  package test.model;
2
3  import org.junit.*;
4  import static org.junit.Assert.*;
5
6  import model.GameConfig;
7
8  public class TestGameConfig {
9
10     GameConfig g;
11
12
13
14     /**
15      * setUp
16      * Creates the test object before every test
17      */
18     @Before()
19     public void setUp() {
20         g = new GameConfig();
21     }
22
23     /**
24      * tearDown
25      * Removes every link to objects (for the GC to do it's work)
26      */
27     @After()
28     public void tearDown() {
29         g = null;
30     }
31
32
33
34     /**
35      * Tests if the pickStartPlayer works
36      * Doing 4 sepreate tests, to have little chance of getting 4 times the same result
37      */
38     @Test()
39     public void testPickStartPlayer() {
40         Boolean b1 = g.pickStartPlayer();
41         Boolean b2 = g.pickStartPlayer();
42         Boolean b3 = g.pickStartPlayer();
43         Boolean b4 = g.pickStartPlayer();
44         if (b1 == b2 && b2 == b3 && b3 == b4) {
45             assertEquals(true, true);
46         }
47         else {
48             assertEquals(true, false);
49         }
50     }
51
52 }
```

2. TestGame

```
1  package test.model;
2
3  import org.junit.*;
4  import static org.junit.Assert.*;
5
6  import java.util.ArrayList;
7
8  import model.Difficulty;
9  import model.Mode;
10 import model.Game;
11 import model.Square;
12 import model.Pawn;
13 import model.Player;
14
15 public class TestGame {
16
17     Game g;
18
19     /**
20      * setUp
21      * Creates the test object before every test
22      */
23     @Before()
24     public void setUp() {
25         g = new Game ("George", "Fred", Mode.HH);
26     }
27
28     /**
29      * tearDown
30      * Removes every link to objects (for the GC to do it's work)
31      */
32     @After()
33     public void tearDown() {
34         g = null;
35     }
36
37
38
39     /**
40      * Tests if games exists
41      */
42     @Test()
43     public void testExists() {
44         assertNotNull(g);
45         Game g2 = new Game("George", "Bot", Mode.HA, Difficulty.EASY);
46         assertNotNull(g2);
47         g2 = null;
48     }
49
50
51     /**
52      * Test grid size
53      */
54     @Test()
55     public void testSizeGrid() {
56         int x = g.getGrid()[0].length;
57         int y = g.getGrid().length;
58         assertEquals(11, x);
59         assertEquals(11, y);
60     }
61
62 }
```

```
63
64  /**
65   * Checks if the method getPawnOnSquare works
66   */
67  @Test()
68  public void testGetPawnOnSquare() {
69      Pawn p1 = g.getPawnList().get(0);
70      p1.setPosition(1, 1);
71
72      Pawn p2 = g.getPawnOnSquare(1, 1);
73
74      assertEquals(p1, p2);
75  }
76
77
78
79
80  /**
81   * Testing the removal of a pawn from the grid
82   */
83  @Test()
84  public void testRemovePawn(){
85
86      Square s = g.getGrid()[5][0];
87      ArrayList<Pawn> list = g.getPawnList();
88      Pawn p = g.getPawnOnSquare(0, 5);
89
90      assertEquals(false, s.isFree());
91      assertEquals(true, list.contains(p));
92
93      g.removePawn(p);
94
95      assertEquals(true, s.isFree());
96      assertEquals(false, list.contains(p));
97  }
98
99
100
101  /**
102   * Tests is the player is able ot move a pawn
103   */
104  public void testMakeMove() {
105
106      Pawn p = g.getPawnOnSquare(0, 5);
107
108      assertEquals(false, g.getGrid()[5][0].isFree());
109      assertEquals(true, g.getGrid()[5][3].isFree());
110      assertEquals(0, p.getXPos());
111      assertEquals(5, p.getYPos());
112
113      g.makeMove(p, 3, 5);
114
115      assertEquals(true, g.getGrid()[5][0].isFree());
116      assertEquals(false, g.getGrid()[5][3].isFree());
117      assertEquals(3, p.getXPos());
118      assertEquals(5, p.getYPos());
119  }
120
121
122
123  /**
124   * Tests if the change of current player works
```

```
125     */
126     @Test()
127     public void testChangePlayer() {
128         Player p1 = g.getPlayer1();
129         Player c = g.getCurrent();
130
131         assertEquals(p1, c);
132
133         Player p2 = g.getPlayer2();
134         g.changePlayer();
135
136         assertEquals(p2, c);
137     }
138
139
140
141     /**
142      * Tests if the isMovePossible works
143      */
144     @Test()
145     public void testIsMovePossible() {
146         Pawn p = g.getPawnOnSquare(0, 5);
147
148         assertEquals(p, g.isMovePossible(p, -3, 5));
149         assertEquals(true, g.isMovePossible(p, 3, 5));
150
151         g.makeMove(p, 3, 5);
152
153         assertEquals(false, g.isMovePossible(p, 6, 5));
154         assertEquals(false, g.isMovePossible(p, 2, 5));
155
156     }
157
158
159     /**
160      * Test if the Longest chain is detected
161      */
162     public void testLongestChain() {
163         assertEquals(1, g.detectLongestChain(g.getPlayer1()));
164         assertEquals(1, g.detectLongestChain(g.getPlayer2()));
165
166         g.makeMove(g.getPawnOnSquare(2, 3), 0, 3);
167         g.makeMove(g.getPawnOnSquare(0, 0), 0, 4);
168         g.makeMove(g.getPawnOnSquare(10, 6), 10, 2);
169         g.makeMove(g.getPawnOnSquare(5, 5), 3, 5);
170         g.makeMove(g.getPawnOnSquare(3, 5), 1, 5);
171
172         assertEquals(4, g.detectLongestChain(g.getPlayer1()));
173         assertEquals(3, g.detectLongestChain(g.getPlayer2()));
174     }
175
176
177     /**
178      * Tests if the number of pawns is correctly detected
179      */
180     public void testNbPawn() {
181         assertEquals(13, g.getNbPawn(g.getPlayer1()));
182         assertEquals(13, g.getNbPawn(g.getPlayer2()));
183
184         g.removePawn(g.getPawnOnSquare(0, 5));
185
186         assertEquals(12, g.getNbPawn(g.getPlayer1()));
```



```
187         assertEquals(13, g.getNbPawn(g.getPlayer2()));  
188     }  
189 }
```

3. TestHuman

```
1  package test.model;
2
3  import org.junit.*;
4  import static org.junit.Assert.*;
5
6  import model.Human;
7
8  public class TestHuman {
9
10     private Human h;
11
12     /**
13      * setUp
14      * Creates the test object before every test
15      */
16     @Before()
17     public void setUp() {
18         h = new Human("Player1");
19     }
20
21     /**
22      * tearDown
23      * Removes every link to objects (for the GC to do it's work)
24      */
25     @After()
26     public void tearDown() {
27         h = null;
28     }
29
30
31
32     /**
33      * Tests if human exists
34      */
35     @Test()
36     public void testExists() {
37         assertNotNull(h);
38     }
39
40
41
42     /**
43      * Tests if name getter works
44      */
45     @Test()
46     public void testGetName() {
47         assertEquals("Player1", h.getName());
48     }
49
50
51
52     /**
53      * Tests if the toString method works
54      */
55     @Test()
56     public void testToString() {
57         String expected = "Human\nName = Player1";
58         String test = h.toString();
59         assertEquals(expected, test);
60     }
61
62 }
```

4. TestComputer

```
1  package test.model;
2
3  import org.junit.*;
4  import static org.junit.Assert.*;
5
6  import model.Computer;
7  import model.Difficulty;
8
9  public class TestComputer {
10
11     private Computer c;
12
13     /**
14      * setUp
15      * Creates the test object before every test
16      */
17     @Before()
18     public void setUp() {
19         c = new Computer("Bot1", Difficulty.EASY);
20     }
21
22     /**
23      * tearDown
24      * Removes every link to objects (for the GC to do it's work)
25      */
26     @After()
27     public void tearDown() {
28         c = null;
29     }
30
31
32
33     /**
34      * Tests if computer exists
35      */
36     @Test()
37     public void testExists() {
38         assertNotNull(c);
39     }
40
41
42
43     /**
44      * Tests if name getter works
45      */
46     @Test()
47     public void testGetName() {
48         assertEquals("Bot1", c.getName());
49     }
50
51
52
53     /**
54      * Tests if difficulty getter works
55      */
56     @Test()
57     public void testGetDifficulty() {
58         assertEquals(Difficulty.EASY, c.getDifficulty());
59     }
60
61
62
```

```
63     /**
64      * Tests if the toString method works
65      */
66     @Test()
67     public void testToString() {
68         String expected = "Computer\nName = Bot1";
69         String test = c.toString();
70         assertEquals(expected, test);
71     }
72
73 }
```

5. TestSquare

```
1  package test.model;
2
3  import org.junit.*;
4  import static org.junit.Assert.*;
5
6  import model.Square;
7
8  public class TestSquare {
9
10     private Square s;
11
12
13     /**
14      * setUp
15      * Creates the test object before every test
16      */
17     @Before()
18     public void setUp() {
19         s = new Square(2,5);
20     }
21
22     /**
23      * tearDown
24      * Removes every link to objects (for the GC to do it's work)
25      */
26     @After()
27     public void tearDown() {
28         s = null;
29     }
30
31
32
33     /**
34      * tests if the square exists
35      */
36     @Test()
37     public void testExists() {
38         assertNotNull(s);
39     }
40
41     /**
42      * Tests if coordinate getters work
43      */
44     @Test()
45     public void testCoordinateGetters() {
46         assertEquals(2,s.getXPos());
47         assertEquals(5,s.getYPos());
48     }
49
50     /**
51      * Tests if isFree returns the right value
52      */
53     @Test()
54     public void testIsFree() {
55         assertEquals(false,s.isFree());
56     }
57
58     /**
59      * Tests if the change of state works (and tests isFree some more)
60      */
61     @Test()
62     public void testChangeState() {
```

```
63     assertEquals(false,s.isFree());
64     s.changeState();
65     assertEquals(true,s.isFree());
66     s.changeState();
67     assertEquals(false,s.isFree());
68 }
69
70 /**
71  * Tests if the toString methods works
72  */
73 @Test()
74 public void testToString() {
75     String test = s.toString();
76     String expected = "X = 2\nY = 5\nFree = false";
77     assertNotNull(test);
78     assertEquals(expected,test);
79 }
80 }
```

6. TestPawn

```
1  package test.model;
2
3  import org.junit.*;
4  import static org.junit.Assert.*;
5
6  import model.Pawn;
7  import model.Color;
8
9  public class TestPawn {
10
11     private Pawn pW;
12     private Pawn pB;
13     private Pawn pZ;
14
15     /**
16      * setUp
17      * Creates the test object before every test
18      */
19     @Before()
20     public void setUp() {
21         pW = new Pawn(Color.WHITE);
22         pB = new Pawn(Color.BLACK);
23         pZ = new Pawn(Color.ZEN);
24     }
25
26
27
28     /**
29      * tearDown
30      * Removes every link to objects (for the GC to do it's work)
31      */
32     @After()
33     public void tearDown() {
34         pW = null;
35         pB = null;
36         pZ = null;
37     }
38
39
40
41     /**
42      * tests if the pawns exists
43      */
44     @Test()
45     public void testExists() {
46         assertNotNull(pW);
47         assertNotNull(pB);
48         assertNotNull(pZ);
49     }
50
51
52
53     /**
54      * Tests if the color getter works
55      */
56     @Test()
57     public void testGetColor() {
58         assertEquals(Color.WHITE, pW.getColor());
59         assertEquals(Color.BLACK, pB.getColor());
60         assertEquals(Color.ZEN, pZ.getColor());
61     }
62 }
```

```
63
64
65     /**
66      * Tests if the Position Setters/Getters work
67      */
68     @Test()
69     public void testPostionGettersSetter() {
70         assertEquals(0, pW.getXPos());
71         assertEquals(0, pW.getYPos());
72         pW.setPosition(3, 5);
73         assertEquals(3, pW.getXPos());
74         assertEquals(5, pW.getYPos());
75     }
76
77
78
79     /**
80      * Tests if the toString method works
81      */
82     @Test
83     public void testToString() {
84         pW.setPosition(3, 5);
85         String test = pW.toString();
86         String expected = "Color = WHITE\nx = 3\ny = 5";
87         assertEquals(expected, test);
88     }
89 }
```


b) Test du package util

1. TestSave

```
1 package test.util;
2
3 import org.junit.*;
4 import static org.junit.Assert.*;
5
6 import model.Game;
7 import util.Save;
8 import model.Mode;
9
10 public class TestSave {
11
12     /**
13      * Tests if a saved game loads the same way
14      */
15     @Test()
16     public void testSaveLoad() {
17         Game g1 = new Game("Player1", "Player1", Mode.HH);
18         Save.writeSave("./saves/game1", g1);
19         Game g2 = Save.readSave("./saves/games");
20         assertEquals(g1, g2);
21     }
22
23 }
```

7) ANT

Il est possible grâce à l'outil ANT, d'automatiser une partie des tâches liées au développement. Le fichier build.xml permet de décrire à ANT les actions à effectuer. Il fait, dans l'ordre :

- De nettoyer les dossiers du projet (qui seront régénérés après)
- De compiler les classes de l'application
- A générer un exécutable (.jar)
- A générer la JavaDoc
- A compiler les classes de tests
- A lancer les tests, et produire un rapport de ces derniers

```
1  <project name="Zen" default="test" basedir=".>
2
3      <description> Compiles, generates javaDoc, and runs JUnit tests </description>
4
5      <property name="main.build.dir" location="../build/main/" />
6      <property name="main.src.dir" location="../src/" />
7      <property name="test.build.dir" location="../build/test/" />
8      <property name="test.src.dir" location="../src/test/" />
9      <property name="dist" location=".." />
10     <property name="javadoc" location="../javaDoc" />
11     <property name="testReport" location="../testReport" />
12
13
14     <path id="classpath.test">
15         <pathelement location="${main.src.dir}lib/junit-4.13.jar" />
16         <pathelement location="${main.src.dir}lib/hamcrest-core-1.3.jar" />
17         <pathelement location="${main.build.dir}" />
18     </path>
19
20
21     <target name="clean" description="Cleans the directories the build.xml works with" >
22         <delete dir="${main.build.dir}" />
23         <delete dir="${test.build.dir}" />
24         <delete dir="${javadoc}" />
25         <delete dir="${testReport}" />
26         <delete file="{dist}/ZenInitie.jar" />
27     </target>
28
29
30     <target name="compile" depends="clean" description="Compiles the project">
31         <mkdir dir="${main.build.dir}" />
32         <javac destdir="${main.build.dir}" includeantruntime="false">
33             <src path="${main.src.dir}" />
34             <include name="controle/*.java" />
35             <include name="model/*.java" />
36             <include name="util/*.java" />
37             <include name="*.java" />
38         </javac>
39     </target>
```

```

42     <target name="dist" depends="compile" description="creates a .jar file of the project">
43         <mkdir dir="${dist}"/>
44         <jar jarfile="${dist}/ZenLInitie.jar" basedir="${main.build.dir}">
45             <manifest>
46                 <attribute name="Main-Class" value="Laucher" />
47             </manifest>
48         </jar>
49     </target>
50
51
52     <target name="javadoc" depends="dist" description="Generates the JavaDoc">
53         <mkdir dir="${javadoc}"/>
54         <javadoc sourcepath="${main.src.dir}" destdir="${javadoc}" />
55     </target>
56
57
58     <target name="test-compile" depends="javadoc">
59         <mkdir dir="${test.build.dir}"/>
60         <javac srcdir="${test.src.dir}" destdir="${test.build.dir}" includeantruntime="true">
61             <classpath refid="classpath.test"/>
62         </javac>
63     </target>
64
65
66     <target name="test" depends="test-compile">
67         <junit printsummary="on" haltonfailure="no" fork="true">
68             <classpath>
69                 <path refid="classpath.test"/>
70                 <pathelement location="${test.build.dir}"/>
71             </classpath>
72             <formatter type="xml" />
73
74             <test name="test.model.PileTest"/>
75             <test name="test.model.TestComputer"/>
76             <test name="test.model.TestGame"/>
77             <test name="test.model.TestGameConfig"/>
78             <test name="test.model.TestHuman"/>
79             <test name="test.model.testPawn"/>
80             <test name="test.model.TestSquare"/>
81             <test name="test.util.TestSave"/>
82
83         </junit>
84         <mkdir dir="${testReport}"/>
85         <junitreport todir="${testReport}">
86             <fileset dir=".">
87                 <include name="TEST-*.xml"/>
88             </fileset>
89             <report format="frames" todir="${testReport}"/>
90         </junitreport>
91         <delete file="TEST-test.model.PileTest.xml"/>
92         <delete file="TEST-test.model.TestComputer.xml"/>
93         <delete file="TEST-test.model.TestGame.xml"/>
94         <delete file="TEST-test.model.TestGameConfig.xml"/>
95         <delete file="TEST-test.model.TestHuman.xml"/>
96         <delete file="TEST-test.model.TestPawn.xml"/>
97         <delete file="TEST-test.model.TestSquare.xml"/>
98         <delete file="TEST-test.util.TestSave.xml"/>
99     </target>
100 </project>

```