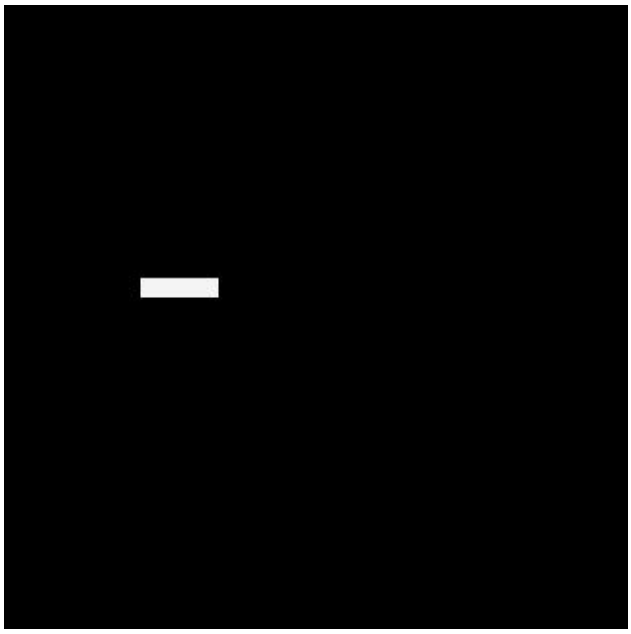


Snake

Software Modeling and Design

To Xavier Ferré

At Tongji University, Shanghai

1. Description of the document

In this document, we will talk about our project *The Snake*. We will describe the structure of our project with different diagram: class and sequence diagram.

Title	[2017][SMD] Group 6 - Snake
Date	09/11/2015
Authors	Arnaud Clément Xian ZHENG Stéphane KHAU Gabriel TANG
Leader	Arnaud Clément Xian ZHENG
E-Mail	arnaud.zg@gmail.com
Subject	Snake
Keyword	software, modeling, design, pattern, tongji
Versioning	2.0

2. Table revisions

Date	Version	Description	Section	Author
09/11/2015	2.0	Add diagram and editing some parts	All	Arnaud ZHENG
08/11/2015	1.5	Edit and correct some part	All	Gabriel TANG
07/11/2015	1.0	Add text about project presentation and others	All	Stéphane KHAU

3. Table of contents

4. The Snake Game

4.1. History

The Snake is an old game, the concept originated arcade game “Blockade” developed and published by Gremlin in 1976 and then there is some variant game made by the phone constructor Nokia. The first known personal computer version of Snake, titled Worm, was programmed in 1978 by *Peter Trefonas*.

This game is composed of a few elements:

- A map which has wall for delimit the game action.
- A Snake who is still running into the map.
- A food which is generated automatically into the map.

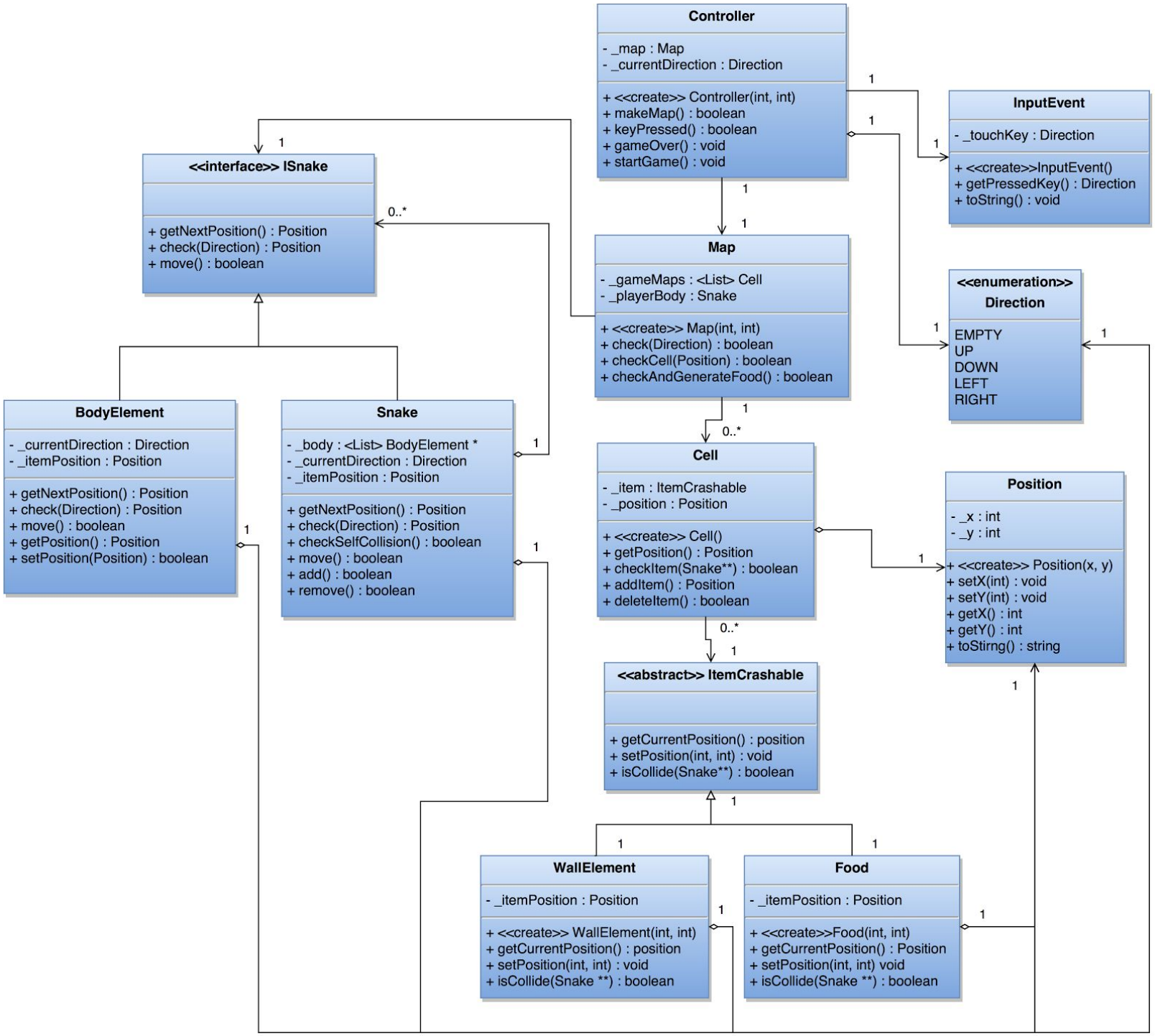
4.2. Rules

- Don't run the snake into the wall, or his own tail: you die.
- Use your cursor keys: up, left, right, and down.

---

5. Description of our design

5.1. Class diagram



Class diagram

5.1.1. Explanations

We just designed a little part of our game, in a loop when the user pressed a key the method **keyPressed()** is called. We used the object **InputEvent** to identify which key is pressed and with that class we can get the direction with the enumeration **Direction**. This class can just build maps, start a game and make a loop or stop the game by the **gameOver()** method.

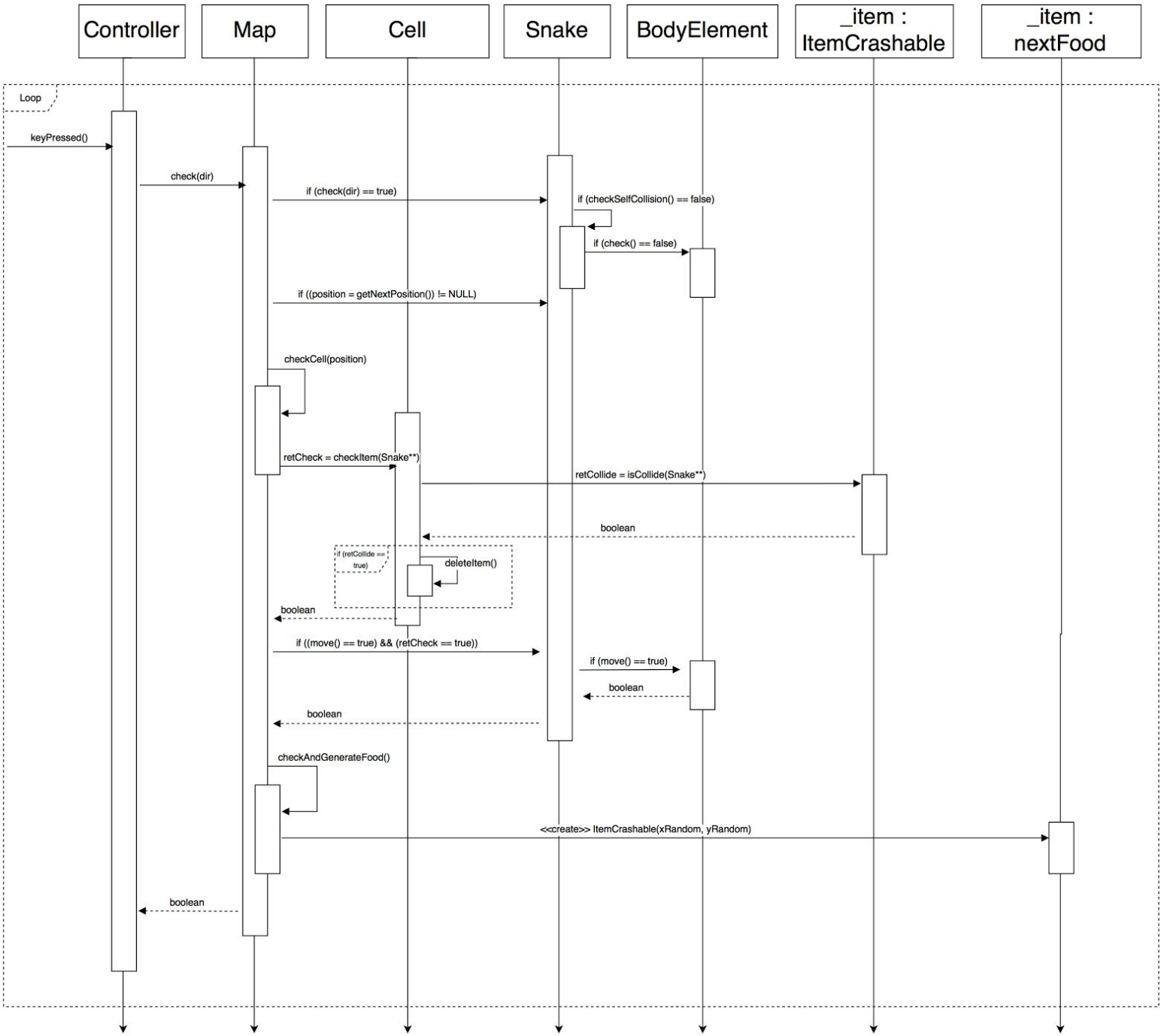
We have only one controller and in this class, it contains an instance of a map, this map will contain every component of our game.

First of all, the map will contain a list of cells, they all have a position x and y value and an item. This item can be a **WallElement**, **Food** or **null**. Every item inherits from **ItemCrashable**. It's very important to know that each cell know if he has an item or not, but it can't tell which type of item it is.

Then map contains an instance of the **Snake** which inherit from **ISnake**, in this class it contains every element of the snake it means a list of **BodyElement**.

5.2. Sequences diagram

5.2.1. Global diagram



Global sequence diagram

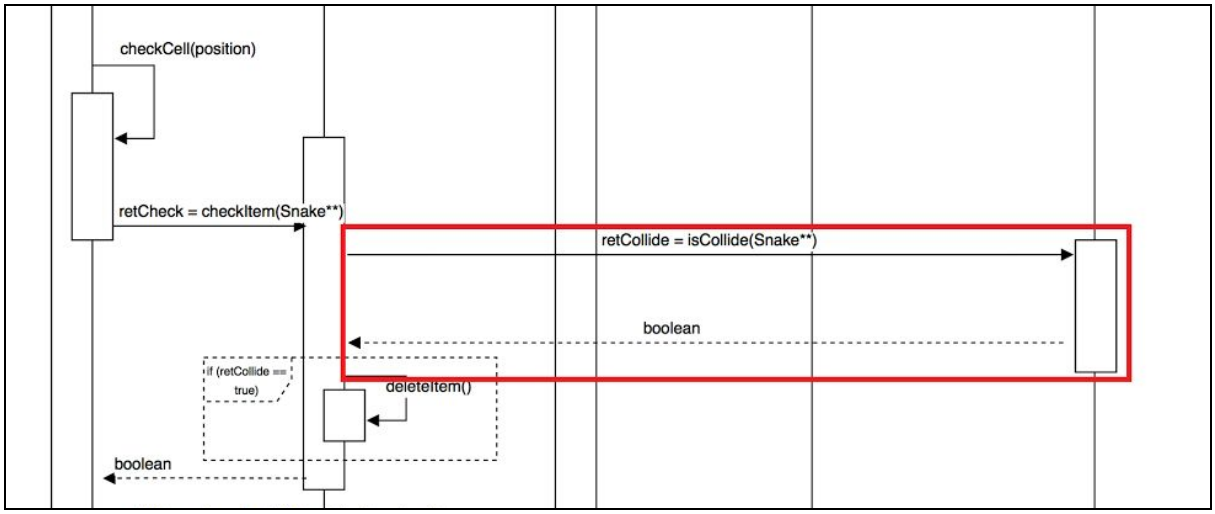
5.2.1.1. Explanations

When the game is initialized, we will make a loop and wait an input event from the user. At every input event we will call the method **keyPressed()**.

Our controller will call the method **check(dir)** and tell to the map that the user have pressed a key. After that, map will first tell to his snake attribute that he has to check also in that direction with the method **check(dir)**. The snake will check if there is some self collision with all the part of his body.

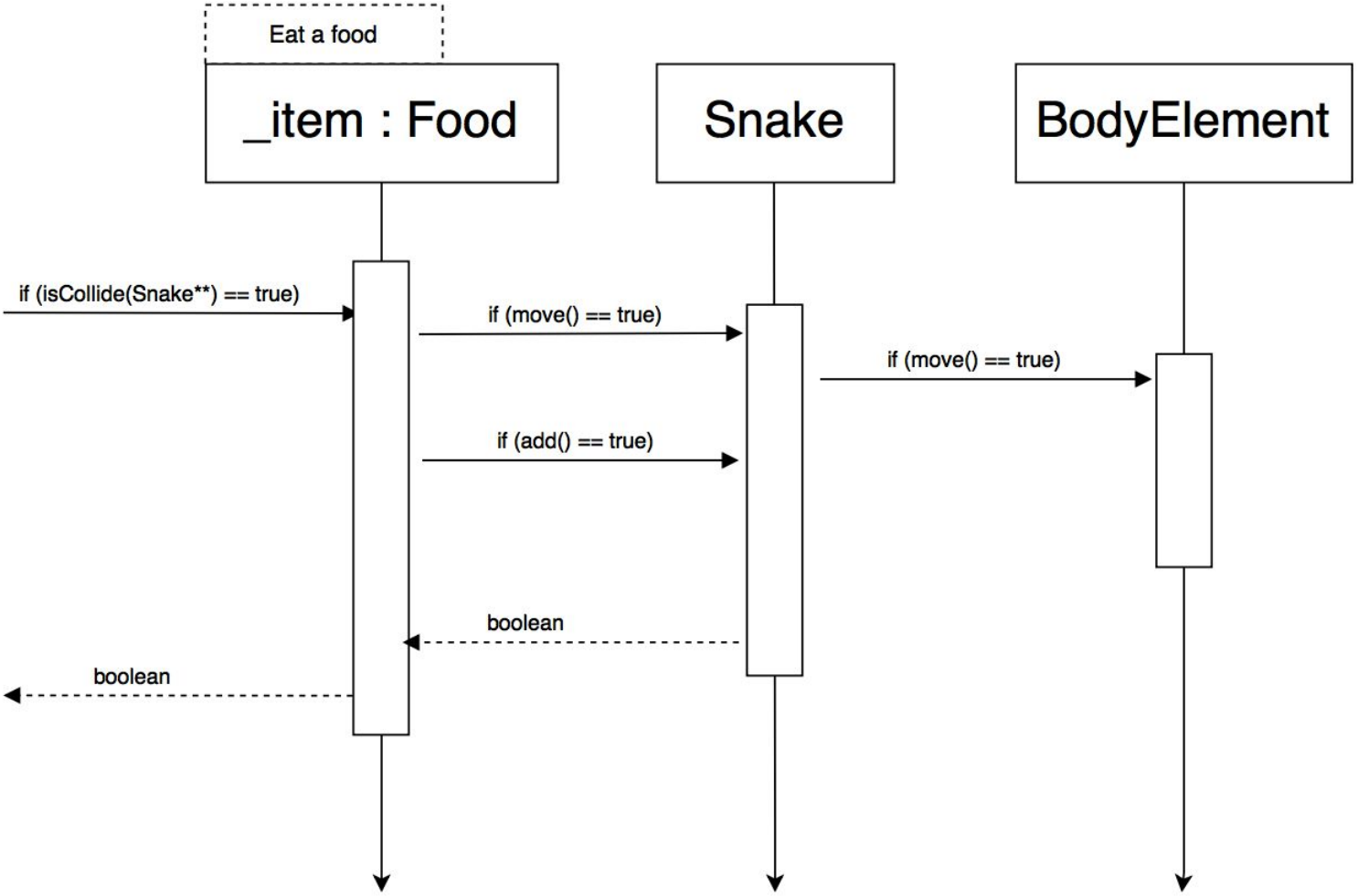
Then map will check if there is some collision with other components of our game. That's why we will ask what's the next position of our snake with the method **getNextPosition()**, our map will know the position to be checked and will find in his list the right cell, and will call the method **checkItem(Snake**)** and that cell will check if there is some collision with the method **isCollide()**. If there is some collision we will just delete that item.

Finally, if there is no collision map will just tell to our snake to make some move, and at the end of our loop map will self called the method **checkAndGenerateFood()**, that will check if there is any food on the map. If there is no food it will generate a food on a random position.



Now we will show you some details about the red square in the image just above.

5.2.2. Case 1 : Eat some food



Sequence diagram - Eat a food

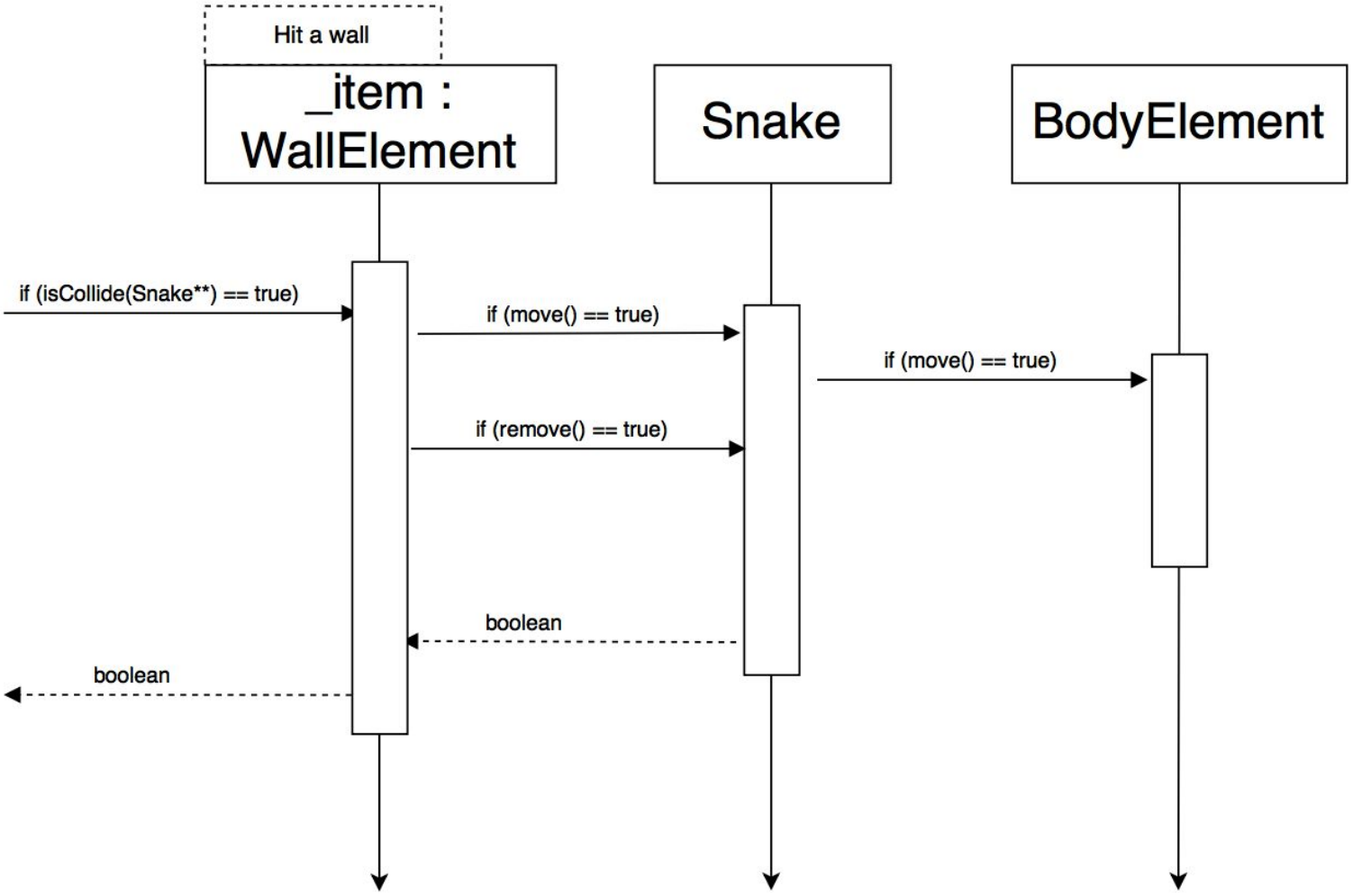
5.2.2.1. Explanations

In this case a **Cell** of the map calls the method **isCollide(Snake**)** on its item. The goal of this function is to check if there is some collision with the snake. Every item inherits from **ItemCrashable**, and if it's a **WallElement** it will tell to the snake to make some movement by calling the method **move()**. That's will move every part of the snake, that's composed by many **BodyElement**.

Then the food will know there is some collision between snake and food, it will call the method **add()** of snake, That method will add an instance of **BodyElement** at the end of the snake.

Then now you will find below the hit wall sequence diagram.

5.2.3. Case 2 : Hit a wall



Sequence diagram - Hit a wall

5.2.3.1. Explanations

In this case a **Cell** of the map calls the method **isCollide(Snake**)** on its item. The goal of this function is to check if there is some collision with the snake. Every item inherits from **ItemCrashable**, and if it's a **WallElement** it will tell to the snake to make some movement by calling the method **move()**. That's will move every part of the snake, that's composed by many **BodyElement**.

After that, seen as item know something is colliding with him, **WallElement** knows what snake should do. So in that case, it's self-killing, that's why our item will call **delete()** method, it will delete each element of the snake. On every loop, before calling **keyPressed()** method, we will check if the snake is still alive by getting his length.

---

6. Design pattern

6.1. Gang of Four

The Gang of Four is the four authors of the book, *Design Patterns: Elements of Reusable Object-Oriented Software*, this is the base of the 23 GoF design pattern.

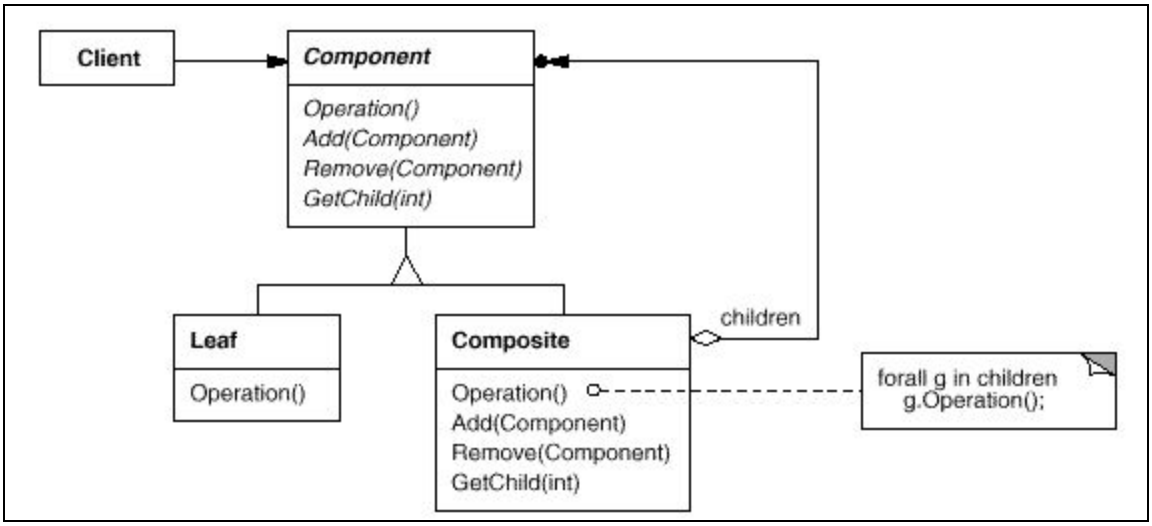
6.2. Design pattern used

In this part, we will talk about the design pattern that we used and designed pattern that we could use, we will explain about how they work and why it fits with our snake game.

First, we will talk about the composite pattern.

Composite Pattern

The composite allow us to represent our objects in a tree structure and can treat object or compositions of objects uniformly. In our case we wanted to represent the part-whole hierarchies of our objects. Composite lets clients treat individual objects and compositions of objects uniformly. We need to manipulate a hierarchical collection collection of composite objects.



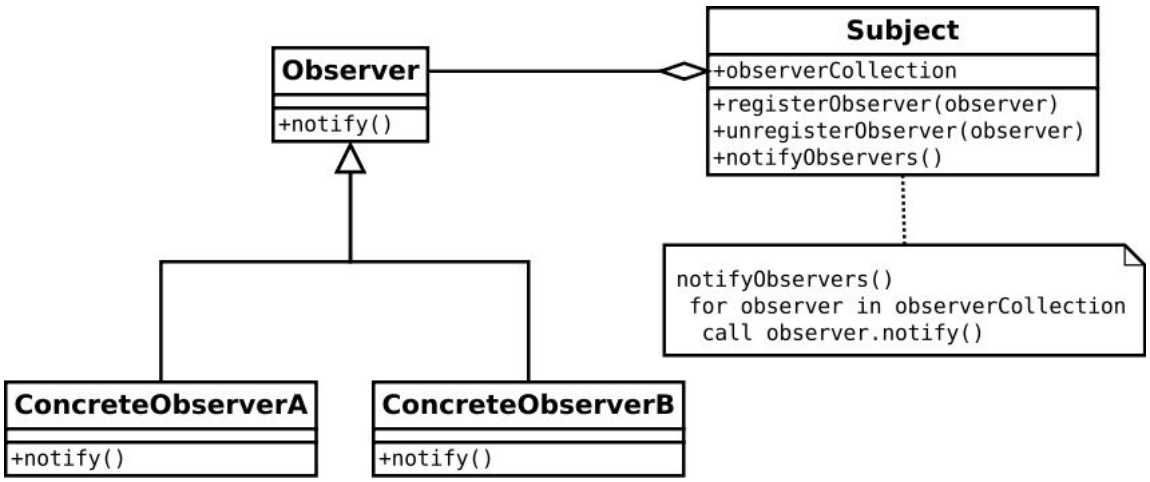
Let's have a little review of our class diagram on *page 5*, in our case composite design pattern is applied to the snake. We have the component **ISnake**, the composite **Snake** and the leaf **BodyElement**.

Secondly, we can also talk about the observer pattern.

Observer Pattern

Observer notifies and update automatically the dependents of one object when this object change his state.

We didn't use in our design, but we think it will be worth it to adapt this design pattern to our software. We can use this design pattern to check the state of any element of our game, for example every item can notify their positions.

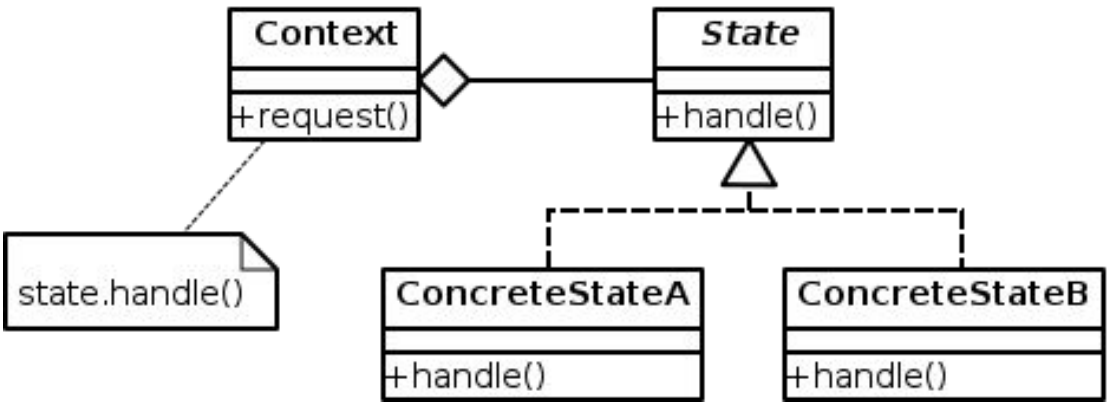


6.3. Design pattern discarded

In our designing process, we discard the state design pattern.

State Pattern
This pattern is used to encapsulate varying behavior for the same object based on its internal state.

We didn't use the state design pattern because our project doesn't need to encapsulate something and snake, wall and food don't need to change some state.



7. Versionning

7.1. Project development

About coding part of our project, we decided to use Github because it's a very useful and flexible tools, with that tool we can create a project and have some collaborators. We should avoid some conflict that why every member of our team has forked this project, on every update we make a pull request.

7.2. Source code

Our source code is available on Github, we work on branch *develop* and *master*. Master is dedicated to releasing some stable version of our project.

Link to our project: https://github.com/Orlands/snake-tj

Branch:
- master
- develop

Member Github login:
- arnaud-zg
- Orlands
- Dasheels

8. Conclusion

To conclude, this project was a good exercise of reflection and designing software. We have discovered how a design pattern can easily change the view of a project and makes the conception more simple that was originally planned. We had developed some games before, but with design pattern, it's made the code more flexible and maintainable.

The design of our project is more flexible in some point. Indeed, if we want to add some new features. We implemented the composite design pattern and that give us many possibilities like to have the ability to be fast for the snake. For example, if we want to add some feature like speed boost to the snake, we just need to add some method to the interface class.

About maintainability, we have an abstract class named ItemCrashable, this class allows us to add many kinds of item. Indeed, with this abstract class, we know every item can do same action. We just have to create a new class inherit from ItemCrashable and add some method in addition, if this object can do more things than expected. Moreover, we can bring some other feature like a destructible wall or different type of food.

---

9. Glossary of classes

9.1. Class Position

Name: Position	Store some data position in two private integers X and Y.

- `Position(int, int);`
  - ➢ Constructor of Position class with position as parameter.
- `virtual ~Position();`
  - ➢ Destructor of Position class.
- `void setX(int);`
  - ➢ Set variable _x equal to the integer given as parameter.
- `void setY(int);`
  - ➢ Set variable _y equal to the interger given as parameter.
- `int getX(void);`
  - ➢ Return integer value of the variable _x.
- `int getY(void);`
  - ➢ Return integer value of the variable _y.
- `void toString(void);`
  - ➢ Display some debugging string about Position class.

9.2. Class Controller

Name: Controller	Manage some interaction with user input.

- `Controller(int, int);`
  - ➢ Constructor of Controller class with position as parameter.
- `virtual ~Controller();`
  - ➢ Destructor of Controller class.
- `bool makeMap(int, int);`
  - ➢ Build a map with a new instance of the class Map.
- `bool keyPressed(void);`
  - ➢ Call a method of event catcher of any key pressed by user.
- `void gameOver(void);`
  - ➢ Stop the game and tell some credits.
- `bool startGame(void);`
  - ➢ Start the game and initialize some components.

9.3. Class Map

Name: Map	That class represent a map, and will make some action as check item and generate item.

- `Map(int, int);`
  - ➢ Constructor of the Map class with map size as parameter.
- `virtual ~Map();`
  - ➢ Destructor of the Map constructor.
- `bool check(Direction);`
  - ➢ Tell others components a key is pressed by user.
- `bool checkCell(Position *);`
  - ➢ That function will find the right cell to be checked, and will tell to that cell there is some collision between the snake and the item of the cell.
- `bool checkAndGenerateFood(void);`
  - ➢ That function will make a loop and generate a new item in a random position if there is no food in map.

9.4. Class Cell

Name: Cell	A Cell is a components of a map, every cell have only one item.

- `Cell(int, int);`
  - ➢ Constructor of the Cell class.
- `virtual ~Cell();`
  - ➢ Destructor of the Cell class.
- `Position *getPosition(void);`
  - ➢ This method will return the current position of cell.
- `bool checkItem(Snake **);`
  - ➢ This function will check if there is some collision between cell's item and snake.
- `Position *addItem(Food *);`
  - ➢ This method will store the Food item given as parameter in the current cell.
- `Position *addItem(WallElement *);`
  - ➢ This method will store the WallElement item given as parameter in the current cell.
- `bool deleteItem(void);`
  - ➢ This method will delete the current item stored in the cell.

9.5. Class ItemCrashable

Name: ItemCrashable	This class is a abstract class, it is a skeleton of every item of this game.

- `virtual Position *getCurrentPosition() = 0;`
  - ➢ Return the position of the current item.
- `virtual void setPosition(int, int) = 0;`
  - ➢ Set a new position of item with X and Y value given as parameters.
- `virtual bool isCollide(Snake **) = 0;`
  - ➢ Check is there is some collision between item and snake.

9.6. Class WallElement

Name: WallElement	This class is a item and represent a wall.

- `WallElement(int, int);`
  - ➢ Constructor of the WallElement class with position as parameter.
- `virtual ~WallElement();`
  - ➢ Destructor of WallElement class.
- `virtual Position *getCurrentPosition();`
- `virtual void setPosition(int, int);`
- `virtual isCollide(Snake **);`

9.7. Class Food

Name: Food	This class is a item and represend a food.

- `Food(int, int);`
  - ➢ Constructor of the Food class with position as parameter.
- `virtual ~Food();`
  - ➢ Destructor of Food class.
- `virtual Position *getCurrentPosition();`
- `virtual void setPosition(int, int);`
- `virtual bool isCollide(Snake **);`

9.8. Class ISnake

Name: ISnake	This class is a interface class, it is a skeleton of every snake elements.

- `virtual Position *getNextPosition() = 0;`
  - ➢ Return the next position of the body element, every part of the snake have a direction.
- `virtual Position *check() = 0;`
  - ➢ This method will check if everything is right, and return the current position.
- `virtual bool move() = 0;`
  - ➢ This method will make some movement with snake components.

9.9. Class Snake

Name: Snake	This class contains every body part of the snake.

- `Snake(int, int);`
  - ➢ Constructor of the Snake class with position given as parameters.
- `virtual ~Snake();`
  - ➢ Destructor of the Snake class.
- `virtual Position *getNextPosition();`
  - ➢ Return the position where the head of the snake is looking for.
- `virtual Position *check(Direction);`
  - ➢ This method will check if everything is right, and return the current position.
- `virtual bool move();`
  - ➢ This method will make some movement with snake components.
- `bool checkSelfCollision();`
  - ➢ Check every component of the snake if there is some self collision.
- `bool add();`
  - ➢ Add a body element of the snake at the end.
- `bool remove(int);`
  - ➢ Delete a BodyElement of the snake in the index given as parameter.

9.10. Class BodyElement

Name: BodyElement	A snake is composed with one or many body element.

- `BodyElement(int, int);`
  - ➢ Constructor of BodyElement class with position given as parameters.
- `virtual ~BodyElement();`
  - ➢ Destructor of BodyElement.
- `virtual Position *getNextPosition();`
  - ➢ Return the position where the head of the snake is looking for.
- `virtual Position *check(Direction);`
  - ➢ This method will check if everything is right, and return the current position.
- `virtual bool move();`
  - ➢ This method will make some movement with snake components.
- `Position *getPosition();`
  - ➢ Return the current position of this class.
- `Direction getCurrentDirection();`
  - ➢ Return the current direction of this class.
- `void setPosition(int, int);`
  - ➢ Set a new position of this body element of the snake.
- `void setCurrentDirection(Direction);`
  - ➢ Set a new direction of this body element of the snake.
- `int getX();`
  - ➢ Return current X position.
- `int getY();`
  - ➢ Reutnr current Y position.

9.11. Class InputEvent

Name: InputEvent	This class will catch if there is some key event.

- `InputEvent();`
  - ➢ Constructor of InputEvent class.
- `virtual ~InputEvent();`
  - ➢ Destructor of InputEvent.
- `Direction getPressedKey();`
  - ➢ Catch if there is some key pressed event.
- `void toString();`
  - ➢ Display some debugging string about InputEvent class.