Report Pokémon Multiplayer

Mattia Ricci, matr:815225, email: `mattia.ricci8@studio.unibo.it`
Francesco Dicara, matr:797532, email: `francesco.dicara@studio.unibo.it`
Gianluca Grossi, matr: 808134, email: `gianluca.grossi2@studio.unibo.it`
Giulia Lucchi, matr:807407, email: `giulia.lucchi3@studio.unibo.it`
Luca Polverelli, matr:807419, email: `luca.polverelli3@studio.unibo.it`

repo: https://github.com/Mattia23/S3-16-poke-mp

14/08/2017

Software development process

We adopted an Agile software development process based on Scrum.

We have chosen Mattia Ricci as Product Owner and, during the first meeting, we have decided to have weekly meetings where weekly tasks were agreed. During each sprint planning, starting from the product backlog, we reviewed high-priority items and discussed goals in order to define the backlog for the given sprint.

Tasks were equally split between team members, in such a way that each member could develop different project features.

After every sprint, we discussed what went good and what went bad, talking about them and trying to resolve problems within the following sprint.

Development support tools we used are:

- IntelliJ IDEA as integrated software environment
- Scala and Java as programming languages
- ScalaTest as testing tool

In addition, to have a continuous and automatic integration we used:

- Git as version control system
- GitHub as version control repository
- Travis CI as continuous integration service
- Trello as web project management application
- Gradle as build automation system

Requirements

Business Requirements:

- Create a multiplayer, distributed Pokémon game, where users (Pokémon trainers) can play within the same map.

Users Requirements:

- Player should be able to see other users within the map
- Player should be able to catch a wild Pokémon
- Player should be able to interact with other players in order to fight against them
- Player should be able to heal its Pokémons
- Player should be able to change its carrying Pokémons with its other caught Pokémons
- Player should be able to see its information (level, experience points, carrying Pokémons, etc)
- Player should be able to see the global trainers rank
- Player should be able to see the Pokédex

Functional Requirements:

- The map should be a 2D map and composed by different elements
- Within the map should be present a Pokémon Center where a trainer should be able to heal its Pokémons
- Within the map should be present some grass areas where the trainer should be able to find and catch wild Pokémons
- At the beginning of a new game a trainer should be able to choose a starter Pokémon between three Pokémons (Bulbasaur, Charmender, Squirtle)
- A trainer should be able to catch and have an unlimited number of Pokémons
- A trainer should be able to carry at most 6 Pokémon
- A Pokémon should have given attributes (level, attack moves, life points, experience points)
- A trainer should have given attributes (nickname, level, experience points)
- Levels and experience points should be able to grow according to victories (against trainers and wild Pokémons).
- The trainer level will determine the level of wild Pokémons that can appear in the grass
- Trainers should be able to catch wild Pokémons through a battle
- The catch probability must be determined by some factors (Pokémon's life, Pokémon's level and trainer's level)
- Trainers should be able to fight one against another
- Battles between trainers and against wild Pokémons should be made up of different rounds
- Pokémons should be able to evolve when reaching a certain level
- Pokédex is present which keeps a list a of all the Pokémons met/caught by the trainer
- A global ranking is present, which shows all the trainers ordered by level and experience points.
- A game menu is present where the user can logout and see trainer's information (Pokedex, level, etc) and global trainers' ranking.

Non-functional Requirements:

- User's password must be encrypted within the database
- There must be no excessive game latency
- There must be no excessive connection latency

Architectural Design

The client side software was created according to the Model-View-Controller pattern, with the precise intent of making the graphic aspect completely independent from the other components of the architecture. In this way it will be much easier to organize, develop and extends each part of the system. It divides a given application into three interconnected parts in order to separate internal representations of information from the ways that information is presented to and accepted from the user.

Concerning the distributed side of the application, we use a Client-Server architecture and after some considerations, we decided that the best solution to manage messages exchange between client and server was the Message-Oriented Middleware(MOM) RabbitMQ. RabbitMQ is a message broker that accepts, stores and forwards messages. Many producers can send messages that go to one queue, and many consumers can try to receive data from one queue. Producer, consumer and broker do not have to reside in the same host.

In our case we have four main entities that compose the distributed application:

- Client: it is the client side of the application, which stores the main logic of the game.
- Server: it is the server side of the application, which stores information about players who are currently connected to the game.
- RabbitMQ: it is the message broker between clients and server
- Database: it is the database that stores all data about trainers and Pokémons which trainers met and caught.

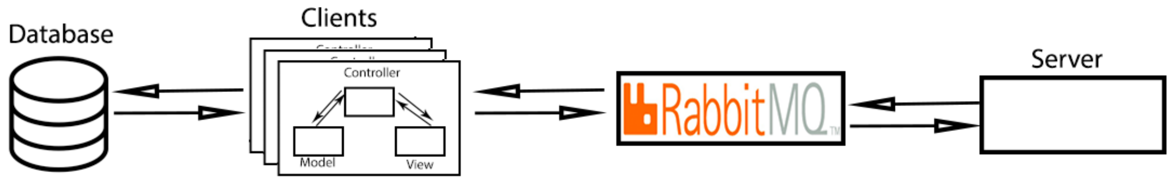


Figure 1: Distributed Architecture

About the message exchange between clients and server we used two main pattern:

- Producer/Consumer: a producer (sender) that sends a single message, and a consumer (receiver) that receives messages. This is used when two trainers are fighting or talking, when a client logs in or logs out, when the server sends to a client the current connected players, and when a client sends to the server a position update.
- Publish/Subscriber: the message that is sent by the sender will be delivered to multiple consumer. This is used when the server notifies all clients the updated position of a trainer, that a new player is logged in or logged out, that a player is/is not busy or visible.

Detailed Design

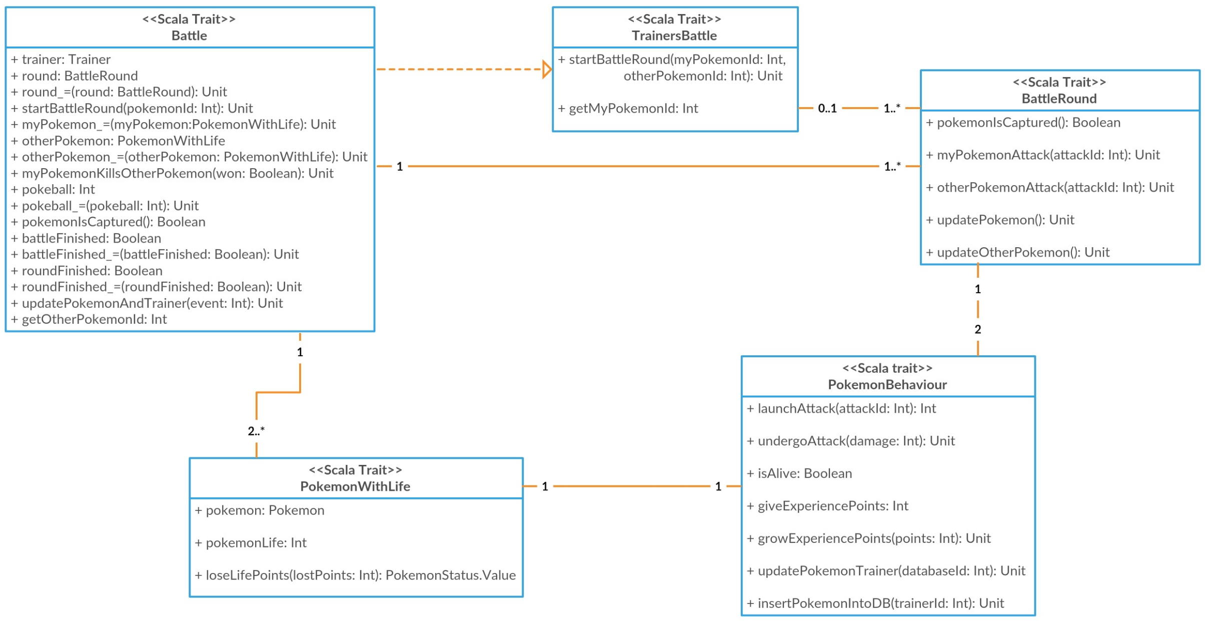
Main game entities

- Trainer: once a user logs in a new Trainer is created with his data stored in the remote database. This entity is able to move on the map and fight against wild Pokémons or other trainers. Every time a trainer fights, he earns experience points to grow his level and go to the ranking top. Trainer's level is also an important variable because it is related with the probability to find Pokémons even more rare and the probability to catch them. Every trainer has a Pokedex that shows which Pokémon have been seen and caught by the trainer. When the player walks in the map can carry up to six Pokémons that are available during battles.

- Pokémon: Pokémons are the main creatures of the game and they are characterized by different aspects: name, level, experience points, attack moves. Battle after battle a Pokémon grows his experience and consequently his level. For this reason some Pokemon can evolve in the next stage. Pokémons are not instantiated when the user logs in but they are created time by time if necessary (during the battle). Trainer's Pokémons are created retrieving the data from the remote database, whereas wild ones are created from zero randomly but considering the following aspects:
  - trainer level;
  - Pokémon rarity;
  - possible level range;
  - moves that a Pokemon can learn.

  When a Pokémon is captured, it is added to the remote database.

Pokemon battles



<<Scala Trait>>
Battle
+ trainer: Trainer
+ round: BattleRound
+ round_=(round: BattleRound): Unit
+ startBattleRound(pokemonId: Int): Unit
+ myPokemon_=(myPokemon:PokemonWithLife): Unit
+ otherPokemon: PokemonWithLife
+ otherPokemon_=(otherPokemon: PokemonWithLife): Unit
+ myPokemonKillsOtherPokemon(won: Boolean): Unit
+ pokeball: Int
+ pokeball_=(pokeball: Int): Unit
+ pokemonIsCaptured(): Boolean
+ battleFinished: Boolean
+ battleFinished_=(battleFinished: Boolean): Unit
+ roundFinished: Boolean
+ roundFinished_=(roundFinished: Boolean): Unit
+ updatePokemonAndTrainer(event: Int): Unit
+ getOtherPokemonId: Int
<<Scala Trait>>
TrainersBattle
+ startBattleRound(myPokemonId: Int, otherPokemonId: Int): Unit
+ getMyPokemonId: Int
0..1
1..*
<<Scala Trait>>
BattleRound
+ pokemonIsCaptured(): Boolean
+ myPokemonAttack(attackId: Int): Unit
+ otherPokemonAttack(attackId: Int): Unit
+ updatePokemon(): Unit
+ updateOtherPokemon(): Unit
1
1..*
1
2
1
2..*
<<Scala trait>>
PokemonBehaviour
+ launchAttack(attackId: Int): Int
+ undergoAttack(damage: Int): Unit
+ isAlive: Boolean
+ giveExperiencePoints: Int
+ growExperiencePoints(points: Int): Unit
+ updatePokemonTrainer(databaseId: Int): Unit
+ insertPokemonIntoDB(trainerId: Int): Unit
<<Scala Trait>>
PokemonWithLife
+ pokemon: Pokemon
+ pokemonLife: Int
+ loseLifePoints(lostPoints: Int): PokemonStatus.Value
1
1

Figure 2: Battle model Class Diagram

The above UML diagram shows the Model classes used during a battle against a wild Pokémon or an other trainer. We decided that a battle is composed of many rounds and every round is a fight between two specific Pokémons.

- Local battle (trainer vs wild Pokémon): When the trainer walking in the grass meets a wild Pokémon a new `BattleController` is created. Then the first Model Class instantiated is `BattleImpl` that extends `Battle`. In this class the two Pokémons are instantiated and after that a new `BattleRound` is created. At this point the view is changed by the controller. Interacting with the view, the trainer can choose if he wants to fight, to change Pokémon, to launch a Pokeball or to escape. The underlying diagram shows the sequence in the case of an attack from the trainer to the wild Pokémon. The damage caused by an attack is directly proportional to the move power and the level of the Pokémons. If the trainer's Pokémon kills the wild Pokémon the game is resumed, instead if the wild Pokémon kills trainer's Pokémon and if in the trainer's bag there is a Pokémon alive, a new round is created automatically. If the trainer chooses to change his Pokémon, a new round starts. If the trainer is interested to catch the wild Pokémon, he can launch a Pokeball: the probability to catch it is related to wild Pokémon level, life and trainer level. Only three Pokeballs are available in every battle. In the event the trainer would like to finish the battle in advance he can try to escape. The probability to escape is managed randomly by the controller. After every trainer choice the wild Pokémon reacts with an attack. At the end of every round and battle the remote database is updated.

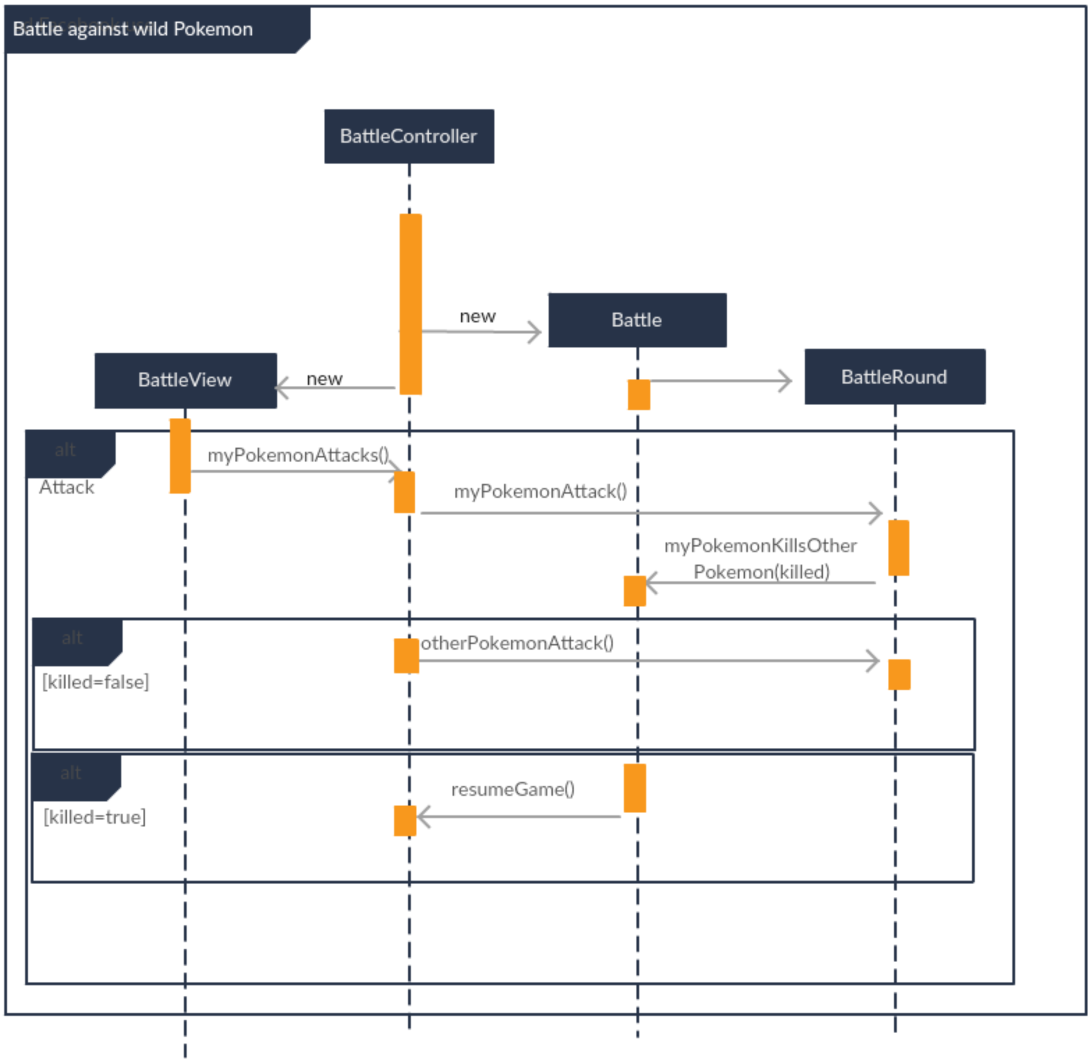


Figure 3: Battle Sequence Diagram

- Distributed battle (trainer vs trainer): when two trainers are one in front of the other, one of the two trainers can ask the other to fight. If the answer is affirmative a new `DistributedBattleController` is created and the similar classes are created. The first round is between the first two Pokémons available of the two trainers. In this type of battle the trainer has only two available choices: attack or change Pokémon. The battle lasts until one of the two trainers finishes his available Pokémons. The two trainers communicate their moves through a message exchange. As in the local battle, at the end of every round and battle the two trainers and their respective Pokémons are updated. In the diagram presented below, you can see the sequence of the battle messages exchange of a simple battle where the first trainer attacks the other trainer that answers with an attack that kills the first trainer's Pokémon, so the game is resumed in both clients.

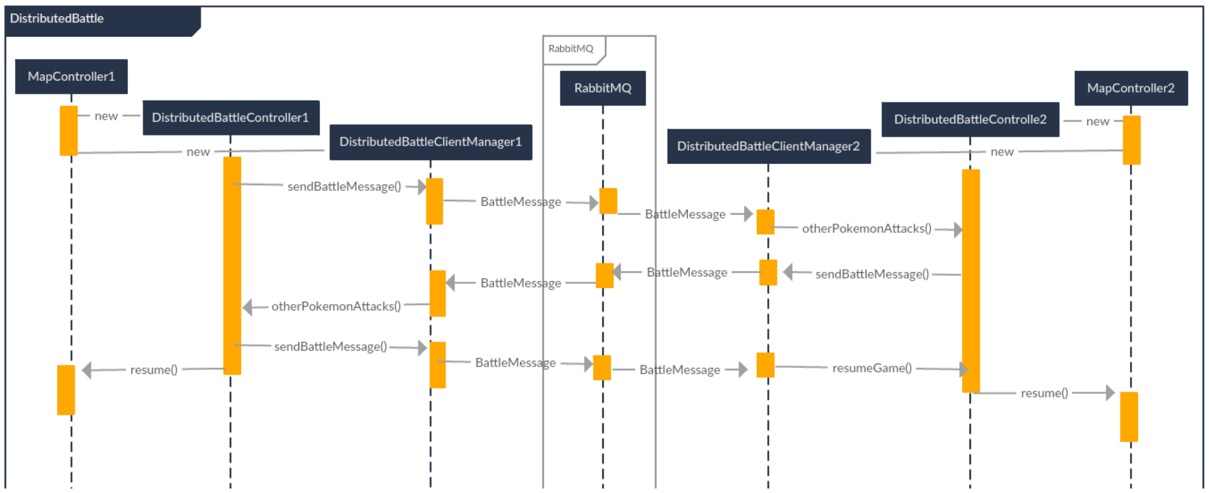


Figure 4: Distributed Battle Sequence Diagram

Game Map

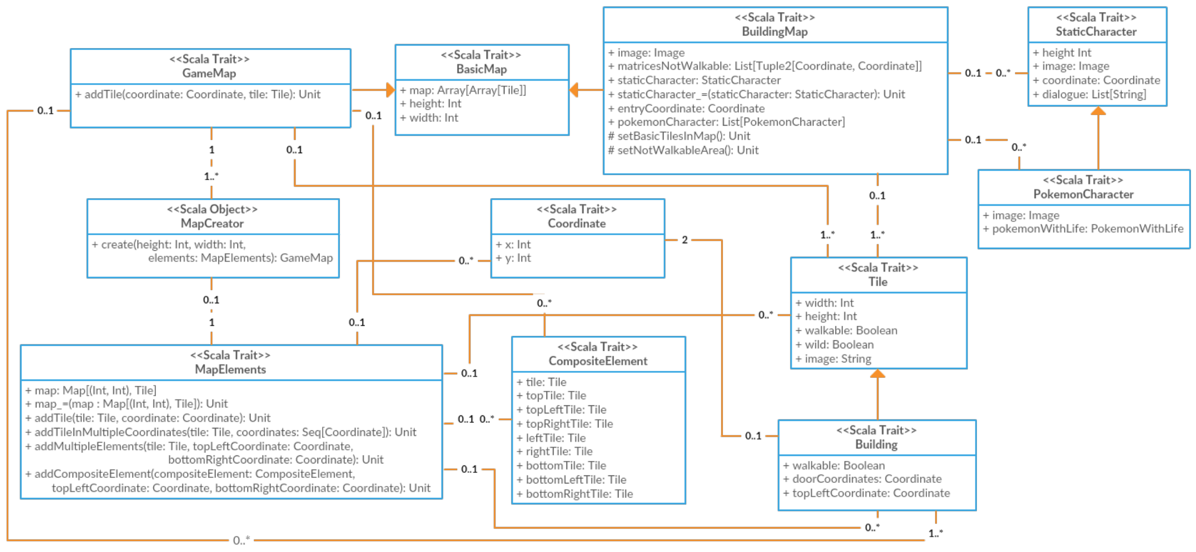

<<Scala Trait>>
GameMap
+ addTile(coordinate: Coordinate, tile: Tile): Unit
<<Scala Trait>>
BasicMap
+ map: Array[Array[Tile]]
+ height: Int
+ width: Int
<<Scala Trait>>
BuildingMap
+ image: Image
+ matricesNotWalkable: List[Tuple2[Coordinate, Coordinate]]
+ staticCharacter: StaticCharacter
+ staticCharacter_=(staticCharacter: StaticCharacter): Unit
+ entryCoordinate: Coordinate
+ pokemonCharacter: List[PokemonCharacter]
# setBasicTilesInMap(): Unit
# setNotWalkableArea(): Unit
<<Scala Trait>>
StaticCharacter
+ height Int
+ image: Image
+ coordinate: Coordinate
+ dialogue: List[String]
<<Scala Trait>>
PokemonCharacter
+ image: Image
+ pokemonWithLife: PokemonWithLife
<<Scala Object>>
MapCreator
+ create(height: Int, width: Int,
elements: MapElements): GameMap
<<Scala Trait>>
Coordinate
+ x: Int
+ y: Int
<<Scala Trait>>
Tile
+ width: Int
+ height: Int
+ walkable: Boolean
+ wild: Boolean
+ image: String
<<Scala Trait>>
MapElements
+ map: Map[(Int, Int), Tile]
+ map_=(map : Map[(Int, Int), Tile]): Unit
+ addTile(tile: Tile, coordinate: Coordinate): Unit
+ addTileInMultipleCoordinates(tile: Tile, coordinates: Seq[Coordinate]): Unit
+ addMultipleElements(tile: Tile, topLeftCoordinate: Coordinate,
bottomRightCoordinate: Coordinate): Unit
+ addCompositeElement(compositeElement: CompositeElement,
topLeftCoordinate: Coordinate, bottomRightCoordinate: Coordinate): Unit
<<Scala Trait>>
CompositeElement
+ tile: Tile
+ topTile: Tile
+ topLeftTile: Tile
+ topRightTile: Tile
+ leftTile: Tile
+ rightTile: Tile
+ bottomTile: Tile
+ bottomLeftTile: Tile
+ bottomRightTile: Tile
<<Scala Trait>>
Building
+ walkable: Boolean
+ doorCoordinates: Coordinate
+ topLeftCoordinate: Coordinate
0..1
0..1
1
0..1
1..*
0..1
1
0..1
0..1
0..*
0..1
0..*
0..1
1..*
1..*
2
0..*
0..*
0..1
0..1
0..1
0..*
0..1
0..1
0..*
1..*
0..*

Figure 5: Map model Class Diagram

The above UML diagram shows the Model classes used to create the game maps.

A map is a matrix of tiles where each tile represents a square element of the map. Different types of tile can exist (for instance: grass, tree, water, etc...), each with its own specific attributes. The game map must show the main player's trainer and all the trainers of other players who are currently connected. The game map can contain some buildings, a building is a specific type of tile with its own dimensions and attributes. The interior of buildings is also a map. Moreover, not-player character can be in the map, who the player can interact with.

Game map elements

- `Tile`: represents a single tile of the map. Main types of tiles are:
  - `Grass`: represents a grass tile which a trainer can walk above;
  - `TallGrass`: represents a tall grass tile which a trainer can walk above and find wild Pokémons;
  - `Tree`: represents a tree tile which a trainer can not walk above;
  - `Water`: represents a water tile which a trainer can not walk above;
  - `Road`: represents a road tile which a trainer can walk above;
  - `Box`: represents a box tile through which a trainer can interact and change its Pokémons;
  - `BasicTile`: represents a tile without image;
  - `Barrier`: represents a tile without image which a trainer can not walk above.

- `Building`: represents a building tile, which have bigger dimensions than a simple tile. Main types of buildings are:
  - `PokemonCenter`: represents a Pokémon center building where a trainer can heal its Pokémons. A not-player character `Doctor` is inside this building;
  - `Laboratory`: represents a laboratory building where a trainer can choose its first Pokémon. A not-player character professor `Oak` is inside this building.

- `CompositeElement`: represents a map element that is composed by more than one tile. Main types of `CompositeElement` are:
  - `Lake`: represents a lake which is composed by water tile and special types of water tiles which represent its borders;
  - `Square`: represents a square which is composed by road tile and special types of road tiles which represent its borders.

Map types

There are two types of map:

- `GameMap`: represents the map of the game composed by tiles that represent elements of the map. A game map can be made using a `MapCreator` which builds a map of specified dimensions and elements. Elements are listed into `MapElements` with their coordinate;

- `BuildingMap`: represents the content of the map of a building.

Game controller

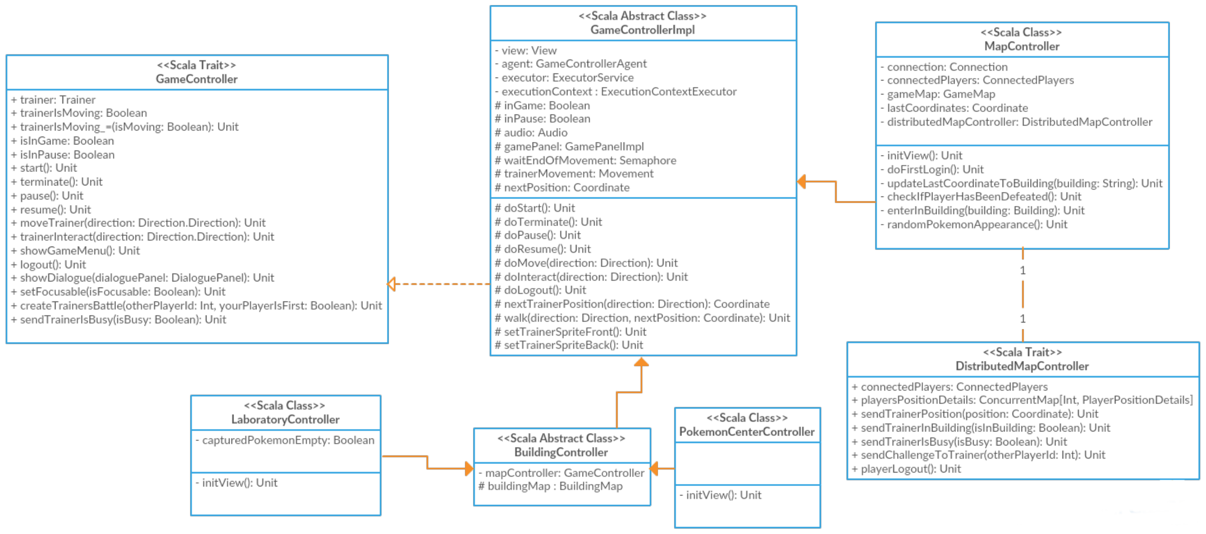

<<Scala Trait>>
GameController
+ trainer: Trainer
+ trainerIsMoving: Boolean
+ trainerIsMoving_=(isMoving: Boolean): Unit
+ isInGame: Boolean
+ isInPause: Boolean
+ start(): Unit
+ terminate(): Unit
+ pause(): Unit
+ resume(): Unit
+ moveTrainer(direction: Direction.Direction): Unit
+ trainerInteract(direction: Direction.Direction): Unit
+ showGameMenu(): Unit
+ logout(): Unit
+ showDialogue(dialoguePanel: DialoguePanel): Unit
+ setFocusable(isFocusable: Boolean): Unit
+ createTrainersBattle(otherPlayerId: Int, yourPlayerIsFirst: Boolean): Unit
+ sendTrainerIsBusy(isBusy: Boolean): Unit
<<Scala Abstract Class>>
GameControllerImpl
- view: View
- agent: GameControllerAgent
- executor: ExecutorService
- executionContext : ExecutionContextExecutor
# inGame: Boolean
# inPause: Boolean
# audio: Audio
# gamePanel: GamePanelImpl
# waitEndOfMovement: Semaphore
# trainerMovement: Movement
# nextPosition: Coordinate
# doStart(): Unit
# doTerminate(): Unit
# doPause(): Unit
# doResume(): Unit
# doMove(direction: Direction): Unit
# doInteract(direction: Direction): Unit
# doLogout(): Unit
# nextTrainerPosition(direction: Direction): Coordinate
# walk(direction: Direction, nextPosition: Coordinate): Unit
# setTrainerSpriteFront(): Unit
# setTrainerSpriteBack(): Unit
<<Scala Class>>
MapController
- connection: Connection
- connectedPlayers: ConnectedPlayers
- gameMap: GameMap
- lastCoordinates: Coordinate
- distributedMapController: DistributedMapController
- initView(): Unit
- doFirstLogin(): Unit
- updateLastCoordinateToBuilding(building: String): Unit
- checkIfPlayerHasBeenDefeated(): Unit
- enterInBuilding(building: Building): Unit
- randomPokemonAppearance(): Unit
1
1
<<Scala Trait>>
DistributedMapController
+ connectedPlayers: ConnectedPlayers
+ playersPositionDetails: ConcurrentMap[Int, PlayerPositionDetails]
+ sendTrainerPosition(position: Coordinate): Unit
+ sendTrainerInBuilding(isInBuilding: Boolean): Unit
+ sendTrainerIsBusy(isBusy: Boolean): Unit
+ sendChallengeToTrainer(otherPlayerId: Int): Unit
+ playerLogout(): Unit
<<Scala Class>>
LaboratoryController
- capturedPokemonEmpty: Boolean
- initView(): Unit
<<Scala Abstract Class>>
BuildingController
- mapController: GameController
# buildingMap : BuildingMap
<<Scala Class>>
PokemonCenterController
- initView(): Unit

Figure 6: Game controllers Class Diagram

The above UML diagram shows the Controller classes used to control the game map.
`GameController` manages the behaviour of entities in the map (game map and buildings map). It provides methods to perform trainer's movements, interactions with other trainers, box, not-player characters, and manage the lifecycle of the game.
`GameControllerImpl` implements `GameController` and it is extended by `MapController` and `BuildingController`.
`MapController` is the controller for the main game map and it creates `DistributedMapController` that manages the interaction with the server, allowing the game to operate at a distributed level. It, also, manages all trainers in the map and the interactions between them.
`BuildingController`, instead, is the controller for buildings map and manages the control of the players in the buildings. `BuildingController` will be extended by every building in the game.
Unlike `MapController`, that manages the possibility to see and interact with other player and fight with them, `BuildingController` manages only the local player so you can not see other people inside the building.

Trainer States

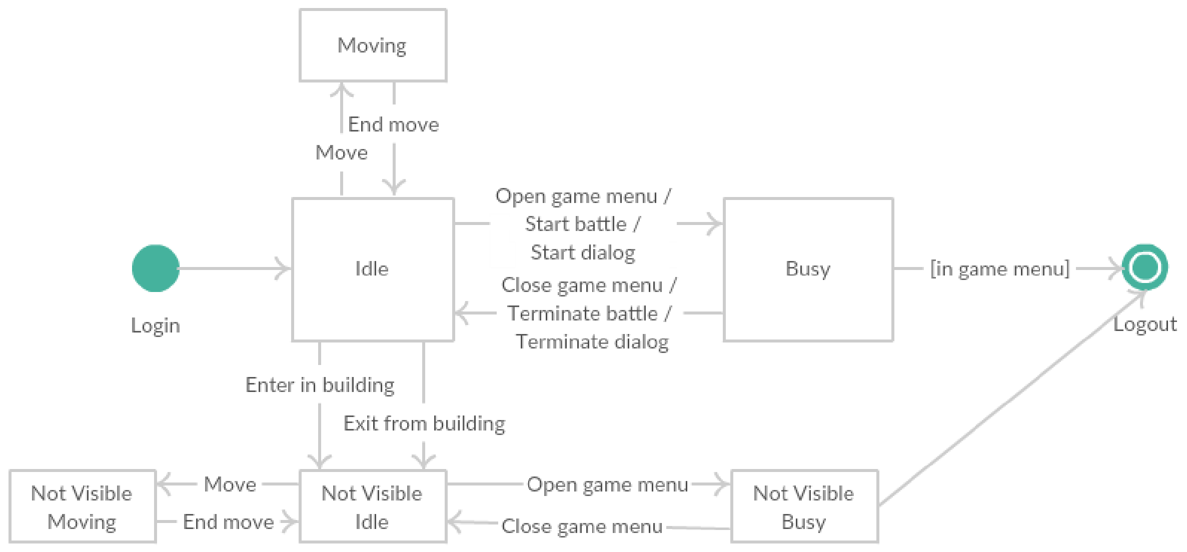


Figure 7: Trainer States Diagram

A trainer in the map can reach different states.

When a new game is started a trainer is in the idle state and moves to the moving state when a keyboard arrow key is pressed. A trainer is busy when he's fighting against a wild Pokémon or another trainer, when he's talking with another trainer, or when the user opens the game menu. In this state other trainers can't interact with him.

When entering inside a building a trainer becomes not visible for other players within the map. Starting from the not visible idle state he can be moved or become busy when a player opens the game menu.

When a trainer is in the busy state, reached by opening the game menu, or in the not visible busy state, inside a building, he can logs out.

Distributed elements

The software is composed by two main parts, client and server, which in order to communicate in a distributed manner use several classes called managers client-side and services server-side.

Clients and server keep the set of players that are currently connected to the game. They use a `ConnectedPlayers` class with a map that keeps this set. `Player` keeps information about a player connected to the game.

Server

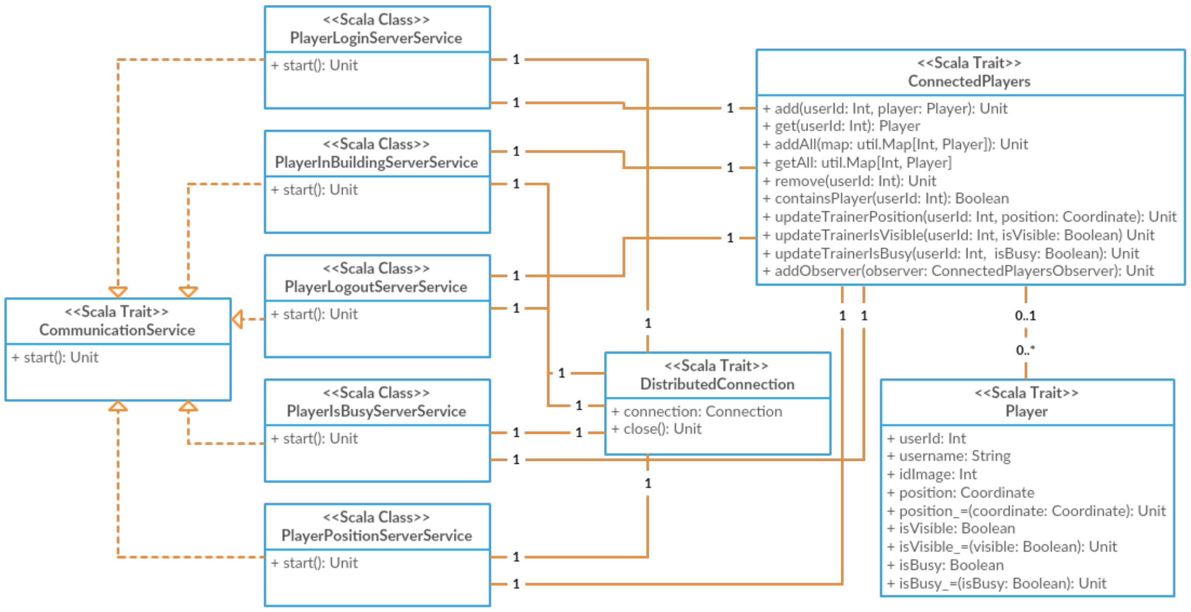
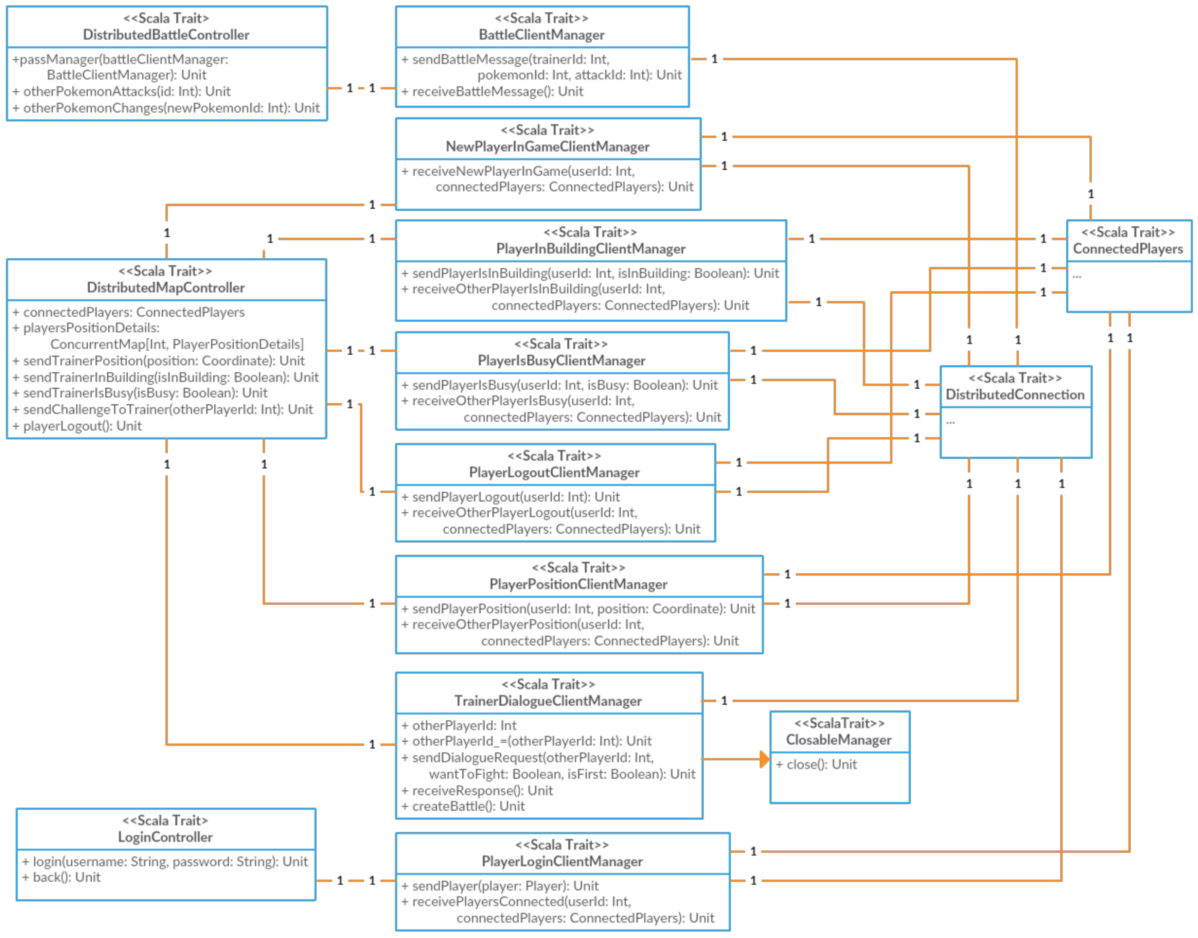

<<Scala Class>>
PlayerLoginServerService
+ start(): Unit
<<Scala Class>>
PlayerInBuildingServerService
+ start(): Unit
<<Scala Class>>
PlayerLogoutServerService
+ start(): Unit
<<Scala Class>>
PlayerIsBusyServerService
+ start(): Unit
<<Scala Class>>
PlayerPositionServerService
+ start(): Unit
<<Scala Trait>>
CommunicationService
+ start(): Unit
<<Scala Trait>>
ConnectedPlayers
+ add(userId: Int, player: Player): Unit
+ get(userId: Int): Player
+ addAll(map: util.Map[Int, Player]): Unit
+ getAll: util.Map[Int, Player]
+ remove(userId: Int): Unit
+ containsPlayer(userId: Int): Boolean
+ updateTrainerPosition(userId: Int, position: Coordinate): Unit
+ updateTrainerIsVisible(userId: Int, isVisible: Boolean) Unit
+ updateTrainerIsBusy(userId: Int, isBusy: Boolean): Unit
+ addObserver(observer: ConnectedPlayersObserver): Unit
<<Scala Trait>>
DistributedConnection
+ connection: Connection
+ close(): Unit
<<Scala Trait>>
Player
+ userId: Int
+ username: String
+ idImage: Int
+ position: Coordinate
+ position_=(coordinate: Coordinate): Unit
+ isVisible: Boolean
+ isVisible_=(visible: Boolean): Unit
+ isBusy: Boolean
+ isBusy_=(isBusy: Boolean): Unit
1
0..1
0..*

Figure 8: Server Classes Diagram

- `PlayerLoginServerService`: receives a message when a new player is connected to the game, updates the current connected players, sends to the new player all the connected players, and sends to all clients that a new player is connected;
- `PlayerLogoutServerService`: receives a message when a player is logged out from the game, update the current connected players and sends to all clients that someone as left the game;
- `PlayerPositionServerService`: receives a message when a player moved, updates the position of the player in the current connected players and sends to all clients that a player moved;
- `PlayerInBuildingServerService`: receives a message when a player entered/left a building, updates the related attributes of the player in the map of connected players and sends to all clients that a player entered/left a building;
- `PlayerIsBusyServerService`: receives a message when a player enters in a busy state, updates the related attributes of the player in the map of connected players and sends to all clients that a players is entered in a busy state.

Client



<<Scala Trait>>
DistributedBattleController
+passManager(battleClientManager: BattleClientManager): Unit
+ otherPokemonAttacks(id: Int): Unit
+ otherPokemonChanges(newPokemonId: Int): Unit
<<Scala Trait>>
BattleClientManager
+ sendBattleMessage(trainerId: Int, pokemonId: Int, attackId: Int): Unit
+ receiveBattleMessage(): Unit
<<Scala Trait>>
NewPlayerInGameClientManager
+ receiveNewPlayerInGame(userId: Int, connectedPlayers: ConnectedPlayers): Unit
<<Scala Trait>>
PlayerInBuildingClientManager
+ sendPlayerIsInBuilding(userId: Int, isInBuilding: Boolean): Unit
+ receiveOtherPlayerIsInBuilding(userId: Int, connectedPlayers: ConnectedPlayers): Unit
<<Scala Trait>>
ConnectedPlayers
...
<<Scala Trait>>
DistributedMapController
+ connectedPlayers: ConnectedPlayers
+ playersPositionDetails: ConcurrentMap[Int, PlayerPositionDetails]
+ sendTrainerPosition(position: Coordinate): Unit
+ sendTrainerInBuilding(isInBuilding: Boolean): Unit
+ sendTrainerIsBusy(isBusy: Boolean): Unit
+ sendChallengeToTrainer(otherPlayerId: Int): Unit
+ playerLogout(): Unit
<<Scala Trait>>
PlayerIsBusyClientManager
+ sendPlayerIsBusy(userId: Int, isBusy: Boolean): Unit
+ receiveOtherPlayerIsBusy(userId: Int, connectedPlayers: ConnectedPlayers): Unit
<<Scala Trait>>
DistributedConnection
...
<<Scala Trait>>
PlayerLogoutClientManager
+ sendPlayerLogout(userId: Int): Unit
+ receiveOtherPlayerLogout(userId: Int, connectedPlayers: ConnectedPlayers): Unit
<<Scala Trait>>
PlayerPositionClientManager
+ sendPlayerPosition(userId: Int, position: Coordinate): Unit
+ receiveOtherPlayerPosition(userId: Int, connectedPlayers: ConnectedPlayers): Unit
<<Scala Trait>>
TrainerDialogueClientManager
+ otherPlayerId: Int
+ otherPlayerId_=(otherPlayerId: Int): Unit
+ sendDialogueRequest(otherPlayerId: Int, wantToFight: Boolean, isFirst: Boolean): Unit
+ receiveResponse(): Unit
+ createBattle(): Unit
<<ScalaTrait>>
ClosableManager
+ close(): Unit
<<Scala Trait>>
LoginController
+ login(username: String, password: String): Unit
+ back(): Unit
<<Scala Trait>>
PlayerLoginClientManager
+ sendPlayer(player: Player): Unit
+ receivePlayersConnected(userId: Int, connectedPlayers: ConnectedPlayers): Unit
1

Figure 9: Client Classes Diagram

- `PlayerLoginClientManager`: sends a message when the player logs in to the game and receives a message with all the player present in the game updating the current connected players;
- `NewPlayerInGameClientManager`: receives a message when a player logs in and add the received player to the current connected players;
- `PlayerLogoutClientManager`: sends a message when the player logs out from the game and receives a message when another player logs out updating the current connected players;
- `PlayerPositionClientManager`: sends and receives messages about players' position, updating the position of the player in the current connected players;
- `PlayerInBuildingClientManager`: sends and receives messages about the players' visible state and updates the related attributes of the player in the map of connected players;
- `PlayerIsBusyClientManager`: sends and receives message about the players' busy state and updates the related attributes of the player in the map of connected players;
- `TrainerDialogClientManager`: manages the delivering and the receiving of messages related to the request of a new battle and the respective answer;

- `BattleClientManager`: sends and receives all the messages related to a battle against an other trainer.

Client-Server sequence diagrams

The graphs below show the interaction between client and server classes using RabbitMQ as message broker.

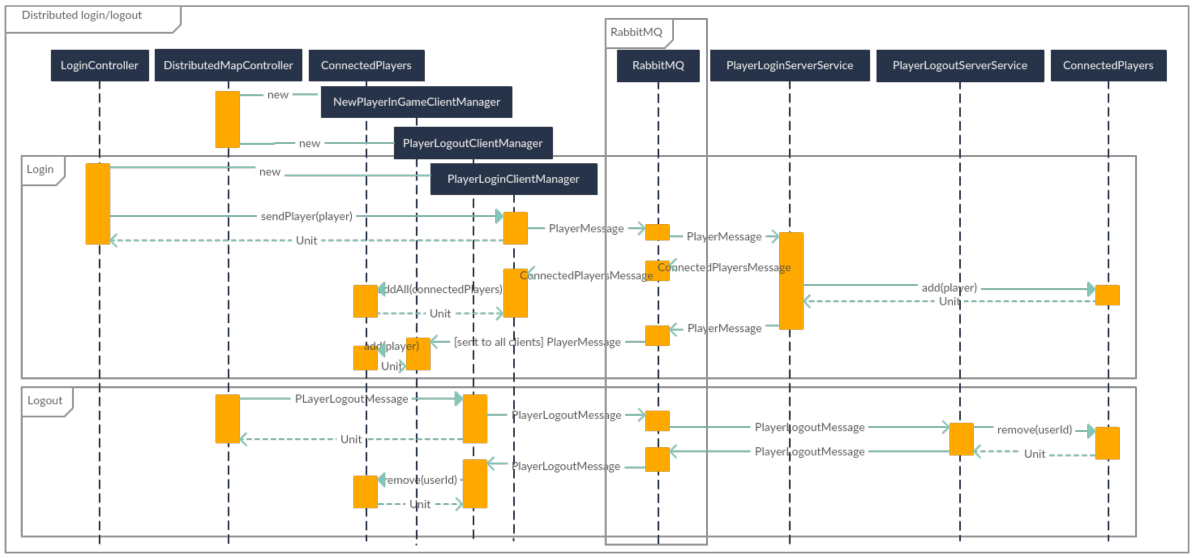


Figure 10: Distributed login/logout Sequence Diagram

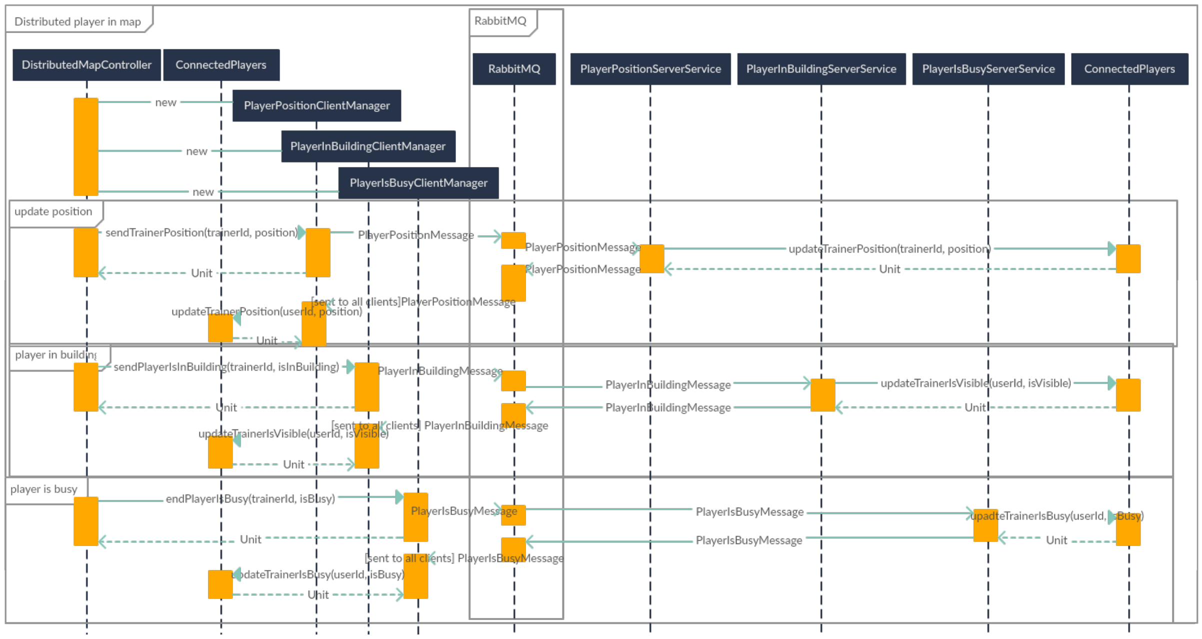

Distributed player in map
RabbitMQ
DistributedMapController
ConnectedPlayers
RabbitMQ
PlayerPositionServerService
PlayerInBuildingServerService
PlayerIsBusyServerService
ConnectedPlayers
new
PlayerPositionClientManager
new
PlayerInBuildingClientManager
new
PlayerIsBusyClientManager
update position
sendTrainerPosition(trainerId, position)
PlayerPositionMessage
PlayerPositionMessage
updateTrainerPosition(trainerId, position)
Unit
PlayerPositionMessage
Unit
[sent to all clients]PlayerPositionMessage
updateTrainerPosition(userId, position)
Unit
player in building
sendPlayerIsInBuilding(trainerId, isInBuilding)
PlayerInBuildingMessage
PlayerInBuildingMessage
updateTrainerIsVisible(userId, isVisible)
Unit
PlayerInBuildingMessage
Unit
[sent to all clients] PlayerInBuildingMessage
updateTrainerIsVisible(userId, isVisible)
Unit
player is busy
sendPlayerIsBusy(trainerId, isBusy)
PlayerIsBusyMessage
PlayerIsBusyMessage
updateTrainerIsBusy(userId, isBusy)
Unit
PlayerIsBusyMessage
Unit
[sent to all clients] PlayerIsBusyMessage
updateTrainerIsBusy(userId, isBusy)
Unit

Figure 11: Distributed player in map Sequence Diagram

Design Pattern

Entities

- Factory Method: used in `TrainerSprites` in order to create objects `Trainer1`, `Trainer2`, `Trainer3`, `Trainer4` by calling `apply()` method of `TrainerSprites`, using enumeration `Trainers` to distinguish them.
- Factory Method: used in `PokemonCharacter` in order to create objects `Bulbasaur`, `Charmander` and `Squirtle` by calling `apply()` method of `PokemonCharacter`, using enumeration `InitialPokemon` to distinguish them.
- Factory: used in `PokemonFactory` for creating each Pokémon entity. You will only need to indicate the type of Pokémon required (wild, trained or initial) and the eventual database id (in the case of trained Pokémon) and a Pokémon entity will be returned.

Map

- Template Method: used in `GameControllerImpl`, which implements common behavior of its subclasses, deferring some steps to them. This pattern is used in methods `start()`, `terminate()`, `pause()`, `resume()`, `moveTrainer()`, and `trainerInteract()`. `GameControllerImpl` is extended by `MapController` and `BuildingController` which implement methods `doStart()`, `doTerminate()`, `doPause()`, `doResume()`, `doMove()`, `doInteract()`.
- Template Method: used in `MovementImpl`. This pattern is used in method `walk()`. `MovementImpl` is extended by `MainTrainerMovement` and `OtherTrainerMovement`, which implement methods getter and setter of `currentTrainerSprite`, `trainerSprites()`, `updateCurrentX()`, `updateCurrentY()`, and `updateTrainerPosition()`.
- Template Method: used in `GamePanelImpl` in method `paintComponent()`. `GamePanelImpl` is extended by `MapPanel` and `BuildingPanel` which implement method `doPaint()`.

Distributed

- Builder: used in `PlayerImpl` defining default values in its constructor.
- Factory Method: used in `CommunicationService` in order to create objects `PlayerLoginServerService`, `PlayerLogoutServerService`, `PlayerInBuildingServerService`, `PlayerIsBusyServerService`, `PlayerPositionServerService`, using enumeration Service to distinguish them, using in `PokemonFactory` to create Pokémon instance.
- Observer: using `DistributedMapController` as observer and `ConnectedPlayers` as observable. `DistributedMapController` implements trait `ConnectedPlayersObserver`, and it is notified by `ConnectedPlayers` every time a player is added to or removed from `ConnectedPlayers`, or a player's position is updated.

Implementation

Mattia Ricci

- Local database: the local database contains any kind of information related to Pokémon (name, levels, possible attacks, evolution chain, etc...). This database should never be updated, but only needs to return the required information regarding the Pokémon. We therefore opted to handle it locally, even for a matter of speed. The class that manages all the queries to the database is `PokedexConnect`. I realized this class as an object because in this way the connection is instantiated only once, and every method use that. Considering that the methods of this class resembled everyone (was created the query's string, the query was executed and the method returned the result), I decided to create the method `executeTheQuery()` to reuse the same code. This method get in input the string with the query (prepared in each single method) and returns the result.

Thanks to this class you can know, taking in input the Pokémon's id: - his name;
- the possible attacks that the Pokémon can learn;
- if the Pokémon has to evolve when grows his level;
- his level's range (min and max level);
- his first evolution's stage;
- his basic experience at first level;
- his rarity;
Moreover, you can know the attack's name from the attack id and the list of Pokémons with a specific rarity.

- Pokemon: this is one of the main entities of the game, in fact the purpose of the game is to capture and train them as much as possible, to beat the other trainers during the battles. As for the code, I decided to use the factory pattern for creating each Pokémon entity (object `PokemonFactory`). In this way you will only need to indicate to this object the type of Pokémon required (wild, trained or initial) and the eventual database id (in the case of trained Pokémon) and a Pokémon entity will be returned. This creature has some final fields, such as id, name, attacks, level, experience, image, and has only one mutable variable, the life. If the Pokémon is trained, the values are retrieved from the remote database by the id, and assigned to each variable. In the event that the Pokémon required is a wild Pokémon the process is more articulated. First of all, considering the level of the trainer, has been implemented an algorithm to determine which Pokémon the trainer encountered (of all 151 possible) using a casual factor weighed on the rarity of the Pokémon. In all this, the level of the trainer is crucial because it belongs to some slots. As the slot grows, more and more Pokémons can be found. For example, at the first level, only about thirty of the Pokémons can be found on 151. Once the Pokémon is created, the Pokémon level must be generated. To do this the algorithm first considers the range of levels which the Pokémon belongs at (for example, minimum 1 and maximum 25). The level is then created within this range, with a casual factor weighted on the trainer level. In doing so, more and more deterministic the other Pokémon fields are created.

Francesco Dicara

- Remote Database: the remote database contains all the records of registered users and the class that manages all the queries to the database is `DBConnect`. When a new user signs in, a new record in the Users table (with id, name, surname, email, username, password, avatar image id) is added using the method `insertCredential()` and then a new record in the Trainers table (with the same id of the user, experience points and six attributes that indicates the six favourite Pokemon) using the method `createTrainer()`. When a user try to log in the method `checkCredentials()` is called to check if the user and password inserted are correct. `DBConnect` contains also methods to add a new Pokémon when the trainer catches it, to get the list of all captured Pokémons, to update or to get the list of the six favourite Pokémons. Every Pokémon captured has a record in the pokemon table with its id, trainer's id, life, level, experience and 4 attacks id. Also Pokémons that a trainer met are saved in the database in a table called pokemon-met where every record has the trainer id, the Pokémon id and the captured flag. This flag is used to know if a Pokémon has been captured also when the Pokémon evolves and it's no more in the pokemon table. Then there are methods to get the playing trainer's rank and to get the complete ranking of the trainers ordered by experience points.

- Trainer: this is one of the main entities of the game, the trainer is the avatar of the user on the map and during battles. The constructor takes in the attributes stored in the remote database (username, id image, experience points). A trainer has two main list of Pokémon: one with his six favourite Pokémons used during battles and one with all the captured Pokémons used at the box in the Pokémon Center. A trainer has also a `Pokedex` that contains all the ids of the Pokémons that the trainer has met during battles. A trainer has also methods to update these lists during the game (both in local variables and in the remote database).
The main info about a trainer can be seen through the class `TrainerPanel`. `PokedexPanel` shows the Pokémons that the trainer has met. Lastly `TeamPanel` shows the information about the six Pokémons that compose trainer's team.

- Player is busy management: this part of the game is composed by two main classes: `PlayerIsBusyClientManager` and `PlayerIsBusyServerService`. The first works on client side and second on server side. The role of these classes is to update `isBusy` flag for every Player so that all the playing users can be informed if the player they want to challenge is available to fight or not. A Player is considered busy when he is fighting (against a wild Pokemon or against an other trainer) or if he has the main menu open. `PlayerIsBusyClientManager` declares two different queues: one to send messages and one to wait messages from the server. When a message is received, `PlayerIsBusyClientManager` updates the player `isBusy` flag in `ConnectedPlayers`. `PlayerInBuildingServerService` waits for messages from clients and, every time it receives a `PlayerIsBusyMessage`, it changes `isBusy` flag in `ConnectedPlayers` in order to inform all clients if a player is busy or not. If a player is busy a dialogue panel appears to inform the user that tried to have a fight.

Mattia Ricci & Francesco Dicara

- Local Battle: In the `MapController` class in the method `randomPokemonAppearance()`, when a wild Pokémon is found, a new `BattleController` is created. As we already told previously, consequently are created Battle and `BattleRound` (Model) and `BattlePanel` (View). Now the control flow is in the View where the player can choose the following options:
  - Pokemon attack: in the view, the trainer decides which attack to use and consequently `myPokemonAttacks()` Controller method is called with the id attack in input; this method calls `myPokemonAttack()` method in the `BattleRound` class. This method updates wild Pokémon life, checks if it is dead and possibly updates remote database record calling the specific methods in `DBConnect` class. If the wild Pokémon is alive, the `BattleController` make the wild Pokémon attack calling the method `otherPokemonAttacks()`. In the event that the wild Pokémon kills trainer's one, the controller makes a new `BattleRound` start (this loop continues until the trainer has alive Pokémon in the bag). If the wild Pokémon kills all trainer's Pokémons, the game will be resumed in front of the Pokémon Center and all trainer's Pokémons will be recharged.
  - Change his Pokémon: when a trainer chooses to change his fighting Pokémon the Controller method `changePokemon()` is called. Here a new `PokemonChoicePanel` is shown and in this panel the trainer can view and choose one of his available Pokémons. The control flow comes back to the `BattleController` that creates a new `BattleRound` and a new Battle View, not before updating the remote database. Since this is considered as a trainer move, then the method `pokemonWildAttacksAfterTrainerChoice()` is called.
  - Throw a pokeball: a trainer can try to catch a Pokémon throwing one of his 3 available Pokeballs during the battle. The method that establish if the wild Pokémon will be caught is `pokemonIsCaptured()` in the Model. This method considers wild Pokémon life and experience and trainer's level with a little of randomness. If the Pokémon is captured `insertPokemonIntoDB()` of DBConnect is called and the method returns true. In case that the capture went bad, the method `pokemonWildAttacksAfterTrainerChoice()` is called.
  - Try to quit: if the trainer wants to quit the battle the method `trainerCanQuit()` is called and it returns true with a probability of 50%.

- Distributed Battle: differently from the previous battle, this type of fight can start when two trainers agree to challenge each other, as a result of an exchange of messages. The trainer who has advanced the initial request will have the right to make the first choice. In the distributed battle the trainer can only attack or change his Pokémon. In the event of the trainer chooses to attack the method `myPokemonAttacks()` is called and it will make the same Model methods start as in the local battle. In addition to that it will send a message to the other trainer calling the method `sendBattleMessage()` of the `BattleClientManager` class. When the other trainer receive this message, the method `otherPokemonAttacks()` is called and automatically the other trainer view will be unlocked so that he can make the next move. When one of the fighting Pokémons dies, a new round starts automatically with the next Pokémon available of the trainer that lose the previous round. Also the change of the Pokémon is managed with a message exchange. The battle ends only if all of the Pokémons of one of the trainer die. The game will be resumed with the loser trainer in front of the Pokémon Center and the other one in the previous position in the map.

- Battle View: `BattlePanel` contains all the code developed to show the graphics of a Pokémon battle. This panel continually updates thanks to the interaction with the controller, through which change the life of the Pokémon after each attack and eventually change round. It was difficult to realize, as each single element had a certain position inside the panel and even more complex was making it all proportional to screen resolution. A nice and innovative feature (at least for us) to realize was a graphic animation, especially the launch of a pokeball. When the coach decides to launch a pokeball, this appears in the bottom of the screen, approach the opponent's Pokémon and enclosing it inside. If the Pokémon has been captured the ball becomes red, otherwise it will reopen and the Pokémon will return to its position. To implement this, it was necessary to combine the use of the `paintComponent` and a timer, which automatically moves the image of the ball every 10 milliseconds.

- Battle message exchange: during a battle between two players, they communicate sending messages using RabbitMQ on the remote server. When a new distributed battle starts, a new `BattleClientManager` is instantiated. This class creates a new channel and declares two queues: the first is the queue of the other player where the messages will be sent, the other one is the queue where the player receives and consumes the messages from the other player. The instance of the `BattleClientManager` is accessible from the `DistributedBattleController` that first calls `receiveBattleMessage()` method in the `BattleClientManager` so that the message consumer starts to listen to on his queue. Every time it is the turn of one of the two players and they do an action a message is sent. The messages are represented by the class `BattleMessage` that contains three variables: the id of the player sending the message, the id of the Pokémon playing and the id of the attack chosen. When a player choose to attack the other Pokémon a message is sent and when is consumed by the other player the `otherPokemonAttacks()` method in the controller is called. If the message received has the Pokémon id but the attack id is equal to 0 then it means that the other trainer changed his Pokémon. Lastly, if the message has Pokémon id and attack id both equal to 0 it means that the battle is finished because the other trainer finished his available Pokémons.

Gianluca Grossi & Francesco Dicara

- Trainers' dialogue to start a battle: every trainer on his game has a `TrainerDialogueClientManager` that is instantiated in the `DistributedMapController`. Then `receiveResponse()` method is called and a message consumer starts waiting for a `DialogueMessage` from an other other. When a trainer on the map finds on his next step an other trainer, the first one can press space key on the keyboard to ask him to fight. If the other trainer is not busy, a new `WaitingTrainerPanel` appears and a `TrainerDialogueMessage` is sent to the other trainer. A `TrainerDialogueMessage` contains the sender's id and username, the receiver's id, a boolean that indicates if the sender wants to fight and a boolean that indicates if the sender is the trainer that asked first to fight. When a trainer receives a message with `wantToFight` and `isFirst` equal to true, a new panel appears where the trainer can answer to the challenge. If the answer is affirmative then a message with `wantTofight` equals to true is sent and in the `MapController` of both clients the method `createTrainersBattle()` is called. This method instantiates a `DistributedBattleController` and a `BattleClientManager`. If the other trainer answer that he do not want to fight, the first trainer receive the message and a new panel appears showing the negative answer.

Giulia Lucchi

- Initial Game Menu: when the application is started the `InitialMenuController` is created that manages the user's interaction with the initial game menu which is shown by `InitialMenuPanel`. `InitialMenuPanel` is composed by login, sign in and quit buttons. Sign in is used in order to create a new user of the game, `SignInPanel` and `SignInController` manage this by checking that each field (name, surname, username, password, etc) in `SignInPanel` are filled in the right way and by inserting the new user in the database. `LoginPanel` and `LoginController` are used to handle when an user logs in to the game. `LoginController` checks if the user has entered the right credentials by calling the `DBConnect` `checkCredential()` method and if so it creates a new game. `LoginController` obtains the user's trainer from the database, creates a new connection with RabbitMQ and interacts with the server in order to register the user and to receive the players present within the game. Finally, it starts the main game map.

(`InitialMenuPanel`, `LoginPanel` and `SignInPanel` are implemented with Mattia Ricci)

- Distributed Game Map: `DistributedMapController` manages the interaction with the server, allowing the game to operate at a distributed level. It manages all trainers in the map and the interactions between them. `DistributedMapControllerImpl` implements `DistributedMapController` and it is created when a `MapController` is initialized. This controller creates client managers and it deals with the server interaction by sending when the main trainer changes his position, when he enters/leaves buildings and when he is or is not busy. Moreover it is used to send a challenge request to another trainer and to send when the user logs out. These operations are performed through a method call of the dedicated client managers. When a user logs out to the game the controller closes the connection with RabbitMQ. `DistributedMapControllerImpl` uses a `ConcurrentMap`, which permits to handle the concurrent access to the map elements by different threads in a safe way, in order to store other trainers' position details who are inside the map that are then used by `MapPanel` for painting them in the game map. This `ConcurrentMap` is a map of couples $\langle userId, PlayerPositionDetails\rangle$, where `PlayerPositionDetails` keeps information about a player's position detail (current coordinates and sprite) who is connected to the game. `DistributedMapControllerImpl` is implemented as a `ConnectedPlayersObserver` in order to be notified, and therefore updating players position details, whenever `ConnectedPlayers` is updated when another player is added or removed to the current connected players, or when another trainer updates his position. In the latter case an `OtherTrainerMovement` is created for managing this trainer's movement in the map. `DistributedMapControllerImpl` uses a fixed thread pool executor (pool size = number of available processors + 1) to execute `OtherTrainerMovement walk()` method by a Scala `Future`.

- Players login client side: `PlayerLoginClientManager` is used to send a `PlayerMessage` to communicate to the server when the main player logs in to the game and to receive a `ConnectedPlayersMessage` from the server with all the players present in the game. It is created by `LoginControllerImpl` when a new game is stared. Messages are exchanged with the server using one queue to send the `PlayerMessage` and another queue to wait the `ConnectedPlayersMessage`. When the `ConnectedPlayersMessage` is received the manager adds all the players in the game to the `ConnectedPlayers`.
  `NewPlayerInGameClientManager` is used in `DistributedMapController` to add a new player to the current `ConnectedPlayers`. This manager receives a `PlayerMessage` when a new player logs in to the game.
  Messages exchanged are in the form of Json string and I used the Gson API to convert message objects into Json strings and vice versa.

- Players logout client side: `PlayerLogoutClientManager` is used to send a `PlayerLogoutMessage` to communicate to the server when the main player logs out to the game and to receive a `PlayerLogoutMessage` from the server when another player logs out to the game. Messages are exchanged with the server using one queue to send the message and another queue to wait the server messages. When a message is received the manager removes the player to the `ConnectedPlayers`.

- Players position client side: `PlayerPositionClientManager` is used to send a `PlayerPositionMessage` to communicate to the server the new main player position when the player moves and to receive a `PlayerPositionMessage` from the server when another player moves. Messages are exchanged with the server using one queue to send the message and another queue to wait the server messages. When a message is received the manager update the player position in the `ConnectedPlayers`.

- Players in building client and server side: `PlayerLogoutClientManager` is used to send a `PlayerInBuildingMessage` to communicate to the server that the main player enters/leaves a building and to receive a `PlayerInBuildingMessage` from the server when another player enters/leaves a building. Messages are exchanged with the server using one queue to send the message and another queue to wait the server messages. When a message is received the manager update the player visibility in the `ConnectedPlayers` (when a trainer enters a building it is not visible in the map). `PlayerInBuildingServerService` waits messages from clients and, when it receives a `PlayerInBuildingMessage`, it updates the player visibility in `ConnectedPlayers`. Finally, the service sends to all clients that a player enters/leaves a building.

- Audio: an audio is created in each controller that want to start a clip audio. `AudioImpl` implements `Audio` and makes use of `AudioStream` Java API. `Audio` provides different methods to handle an audio clip during the game (`play()`, `stop()`, `pause()` and `loop()`) that are called from the controllers during the game lifecycle.

Luca Polverelli

- Trainer's sprites: in the game there are 2 male trainers and 2 female trainers each of whom has its sprites. Trainer's sprites are represented by case classes `Trainer1`, `Trainer2`, `Trainer3`, `Trainer4` which implement `TrainerSprites` trait, and contain twelve different sprites for each trainer type. Sprites are represented by Sprite trait that is implemented by `Back1`, `Back2`, `BackS`, `Front1`, `Front2`, `FrontS`, `Right1`, `Right2`, `RightS`, `Left1`, `Left2`, `LeftS` case classes, one for each position that a trainer can assume. The sprites identified by the number 1 indicate sprites that trainers have during the first step of a movement, sprites identified by number 2 indicates sprites that trainers have during the second step of a movement, and those identified by S indicates sprites that trainers have when they are not moving. Each `Sprite` has a different image. `TrainerSprites` images are used by `MapPanel` and `BuildingPanel` when the trainer is painted.

- Json deserializers: messages exchanged by clients and server are in the form of Json string. The Gson API is used in order to convert message objects into Json strings and vice versa. When Gson API is used to convert complex objects, that are objects which fields are not primitive types, it needs to use custom deserializers which describe how to return an object from its Json string. I implemented deserializers for messages and objects that are exchanged:
  - `ConnectedPlayersDeserializer` converts `ConnectedPlayers` object;
  - `ConnectedPlayerMessageDeserializer` converts `ConnectedPlayerMessage` object;
  - `PlayerDeserializer` converts `Player` object;
  - `PlayerMessageDeserializer` converts `PlayerMessage` object;
  - `PlayerPositionMessageDeserializer` converts `PlayerPositionMessage` object.

- Distributed player's login server side: `PlayerLoginServerService` implements `ServerService`. It receives when a new player has connected to the game, sends to the new player all the connected players, and sends to all clients that a new player is connected. `PlayerLoginServerService` waits messages on the `PLAYERS_CONNECTED_CHANNEL_QUEUE` queue, when it receives a `PlayerMessage` on that queue it sends on the `PLAYERS_CONNECTED_CHANNEL_QUEUE+player.userId` queue the current `ConnectedPlayers` using a `ConnectedPlayersMessage`, where `player.userId` is the id of the player who sent the message. Then it adds the `PlayerMessage Player` into `ConnectedPlayers`. Finally it sends the `PlayerMessage` on the `NEW_PLAYER_EXCHANGE` exchange.

- Distributed player's logout server side: `PlayerLogoutServerService` implements `ServerService`. It receives when a player has logged out from the game, and sends to all clients that someone as left the game. `PlayerLogoutServerService` waits messages on the `PLAYER_LOGOUT_CHANNEL_QUEUE` queue, when it receives a `PlayerLogoutMessage` on that queue it removes the player from `ConnectedPlayers` whose id is in the message. Finally it sends the `PlayerLogoutMessage` on the `PLAYER_LOGOUT_EXCHANGE` exchange.

- Distributed player's movement server side: `PlayerPositionServerService` implements `ServerService`. It receives a message when a player moved, and sends to all clients that that player moved. `PlayerPositionServerService` waits messages in the `PLAYER_POSITION_CHANNEL_QUEUE` queue, when it receives a `PlayerPositionMessage` on that queue it updates the player's position in `ConnectedPlayers` whose id is in the message. Finally it sends the `PlayerPositionMessage` on the `PLAYER_POSITION_EXCHANGE` exchange.

Giulia Lucchi & Luca Polverelli

- Game Map: a game map is created by `MapController` calling `MapCreator.create()` method passing to it map dimensions and elements that need to be created as `MapElements`. `MapCreator` returns a `GameMap` to which are added elements by `addTile()` method. In order to populate the initial map of the game we have created `InitialTownElements`, which extends `MapElementsImpl`, which contains all the elements of the game map (buildings, tall grass, trees, lakes, etc..). `MapElements` has a map of $\langle (Int, Int), Tile \rangle$ where (Int,Int) is a `Tuple2` which represents the tile coordinates, and has some methods to add tiles to the map. `GameMap` implements `BasicMap` that provides a matrix of tiles. This matrix, implemented in `GameMap`, is initially filled with grass and it is then populate by the `InitialTownElements` tiles. Elements that compose the map are `Tile`, `Building` and `CompositeElement`. Each tile has specific attributes which are height, width, walkable, wild and image. `Building` extends `Tile` with two more attributes that are `doorCoordinate` and `topLeftCoordinate`. A `CompositeElement` is composed of tile of an element with margins. The tile position inside the map is specified by `Coordinate`, which as two attributes, `x` and `y`, and we defined two `implicits` in order to convert from `Coordinate` to `Tuple2` and vice versa.
  We implemented `GameController` in order to manage all the events that a player can trigger on the map, as moving the trainer, interacting with other players, opening or closing the game menu. When a player presses an arrow key on the keyboard makes its trainer move, this is possible thank to `GameKeyListener` which calls the `GameController` method `moveTrainer()` which calls the abstract method `doMove()` implemented in `MapController`. `doMove()` checks the tile returned by `nextTrainerPosition()`, if it is a position of a building door the trainer will go into the building, else if it is a position of a not walkable tile the trainer will not move, otherwise the `walk()` method will be called and if the tile is wild the `randomPokemonAppearance()` method will be called too. `randomPokemonAppearance()` checks if a wild `Pokmon` has been met using a random number from 0 to 9, if the number is greater or equals to 8 a new `BattleController` is created. We used a `Semaphore` in order to wait the end of the trainer movement before creating the `BattleController`. When `walk()` is called it creates a new`MainTrainerMovement` and it uses a single thread executor in order to execute the `MainTrainerMovement` method `walk()` by a Scala `Future`. `MovementImpl` is an abstract class that implements `Movement` and is extended by `MainTrainerMovement` and `OtherTrainerMovement`. This class deals with executing the trainer's movement animation inside the map. `MovementImpl` `walk()` method splits the trainer's movement, from a coordinate to another, in 4 parts updating trainer's position and sprite which are drawn by `MapPanel`.
  `GameController` has methods to handle the trainer's position update when he loses a battle and during the first login of the player. Moreover, it has `start()`, `terminate()`, `pause()`, `resume()` method to handle the game lifecycle when the player starts/terminates a battle, opens/closes the game menu, enters/leaves a building, logs out from the game calling `logout()`.
  The private class `GameControllerAgent` inside `GameController` is a thread which calls the current panel `repaint()` method in order to repaint the view. It is started everytime `start()` and `resume()` method of `GameController` are called, and stopped when `pause()` and `terminate()` are called.

- Game Map View: `MapPanel` shows the map of the game with all the elements inside it. `MapPanel` is created by `ViewImpl` when `MapController` is created. `MapPanel` extends `GamePanelImpl` which provides methods `updateCurrentX()` and `updateCurrentY()` to update a dynamic trainer on the map. `MapPanel` deals with drawing the game map with tiles, buildings, other trainers and the main trainer. The main trainer is always drawn to the center of the view. The map area that is drawn is as wide as the view plus an offset of 2 tiles on each side in order to reduce the weight of the elements to be painted. `GameControllerAgent` calls the method `repaint()` every GAME_REFRESH_TIME in order to refresh the map. We used `GamePanelImpl` methods `paintComponent()`, `updateCurrentX()`, `updateCurrentY()` as synchronized so that the current x and y coordinated can not be updated while the map is being painted.

- Connection with RabbitMQ: `DistributedConnection` creates and closes a client-server connection with **RabbitMQ** by setting the remote host address, port, username and password.

- Connected players: `DistributedMapController` and `ServerMain` maintain both a `ConnectedPlayers`, that keeps the set of players that are currently connected to the game, which are updated in `ClientManager` and `ServerServices` classes. `ConnectedPlayers` class uses a `ConcurrentMap` in order to keep this set, which permits to handle the concurrent access to the map elements by different threads in a safe way. This `ConcurrentMap` is a map of couples $\langle userId, Player \rangle$, where `Player` keeps information about a player connected to the game (`userId`, `username`, `idImage`, `position`, `isVisible`, `isBusy`). `ConnectedPlayers` has some methods to update its player's map, moreover keeps a list of `ConnectedPlayersObservers` which are notified whenever this map is updated.

Gianluca Grossi

- `BuildingMap`: Unlike the map, in the buildings the trainer moves while the map remains stationary. Also building map is managed by a set of tile. At first every tile is set as walkable, subsequently, tiles not walkable and static characters are set above them in the appropriate points (`Box`, `StaticCharacter`, etc...).
  In `BuildingMap` is not possible visualize the other players position. Consequently the other players in the `GameMap` are not able to see the local player (the variable `player.isVisible` will be set to false).
  The inner map of any building extends the `BuildingMap` trait. In our case there are two buildings and, accordingly, two classes that implement `BuildingMap`: `PokemonCenterMap` and `LaboratoryMap`. These two classes differ for image, entry coordinate, dimensions, etc... as well as elements they have.

  - `PokemonCenterMap`: it contains a `Tile` of type `Box`. When a player interacts with `Box`, it will be shown a `BoxPanel` in which is possible change Pokémons from own favourite team (`trainer.favouritePokemons()`, i.e. Pokémon team momentarily chosen for battles) with the remaining captured during the game (`trainer.capturedPokemons()` without `trainer.favouritePokemons()`). For convenience, in the box will be possible visualize the level and the image of each Pokémon, furthermore by clicking on a Pokémon it will be shown `PokemonPanel`: a panel that allows to show all attributes of a single Pokémon (life, experience points, moves, etc...). At last, after making the changes, the player can save those changes and return to the game.
    Moreover, in the `PokemonCenterMap` a `Doctor` is present (`StaticCharacter`), who is able to heal trainer's Pokémons when the trainer interacts to her.
  - `LaboratoryMap`: when a player logs in the game for the first time he has no Pokémons. So he is spawned inside the laboratory where he can not leave until he does not choose a starter Pokémon from three available `PokemonCharacter`.
    `PokemonCharacter` is a class that extends `StaticCharacter` (described below) but it also contains a `PokemonWithLife` variable. This `PokemonWithLife` are Bulbasaur, Charmander and Squirtle and they will have random attributes values. The trainer is able to see the attributes of each of this three Pokémons and he can choose one of them. Once he made his choice, that Pokémon joins the trainer's team, and the remaining two Pokémons will be not available anymore. The `StaticCharacter` in `LaboratoryMap` is `Oak` or `OakAfterChoice`: they have got a different dialogue.

- `StaticCharacter`: it's a static character in the `BuildingMap` which you can interact with.
  The characteristic part of a `StaticCharacter` is the dialogue. The dialogue is a `List` of strings that represents the dialogue provided by the static character when the trainer interacts with him. The dialogue will be shown in a specific panel (`DialoguePanel`) and it will be possible switch from one string to the next one. It is showed at the bottom of the screen (`BorderLayout.SOUTH`) and it has the view focus until its closure: during the dialogue the trainer is set as busy (`player.isBusy=false`). Every static character has a different dialogue from the others, but you can split out two typologies of dialogue:

  - Normal dialogue: a dialogue where the static character provides tips or information to the trainer. For example the interaction with `Oak` shows a `ClassicDialoguePanel`) that provides the trainer with information about he must make.

  - Dialogue with answer by choice: a dialogue that provides the trainer with the possibility of select two of more answer that will affect the gaming experience. For example `DoctorDialoguePanel` gives the trainer the choice about heal his Pokémons. If so the trainer team Pokémons life (`trainer.favouritePokemons`) is maxed. Otherwise nothing happens.

- Game menu: At any time, by pressing `ESC` key, you can open the game menu (`GameMenuPanel`), which is shown on the right side of the view.
  When opening the menu, `GameControllerAgent` is stopped and `player.isBusy` is set `true`.
  The game menu has these sections:

  - Pokedex: it allows to see all Pokémons that are met and caught by the player.
    The method `trainer().pokedex().pokedex()` allows to get a list of Pokémon id of all met Pokémon by the trainer.
    The method `trainer().capturedPokemonId`, instead, allows to get a list of Pokémon id of all caught Pokémon by the trainer.
    If a Pokémon is not met or caught it is represented by `???`.
  - Team: it allows to show all Pokémons that the trainer is carrying and their attributes (life, attack moves, experience points, level, etc...).
    A trainer can carry up to 6 Pokémons at a time and can change them into the Box in the Pokémon Center;
  - Trainer: it allows to visualize all trainer attributes (level, experience points and ranking position compared to all other trainers);
  - Ranking: it allows to show the trainers rank. The rank is based on trainer's level and experience point. This data are stored into and retrieved from the remote database (you can get the trainer position by the method `DBConnect.getTrainerRank()` passing the trainer id as a parameter);
  - Keyboard: it allows to show game controls;
  - Logout: it allows to perform the logout from the game. It calls `GameController logout()` and shows the `InitialMenuPanel`;
  - Exit: it closes the `GameMenuPanel` and resumes the game.

Common parts

- `View`, `ViewImpl`: `ViewImpl` is the class in view part that implements `View` interface. Thanks to this class it is possible to connect controllers and view to change the panel in the game frame every time it is necessary.

- `GameKeyListener`: this class extends Java `KeyListener` and was created to manage the different keys that a user can press during the game, particularly moving in the map.

- `Settings`: it contains all the constant values and Strings in the game (like image and audio's path). Here every variable is a Scala lazy val to save memory.

- Test: each element of the group dealt with test the classes that are created.

Retrospective

During the first meeting we have prepared the project build, established main project requirements and the basic architecture. In every sprint we decided which aspects to develop and who has to realize them, adopting an Agile development style. We exchanged opinions to adopt the best implementation choice about every significant aspect of design. We have organized every sprint to get tangible results in order to show project developments to the stakeholders. For every sprint we created a more detailed sprint backlog, following the directives of the initial product backlog, both opportunely updated during the development.

In general, we have slightly exceeded the 100-hour thresholds in order to fully conclude all the features of the game. We did not encounter any particular difficulties that interrupted the development of the project, which was fairly straightforward. We are fully satisfied with the final product, perhaps because we have been able to partially reproduce a game that has always been passionate all the group members. To get a general feedback, several of our friends tried the game, feeling satisfied. Since the game is interesting we sketched out some additional features:

- Pokemon exchange between trainers;
- extension of the map;
- Pokemon addition of successive generations.