Project Report Dev4

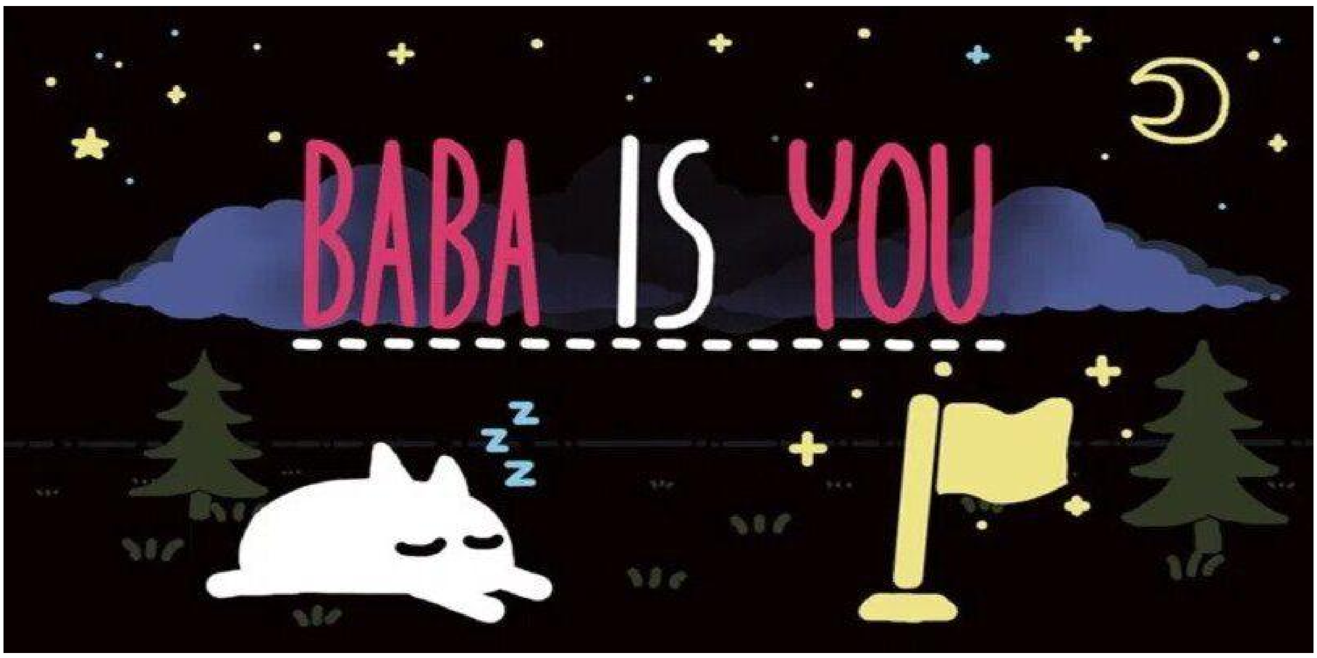


Table of contents

Introduction

For this project we will recreate a game called "Baba Is You". It is a puzzle game. The particularity of this game is that the rules governing allowed player behavior are directly visible on the game board. These can be changed at any time by pushing an element of the rule to move it to a new place on the game board.
For example, if there is the sequence, "Wall Is Stop", as soon as you push one of the three elements and they are no longer aligned horizontally or vertically in the reading direction, then you can pass through the walls. The arrangement of rule blocks make rules,that can kill you, block you, make you win or move objects around the map.
For our implementation, we will recreate the first 5 levels of the game. We may add in extra levels if time permits.

For this project, we have to render 3 different iterations, which are called “Metier”, “console” and “GUI”. In this first iteration (February 17), we need to hand in a UML representation of the project and the method signatures of each class, this part of the project isn’t functional. For the second iteration (April 21), an executable and functional version in console format must be submitted and for the last iteration(May 26), an executable and functional GUI version must be submitted. Tests must also be carried out to verify the correct implementation of the methods.

For this project we are in groups of 2 students. We are going to be coding on QTCreator in C++ and for the UML representation of the project we will be using StarUML and then exporting it into a svg version. The documentation for the code will be written in Doxygen format.

All the code and the documentation will be written in English. My colleague and I found it better, than writing it in French, for our future work.

First Iteration: "Metier"

In this section, as mentioned in the introduction, we will be first explaining our UML representation and our motivations for creating it that way. Then we will be explaining all of our methods of each class.

UML REPRESENTATION (SEE APPENDIX 1)

1. ENUMERATION CLASS "DIRECTION" AND CLASS "POSITION"

The Direction class is an interesting one. We will use a private constructor to instantiate the different attributes of this class. This attribute will be static const attributes so they can't change during the execution of the program.

The 4 attributes will be the 4 cardinilaties:

- static const Direction **N**;
- static const Direction **S**;
- static const Direction **W**;
- static const Direction **E**;

Each of those directions hold a pair of Integers that represents the dx and dy values. Those values are the difference between the initial position and the position after its movement towards the direction.

Note: There is Also a 5th one that represents no direction at all: static const Direction **NONE.**
Its dx and dy values are null. It will be useful in case we need to provide a direction that has no values.

The two classes will be useful when moving the player and other objects around the board as we will see later in the other classes. The position class will have an x and y attribute and will have a bunch of operators (for example operator +=, operator +, operator <=> , etc…).

2. ENUMERATION "TYPE" AND ENUMERATION "GAMESTATE"

This enumeration defines 4 literals: enum class **Type**{**BEHAVIOR**,**SUBJECT**,**CONNECTOR**,**BLOCK**}.
The literals "**SUBJECT**"," **CONNECTOR**"," **BEHAVIOR**" will be very useful when checking for rules on the board. We will have to look for example for the sequence:
"**SUBJECT**"," **CONNECTOR**"," **BEHAVIOR**".

The "**BLOCK**" literal represents an ordinary block on the board (such as a Wall or a Rock).
We used an enumeration so the number of different types of blocks are already defined. This enumeration is fixed and should not be modified in the future.

We will also have a **GameState** enumeration which will be useful for the console but also the GUI view. This enum has 4 states: enum class **GameState**{**PLAYING**,**WON**,**LOST**,**PAUSED**}. The state will be useful to print on the screen the information that the player needs to understand what happened or what he must do.

3. Class Block

This is one of the main classes of our game. It will contain all the information about the blocks on the board. We created one class for all the type of blocks as they all share the same number of attributes and the same type of attributes.

A. Attributes

We have 5 different attributes for this class:

- **pos**: This attribute contains the position of the block on the board.
- **type**: This attribute will be useful when identifying the different elements on the board for the creation of the rules. We will have to identifyfor example the sequence "**SUBJECT**, **CONNECTOR**,**BEHAVIOR**" to find a rule on the board. And for the blocks that aren't used to create rules, they will have as type **BLOCK**.
- **name**: The name of the block. For example, Wall or Rock, etc.

B. Methods

This class doesn't have any special method. It will have a constructor and methods to get and set attributes. The class only has to modify the board and check the player isn't going out of bounds. This class also has an bool operator**==**and an std::ostream& operator**<<** the first one will be used to compare two blocks and the second one to print out the block on the board.

4. Block Factory class

This class will allow us to create different instances of the Block class. All currently instantiable blocks are stored in a std::map<std::string, Block> **blocks** map**.** We decided to create a std::map so when we read the text file to create the level, we can only create a new Block object that exists in the game. For example, if mouse is written in the text file we can't create a new mouse because in our levels there aren't any mouses**.** We will be using this std::map when we create the new Blocks with the method Block& **instanciateBlock** (std::string **name**) which returns a reference to the new Bock.
We used a map because it is the quickest way to identify a value to a key and the manipulation of a map is always O(1) which is the quickest algorithm that exists.

5. Class Board

This class is the main class of our game. It will be as it names says, the board of the game. It will represent the level, our current game. It will allow us to save our board, load a level and much more. For this class we will declare an alias: using **BlockRef** = std::reference_wrapper<Block>. This alias will be very useful in the whole class. It is a std::reference_wrapper<Block>. This will allow us to instantiate a std::vector of references which is much easier to handle than a vector of pointers since we can't have any nullptr exceptions.

A. Attributes

This class doesn't have many attributes. It only has, and needs, the height of the Level, the width of the Level and a std::map<Position,std::vector<Block>> **tiles** which stores all of the blocks on the board at their respective positions. The std::vector is used for stacking blocks within each position's std::vector in tiles.

We thought of using a std::stack but it wouldn't have been easy to change the appearance of the blocks that weren't on the top of the stack. We would have to unstack all the elements, modify them then restack them. This wouldn't have been efficient.

The key to our std::map is the Position, this will be useful when looking for a block at a certain position. We will just have to search for the key then take the last element of the vector. We will always insert the blocks at the end of the vector. The element on the top will then always be the last element of the vector. The map also has the quickest methos to access it (complexity O(1)). The std::vector has a complexity of O(n) which is quick but not the quickest but we didn't find another way to make it quicker.

B. METHODS

The Board class will have the method to save the board. We will be using the method void **saveBoard**(int **levelNumber**, std::string **saveFolderPath**)to save the board, we will create a text file that can be then read to create the level we want to play on or restore a saved Board. We will also save in this text file the current level number.
To create the board, we will have the method std::optional<int> Board::**loadLevel**(std::string **path**, bool **isSave**). This method returns an optional, either a level number or not. If it is a save, they will be a levelNumber to know which level we are at.

The Board class will also contain the methods that will enable us to add and remove a block at a certain position. These methods will be useful when creating the board or modifying it (for example, if the player pushes a block into lava we have to be able to destroy it).

The Board, will have a method std::vector<BlockRef> **getBlock**(std::string **identity**, Type **type**) to get all of the blocks of a certain type and name and also a method Block **getBlock**(Position **pos**) to get the block at a certain position on the board. We will be using these methods to apply the rules of the game and to move the blocks around on the board.
This method bool **isInsideBoard**(Position **pos**) will allow us to check if the position is in the board or not. We will be using it when moving the player around the board.

We also added in a method void **replaceBlocks**(String **targetName,** String **newName**) to replace all the blocks of one type into blocks of the new type. This method will be useful when we will have to change the appearance of one type of block with the new appearance (for example, if the player creates the rule Wall is Rock, this method will be called to change all the Wall blocks in Rock blocks).

For the movement on the board, we will use the void Board::**moveBlock**(Block& **block**, Position **destination**). This method will be used to move a block to a destination position. To do this we will remove the block at his position, it will always be the last element. We will then set his position to the new position then add it to the end of the std::vector at this position in the board.

6. CLASS GAME

This class is the main class of the Game. It contains all the methods to control the game and the main attributes that we will have to use at any time.

A. ATTRIBUTES

This class has four important attributes:

- **player**: This references the name of the block that will be the current player.

- **Connectors**: This is a std::vector of all the connectors on the board. This list will be useful when we will have to find the rules around the board. We only need to look around the connectors to find a new rule.
- **Board**: This is an instance of the class Board.
- **CurrentLevel**: this is an Integer that will keep track of which level the player is playing.

B. : METHODS

This class will contain all the important methods to control the Game

- bool **isFinished**() => indicates if the current game (level) finished or not
- bool **startgame**() =>starts the current game Level
- bool **saveGame**() => saves the current game by creating a new file
- bool **loadGame**(int level) => loads the given level and starts the game.
- bool **loadSave**()=> loads the saved game from its save file
- std::vector<BlockRef> **getPlayerBlocks**() const -> this will get all of the player blocks on the board.

This class will also have the method void **move**(Direction **dir**). This is the main method of the Game. When the void **move**(Direction **dir**) method is called, for every instance of a player, we will check if at the new position of the player(after the move), this instance of player is on a block. If this instance is on a block, we will then execute the rule of this block otherwise we will ignore the block and stack on top of it.
Every time this method is called, we will call the method void **move**(Block& **block,** Position **destination**) in the class Board. This method will move the blocks in the board then will call void **notifyObservers**(). This method will then call the void update() method of the class Game which will empty the rules and scan around the board to find the new rules created by doing this move.
The class Board will implement Observable and the class Game will implement Observer. This will allow us to keep the rules up to date without having to do anything. It will be useful when creating the GUI version of this project.

There are other methods to check the rules and apply them:

- void Game::**fetchAndApplyRules**()
- void Game::**applyRule**(RuleChain **ruleChain**)
- void Game::**removeRule**(RuleChain **ruleChain**)
- std::optional<RuleChain> Game::**fetchRule**(Position **rulePos**, CheckDirection **dir**, Block& **block**)
- void Game::***updateRuleChange***(Position **oldPos**, Position **newPos**, Block& **ruleBlock**)

The first method will be used to fetch and apply the rules. We will be using the **fecthRule** method which looks for Rules vertically and horizontally and creates the rules, then applies them using the **applyRule** method.

The ***updateRuleChange*** method will be used for the Observer, Observable. We will check for the eventual rule breaks and rule formations.

7. RULECHAIN

This class will be used to check if there is a rule on the board. There is a constructor to create the RuleChain and check if the rule is valid or if the rule transforms the block (for example the Block is Wall rule).

8. Factory Class, Class Rules and childrens

A. class Rule

This class is the parent Class of the Rules. It will have as attributes the name of the Rule and a reference to the board so we can change it when we execute the rule. It will also have a method virtual void ***execute***(Block& **targetBlock**, Block& **collidingBlock,** Positions& **collidingBlockInitialPos**)which the child classes will implement differently depending on the specific rule's behavior. By defining this method as virtual, it allows flexibility and extensibility in the design, as new rules can be added without modifying the base class.

B. Rule Factory class

This class will allow us to create different instances of the Rule class. All currently instantiable rules are stored in a std::map<std::string, Rule> **rules** map**.** We decided to create a map so when we read the text file to create the level, we can only create a new Rule object that exists in the game. For example, if text_mouse is written in the text file we can't create a new mouse because in our levels they aren't any mouse rule**.** We will be using this std::map when we create the new Rules with the method Rule& **instanciateRule**(std::string **name**) which returns a reference to the new Rule.
It also contains a std::map<std::string, std::set<Rule>> containing all currently active rules in the game, which will be used by the different rules in their execute method.

C. Rule sub classes

They implement the execute function of the parent Rule class. Each rule has its own execute and executes a certain way.

D. GameRules

This class will contain all the rules in the current game. It will have as attribute:

- std::map<std::string, std::vector<std::string>> **rules**

This is a map of all of the rules on the board. Each block will be a key and when a new rule is created, it will be added to the vector of the block. This will allow us to have more than one Rule on a block and to execute the rule directly on the concerned block. Each block on the board can have multiple rules and each block will have its own std:: vector <Rule> in the std::map. We will use the std::string to identify the block on which the rules are applied.

In this class we will have the following methods:

- void **addRule**(std::string **blockName**, std::string **ruleName**)
- void **removeRule**(std::string **blockName**, std::string **ruleName**)
- std::string **getBlockRuleName**(std::string **blockName**, Type **blockType**)
- std::vector<std::string> **getPlayer**()

The **getBlockRuleName** method we will be used to return the rule on the block. If the block has a rule, this method will return the rule of this block, otherwise it will return the inactive. If it is a rule block, this method will return push, which is the default rule on this type of block.

The **getPlayer** method will return all the block with the you rule. We will identify the "you" rule with the PLAYER_IDENTIFIER attribute.

9. Observer and Observable

Observer and Observable are interfaces. It will be used when the player collides with a Rule on the Board (a Behavior, a connector or a Subject). When this player is moved in the board by the move method in the Game class, the board will identify the block the player is colliding with. If it is a rule, we will execute the rule’s push behavior and notify the Observers (Class Game) and the Game class will call it’s update method which updates the std::map of rules.

Second Iteration: "Console"

1. Block class

This class doesn't have any particular methods apart from:

- bool operator==(Block **block**, Block **other**)
  This operator will be used to check that 2 blocks are the same. We only check that the attribute name is the same for the two blocks. If they are the equal it will return True otherwise false.
- std::ostream& operator<<(std::ostream& **out**, Block& **b**)
  This operator is used to print the blocks on the screen. We print the consoleName attribute of the block on the screen. This will be used for the console view.

2. Block Factory

This class will contain a std::map with all of the blocks that can be on the board. This map will be very important. We used a map since the difficulty is O(1) which is the quickest search algorithm.

This map will look as follows:

```
std::map<std::string, Block> blocks{
        {"baba", Block("baba",Type::BLOCK, pos, " $ ",dir)},
        {"text_baba", Block("text_baba",Type::SUBJECT, pos, "bba",dir)},
        {"rock", Block("rock",Type::BLOCK, pos," 0 ", dir)},
        {"text_rock", Block("text_rock",Type::SUBJECT, pos,"rok", dir)},
        {"wall", Block("wall",Type::BLOCK,pos," = ", dir)},
        {"text_wall", Block("text_wall",Type::SUBJECT, pos,"wal", dir)},
        {"grass", Block("grass",Type::BLOCK, pos," ` ", dir)},
        {"text_grass", Block("text_grass",Type::SUBJECT, pos,"grs", dir)},
        {"metal", Block("metal",Type::BLOCK, pos,"[ ]", dir)},
        {"text_metal", Block("text_metal",Type::SUBJECT, pos,"mtl", dir)},
        {"lava", Block("lava",Type::BLOCK, pos," # ", dir)},
        {"text_lava", Block("text_lava",Type::SUBJECT, pos,"lva", dir)},
        {"flag", Block("flag",Type::BLOCK, pos," F ", dir)},
        {"text_flag", Block("text_flag",Type::SUBJECT, pos,"flg", dir)},
        {"best", Block("best",Type::BLOCK, pos," * ", dir)},
        {"text_best", Block("text_best",Type::SUBJECT, pos,"bst", dir)},
        {"water", Block("water",Type::BLOCK, pos,"~~~", dir)},
        {"text_water", Block("text_water",Type::SUBJECT, pos,"wtr", dir)},
        {"is", Block("is",Type::CONNECTOR, pos,"i_s", dir)},
        {"you", Block("you",Type::BEHAVIOR, pos,"you", dir)},
        {"kill", Block("kill",Type::BEHAVIOR, pos,"kll", dir)},
        {"push", Block("push",Type::BEHAVIOR, pos,"psh", dir)},
        {"stop", Block("stop",Type::BEHAVIOR, pos,"stp", dir)},
        {"best", Block("best",Type::BEHAVIOR, pos,"bst", dir)},
        {"sink", Block("sink",Type::BEHAVIOR, pos,"snk", dir)},
        {"win", Block("win",Type::BEHAVIOR, pos,"win", dir)},
        {"void", Block("void",Type::BLOCK, pos,"   ",dir)}
  };
```

The other interesting method in this class will be the Block **instanciateBlock**(std::string **name**, Position **position**, Direction **direction** = Direction::NONE). This method will look for the constructor for the block with the same name as the name parameter. Then it will instantiate this block using the constructor and the position and direction parameter.

3. Board

The methods with a certain complexity are:

- The following methods with O(1) complexity:
    - void Board::**addBlock**(Block& **block**, Position **pos**) -> worst case -> 2 elementary operations
    - Block Board::**pop**(Position **pos**) -> worst case -> 2 elementary operations

    All of these methods have 0(1) complexity since they are searching in a map and taking or adding an element at the end of the vector at the position in the map.

- The following methods with O(n^2) complexity:
    - void Board::**saveBoard**(int **levelNumber**, std::string **saveFolderPath**) -> worst case -> $n^2 + 2n$ elementary operations
    - std::vector<BlockRef> Board::**getBlock**(std::string **identity**, Type **type**)
    - void Board::**replaceBlocks**(std::string **targetName**, std::string **replacementName**) -> worst case -> $n^2 + n$ elementary operations.

    These methods have a double for loop because they have to go through the whole board: std::map<std::vector>>.

4. GameRules

For this class, they aren't any complex algorithms apart from std::vector<std::string> GameRules::**getPlayer**() method which uses a standard for loop to iterate through the rules of the game and look for the objects that have the rule "you".

5. Position ,RuleChain and Direction

There aren't any complex algorithms in this class.

6. Game

The following method is with $2n^2+n$ elementary operations:

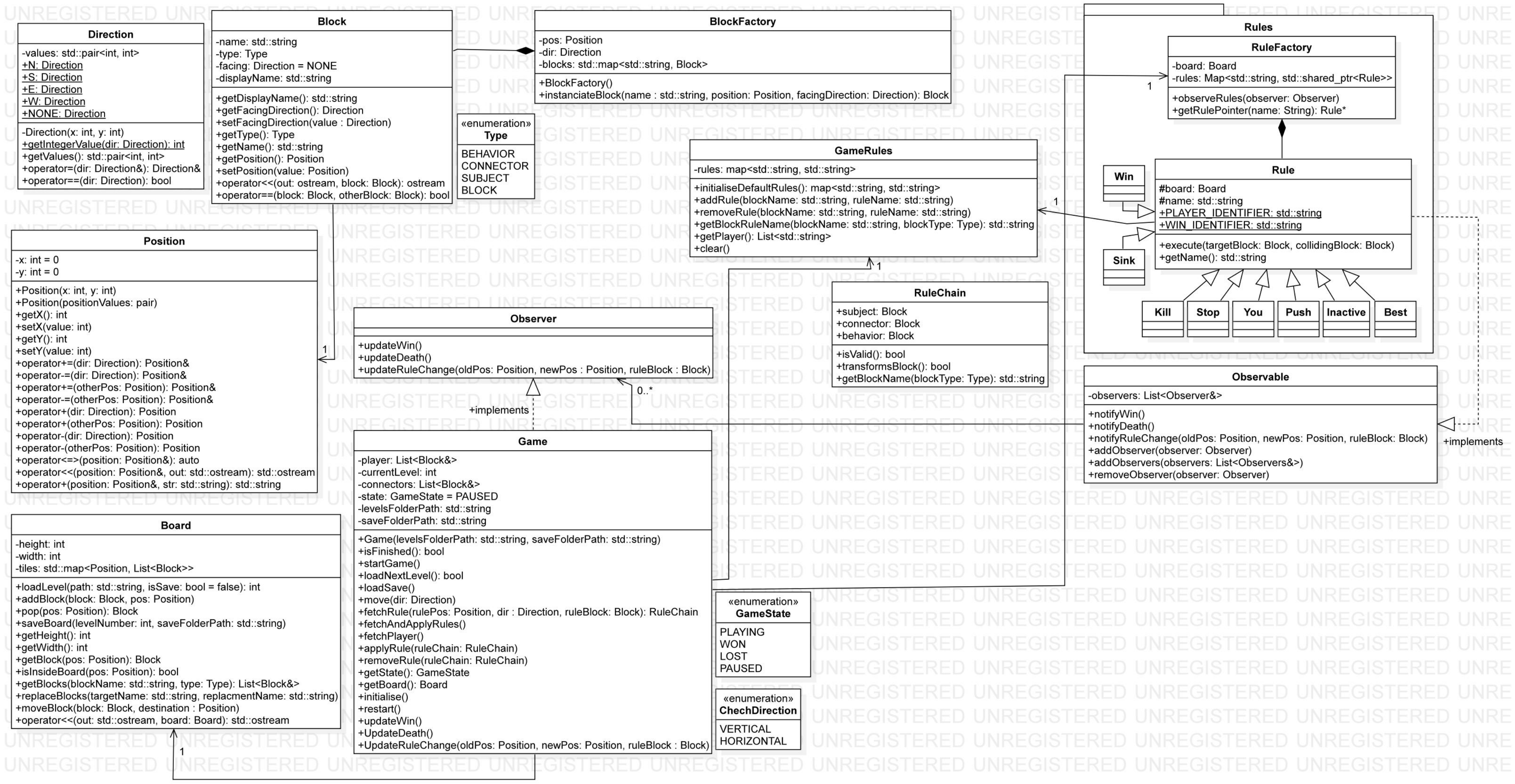
- void Game::**fetchPlayer**()

This method uses a standard foreach loop to go through the playerBlock with is populated with the return result of the getPlayer method that we talked about in the GameRules class. We then populate the std::vector<BlockRef> **blocks** with the return value of the getBlock method. We then go through the blocks variable in the foreach loop.

7. Rules

In the Rules section of the project, we don't have any complex algorithms. Only standard for loops and calls to methods seen before.

Annexe 1 : UML Class Diagram


Direction
-values: std::pair<int, int>
+N: Direction
+S: Direction
+E: Direction
+W: Direction
+NONE: Direction
-Direction(x: int, y: int)
+getIntegerValue(dir: Direction): int
+getValues(): std::pair<int, int>
+operator=(dir: Direction&): Direction&
+operator==(dir: Direction): bool
Block
-name: std::string
-type: Type
-facing: Direction = NONE
-displayName: std::string
+getDisplayName(): std::string
+getFacingDirection(): Direction
+setFacingDirection(value : Direction)
+getType(): Type
+getName(): std::string
+getPosition(): Position
+setPosition(value: Position)
+operator<<(out: ostream, block: Block): ostream
+operator==(block: Block, otherBlock: Block): bool
«enumeration» Type
BEHAVIOR
CONNECTOR
SUBJECT
BLOCK
BlockFactory
-pos: Position
-dir: Direction
-blocks: std::map<std::string, Block>
+BlockFactory()
+instanciateBlock(name : std::string, position: Position, facingDirection: Direction): Block
Rules
RuleFactory
-board: Board
-rules: Map<std::string, std::shared_ptr<Rule>>
+observeRules(observer: Observer)
+getRulePointer(name: String): Rule*
1
Win
Rule
#board: Board
#name: std::string
+PLAYER_IDENTIFIER: std::string
+WIN_IDENTIFIER: std::string
+execute(targetBlock: Block, collidingBlock: Block)
+getName(): std::string
Sink
Kill
Stop
You
Push
Inactive
Best
GameRules
-rules: map<std::string, std::string>
+initialiseDefaultRules(): map<std::string, std::string>
+addRule(blockName: std::string, ruleName: std::string)
+removeRule(blockName: std::string, ruleName: std::string)
+getBlockRuleName(blockName: std::string, blockType: Type): std::string
+getPlayer(): List<std::string>
+clear()
1
1
Position
-x: int = 0
-y: int = 0
+Position(x: int, y: int)
+Position(positionValues: pair)
+getX(): int
+setX(value: int)
+getY(): int
+setY(value: int)
+operator+=(dir: Direction): Position&
+operator-=(dir: Direction): Position&
+operator+=(otherPos: Position): Position&
+operator-=(otherPos: Position): Position&
+operator+(dir: Direction): Position
+operator+(otherPos: Position): Position
+operator-(dir: Direction): Position
+operator-(otherPos: Position): Position
+operator<=>(position: Position&): auto
+operator<<(position: Position&, out: std::ostream): std::ostream
+operator+(position: Position&, str: std::string): std::string
1
RuleChain
+subject: Block
+connector: Block
+behavior: Block
+isValid(): bool
+transformsBlock(): bool
+getBlockName(blockType: Type): std::string
Observer
+updateWin()
+updateDeath()
+updateRuleChange(oldPos: Position, newPos : Position, ruleBlock : Block)
0..*
+implements
Observable
-observers: List<Observer&>
+notifyWin()
+notifyDeath()
+notifyRuleChange(oldPos: Position, newPos: Position, ruleBlock: Block)
+addObserver(observer: Observer)
+addObservers(observers: List<Observers&>)
+removeObserver(observer: Observer)
+implements
Game
-player: List<Block&>
-currentLevel: int
-connectors: List<Block&>
-state: GameState = PAUSED
-levelsFolderPath: std::string
-saveFolderPath: std::string
+Game(levelsFolderPath: std::string, saveFolderPath: std::string)
+isFinished(): bool
+startGame()
+loadNextLevel(): bool
+loadSave()
+move(dir: Direction)
+fetchRule(rulePos: Position, dir : Direction, ruleBlock: Block): RuleChain
+fetchAndApplyRules()
+fetchPlayer()
+applyRule(ruleChain: RuleChain)
+removeRule(ruleChain: RuleChain)
+getState(): GameState
+getBoard(): Board
+initialise()
+restart()
+updateWin()
+UpdateDeath()
+UpdateRuleChange(oldPos: Position, newPos: Position, ruleBlock : Block)
Board
-height: int
-width: int
-tiles: std::map<Position, List<Block>>
+loadLevel(path: std::string, isSave: bool = false): int
+addBlock(block: Block, pos: Position)
+pop(pos: Position): Block
+saveBoard(levelNumber: int, saveFolderPath: std::string)
+getHeight(): int
+getWidth(): int
+getBlock(pos: Position): Block
+isInsideBoard(pos: Position): bool
+getBlocks(blockName: std::string, type: Type): List<Block&>
+replaceBlocks(targetName: std::string, replacmentName: std::string)
+moveBlock(block: Block, destination : Position)
+operator<<(out: std::ostream, board: Board): std::ostream
1
«enumeration» GameState
PLAYING
WON
LOST
PAUSED
«enumeration» ChechDirection
VERTICAL
HORIZONTAL