German University in Cairo
Media Engineering and Technology
Prof. Dr. Slim Abdennadher

Computer Programming Lab, *Spring 2016*
Dragon Ball Adventures: **Milestone 2**

Deadline: Mon 28.3.2016 @23:59

This milestone is a further *exercise* on the concepts of **object oriented programming (OOP)** and putting your code together. The following sections describe the requirements of the milestone. Refer to the (**Game Description**) document for more details about the rules.

By the **end of this milestone**, you should have:

- Completed your implementation of the game engine
- Handled most of the special cases and scenarios in the game

Please note that you are not allowed to change the hierarchy provided in milestone 1. You should also conform to the method signatures provided in this milestone. However, you are free to add more helper methods and instance variables. You should decide about the correct access modifier for all methods. All class attributes should be private with the appropriate access modifiers and naming conventions for the getters and setters as needed.

Any compile error in the public or private tests means you have an unimplemented class, method or attribute. You need to fix that as it would mean getting a zero in the tests on evaluator. It is your responsibility to ensure that you have no compile errors in the tests. In case of compile errors you will see a bold `error` word in the trace on evaluator.in followed by the cause of the compile error.

The model answer for M1 is available on the met website. It is recommended to use this version. A full grade in M1 doesn't guarantee a 100 percent correct code, it just indicates that you completed all the requirements successfully :)

1 Creation of New Classes and Datatypes

Add the following to your code.

1.1 GameState

Name : `GameState`

Package : `dragonball.model.game`

Type : Enum

Description : Enum represeting the different game states. Its values: WORLD, BATTLE and DRAGON.

1.2 DragonWishType

Name : `DragonWishType`

Package : `dragonball.model.dragon`

Type : Enum

Description : Enum represeting the different types of dragon wishes. Its values: SENZU_BEANS, ABILITY_POINTS, SUPER_ATTACK, ULTIMATE_ATTACK.

1.3 BattleEventType

Name : `BattleEventType`

Package : `dragonball.model.battle`

Type : Enum

Description : Enum represeting the different battle events. Its values: STARTED, ENDED, NEW_TURN, ATTACK, BLOCK and USE.

1.4 Game

Add the following attribute to the `Game` class. The attribute is READ only:

`GameState state`: Indicates the current state of the game.

1.5 Battle

Rename `currentOpponent` to `attacker`. Remember to rename the corresponding getter method. Note, that in `Eclipse` you can rename a variable in the whole project by using the refactor option (see http://goo.gl/zUbI7G). Add the following attributes to the `Battle` class. Both attributes are READ only:

1. `boolean meBlocking`: Indicates whether the active fighter is blocking i.e. true if the active fighter is blocking.
2. `boolean foeBlocking`: Indicates whether the foe fighter is blocking i.e. true if the foe fighter is blocking.

1.6 BattleEvent

Name : `BattleEvent`

Package : `dragonball.model.battle`

Type : Class

Description : Class representing a single battle event. It should extend the predefined `EventObject` belonging to Java.

1.6.1 Attributes

All the class attributes are READ only.

1. **BattleEventType type**: Specifies the type of the current battle event which is fired by the `Battle` class. The `START` event is fired whenever the battle starts while the `END` is fired upon the winning of any battle party. Each of the remaining events is fired in the end of the method in the `Battle` class with the same name as the event.
2. **BattleOpponent currentOpponent**: The current opponent in the battle.
3. **BattleOpponent winner**: The winner of the battle.
4. **Attack attack**: The performed attack within the current battle event. It denotes the attack that caused the ATTACK event to be fired. It is an extra piece of information in the BattleEvent object and is changed each time the ATTACK event is fired to reflect the attack that was done.
5. **Collectible collectible**: The current collectible (senzu bean) used by the fighter within this battle event.

1.6.2 Constructors

1. `public BattleEvent(Battle battle, BattleEventType type)`: Base constructor that initializes a `BattleEvent` which is used by the other `BattleEvent` constructors. It should use the constructor of the superclass.
2. `public BattleEvent(Battle battle, BattleEventType type, BattleOpponent winner)`: Used to set winner in case of ENDED event.
3. `public BattleEvent(Battle battle, BattleEventType type, Attack attack)`: Used to set attack in case of ATTACK event.
4. `public BattleEvent(Battle battle, BattleEventType type, Collectible collectible)`: Used to set collectible in case of USE event.

1.7 DragonWish

Name : `DragonWish`

Package : `dragonball.model.dragon`

Type : Class

Description : Class representing a dragon wish. It should extend the predefined `EventObject` belonging to Java.

1.7.1 Attributes

All the class attributes are READ only.

1. `DragonWishType type`: Specifies the type of the current dragon wish.
2. `int senzuBeans`: Specifies the number of senzu beans granted by the wish.
3. `int abilityPoints`: Specifies the number of ability points granted by the wish.
4. `SuperAttack superAttack`: Specifies the super attack granted by the wish.
5. `UltimateAttack ultimateAttack`: Specifies the ultimate attack granted by the wish.

1.7.2 Constructors

1. `DragonWish(Dragon dragon, DragonWishType type)`: Base constructor that initializes a `DragonWish` which is used by other `DragonWish` constructors. It should use the constructor of the superclass.
2. `DragonWish(Dragon dragon, DragonWishType type, int senzuBeansOrAbilityPoints)`: Used to initialize senzu beans or ability points depending on whether `type` is SENZU_BEANS or ABILITY_POINTS.
3. `DragonWish(Dragon dragon, DragonWishType type, SuperAttack superAttack)`: Used to initialize the super attack in case the wish type is SUPER_ATTACK.
4. `DragonWish(Dragon dragon, DragonWishType type, UltimateAttack ultimateAttack)`: Used to initialize the ultimate attack in case the wish type is ULTIMATE_ATTACK.

1.8 GameListener

Name : `GameListener`

Package : `dragonball.model.game`

Type : Interface

Description : Interface containing the methods needed for the game listeners. This interface does not play any rule in this milestone. It listens for events that will later be needed for the GUI and for integrating the whole game play.

1.8.1 Methods

1. `void onDragonCalled(Dragon dragon)`: After choosing a random `dragon`, the method is notified by the `Game` that this `dragon` was called.
2. `void onCollectibleFound(Collectible collectible)`: The method is notified by the `Game` that a `collectible` was found.
3. `void onBattleEvent(BattleEvent e)`: The method is notified by the `Game` when a new battle event occurs.

1.9 PlayerListener

Name : `PlayerListener`

Package : `dragonball.model.player`

Type : Interface

Description : Interface containing the methods needed for the player listeners.

1.9.1 Methods

1. `void onDragonCalled()`: In the method's implementation in the `Game` class a random dragon should be chosen from the list of dragons. The seven used dragon balls should then be removed from the player. Think about updating the `GameState`.
2. `void onWishChosen(DragonWish wish)`: Think about updating the `GameState`.

1.10 WorldListener

Name : `WorldListener`

Package : `dragonball.model.world`

Type : Interface

Description : Interface containing the methods needed for the world listeners.

1.10.1 Methods

1. `void onFoeEncountered(NonPlayableFighter foe)`: In the method's implementation in the `Game` class create a new `Battle` object and add the `Game` to its listeners. The method `start()` of the battle should then be called. You should also make sure to replace the `FoeCell` with an `EmptyCell`. Note, that the active fighter has the first turn in the battle.
2. `void onCollectibleFound(Collectible collectible)`: In the method's implementation in the `Game` class adjust the player's senzu beans and dragon balls. Also, whenever the player has gathered 7 dragon balls invoke `callDragon` on the player.

1.11 CellListener

Name : `CellListener`

Package : `dragonball.model.cell`

Type : Interface

Description : Interface containing the methods needed for the cell listeners.

1.11.1 Methods

1. `void onFoeEncountered(NonPlayableFighter foe)`: This event should be forwarded to the `World`.
2. `void onCollectibleFound(Collectible collectible)`: This event should be forwarded to the `World`. You should also make sure to replace the `CollectibleCell` with an `EmptyCell`.

1.12 BattleListener

Name : `BattleListener`

Package : `dragonball.model.battle`

Type : Interface

Description : Interface containing the methods needed for the battle listeners.

1.12.1 Methods

`void onBattleEvent(BattleEvent e)`: In the method's implementation in the `Game` class you should handle different Battle events, including when the battle ends i.e. if the fighter won or lost, according to game description). Think about updating the `GameState`.

Game Logic

Event Handling Tips

Please note that whenever a class should fire events handled by a specific listener, you need to consider the following:

Firing Class

1. An instance of the listener should be added to the class with the appropriate getters and setters.
2. The class should call the listener-side method that is responsible for listening to this specific event, usually starting with the "`on`" keyword.
3. It should be checked that the listener is not null before using it.

Listening Class

1. The class implementing the listener is responsible for adding itself as a listener to the corresponding events.
2. The class should implement the methods in the corresponding listener interface.

Figure 1 shows the interaction of listeners used for the game and their hierarchy.

You should add the following methods to the previously built classes to add functionality to your game hierarchy and finalize the game engine. To get a better idea of how some of the main methods interact with each other check the `MethodsHierarchy.pdf` document posted on the met website.

2 Player

1. Should fire events handled by `PlayerListener`.
2. `int getMaxFighterLevel()`: Returns the highest level among the Player's fighters.
3. `void callDragon()`: Notifies listeners that the dragon is called.
4. `void chooseWish(DragonWish wish)`: It will be called whenever thee player chooses a wish from the dragon's wishes and it should perform the action of granting the player's wish according to the player's `wish`. Recall that each dragon can grant special super or ultimate attacks or a specific number of senzu beans or ability points. The chosen wish is one of the following:
   (a) senzu beans
   (b) ability points
   (c) new random super attack
   (d) new random ultimate attack

   The method should also notify the listener that a wish has been made.
5. `void createFighter(char race, String name)`: Creates a fighter belonging to the player with the specified `race` and `name`. Race can be one of: E, S, N, F and M representing Earthling, Saiyan, Namekian, Frieza and Majin, respectively. If this is the first fighter the player creates, set it as the active fighter.
6. `void upgradeFighter(PlayableFighter fighter, char fighterAttribute)`: Uses one ability point to upgrade the chosen `fighterAttribute` of the `fighter` and adds points or bars to the attribute as described in the game description. `fighterAttribute` is one of H, B, P, K, S representing the max health points, blast damage, physical damage, max ki and max stamina, respectively.

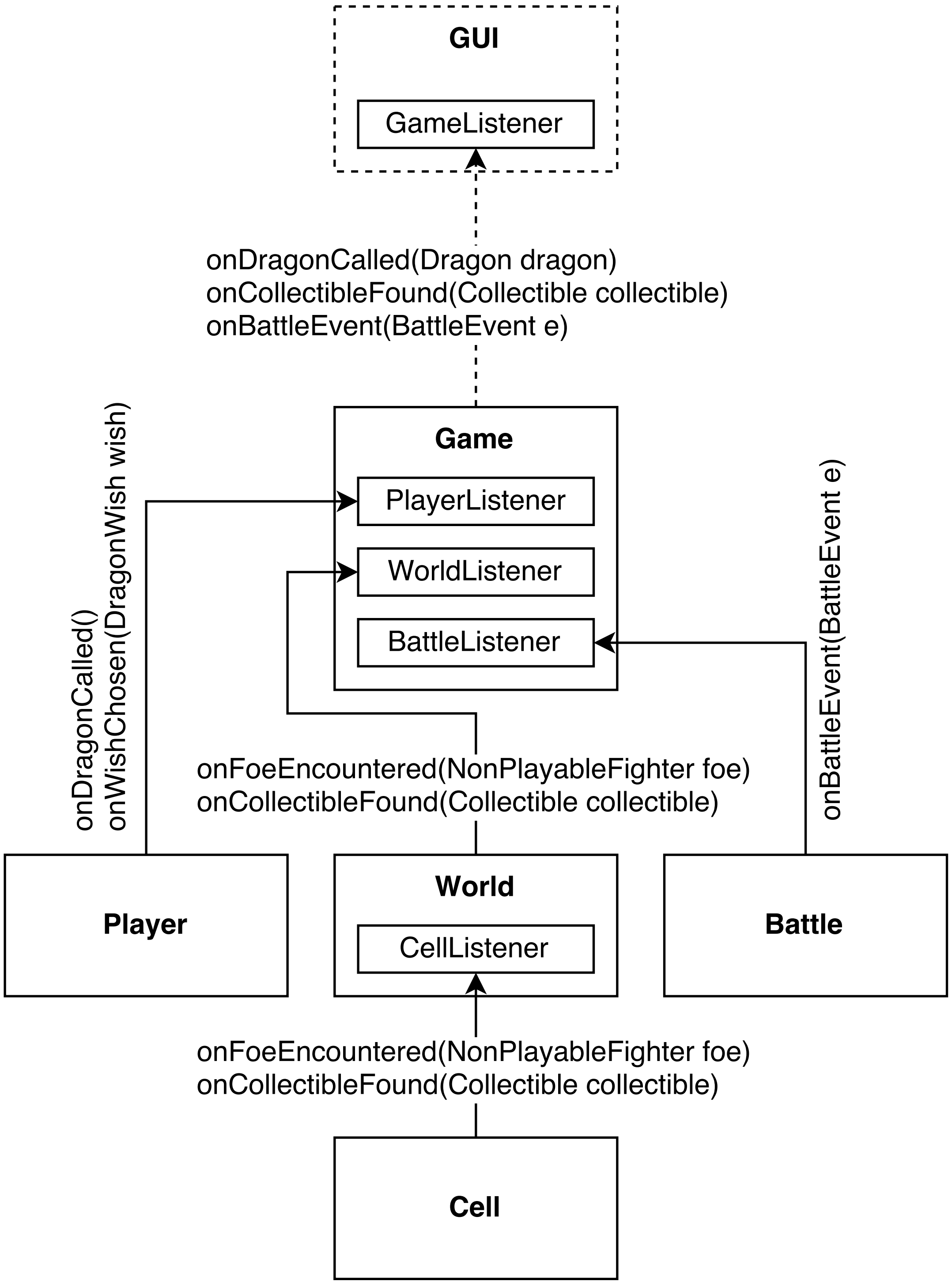
GUI
GameListener
onDragonCalled(Dragon dragon)
onCollectibleFound(Collectible collectible)
onBattleEvent(BattleEvent e)
Game
PlayerListener
WorldListener
BattleListener
onDragonCalled()
onWishChosen(DragonWish wish)
onBattleEvent(BattleEvent e)
onFoeEncountered(NonPlayableFighter foe)
onCollectibleFound(Collectible collectible)
Player
World
CellListener
Battle
onFoeEncountered(NonPlayableFighter foe)
onCollectibleFound(Collectible collectible)
Cell

Figure 1: Game Listeners Hierarchy

7. `void assignAttack(PlayableFighter fighter, SuperAttack newAttack, SuperAttack oldAttack)`: Assigns to the fighter a new super attack by calling the appropriate methods in the `PlayableFighter` class. If `oldAttack` is not null, then it should be replaced with `newAttack` from fighter's list of super attacks. If `oldAttack` is null, then it's assigning a new attack without intending to replace an old one. The attack will not be added unless it is ensured that the list of fighter attacks does not already contain the maximum number of attacks allowed.

8. `void assignAttack(PlayableFighter fighter, UltimateAttack newAttack, UltimateAttack oldAttack)`: Assigns to the fighter a new ultimate attack by calling the appropriate methods in the `PlayableFighter` class. If `oldAttack` is not null, then it should be replaced with `newAttack` from fighter's list of ultimate attacks. If `oldAttack` is null, then it's assigning a new attack without intending to replace an old one. The attack will not be added unless it is ensured that the list of fighter attacks does not already contain the maximum number of attacks allowed.

3 BattleOpponent Interface

1. `void onAttackerTurn()`: Performs the actions that are repeated in the attacker's own turn only i.e. actions that repeat in the turn of the battle member that's currently active.
2. `void onDefenderTurn()`:Performs the actions that are repeated in the defender's turn only i.e. actions that repeat in the turn of the battle member that's currently inactive.

4 Fighter

1. Think about the upper and lower bound of the health points, Ki and stamina while setting them. They have to be between 0 and the maximum value. In case of any value less that 0 it should be set to 0. In case of any number greater than the maximum it should be set to the maximum.

4.1 PlayableFighter

Think about how to handle the setter of the XP based on the restrictions provided in the game description. As long as the XP is more than the target XP, keep on leveling up. On each level up, increment target XP by 20, ability points by 2 and reset fighter's XP to 0.

4.1.1 Earthling

Think about if/how you will override the `void onAttackerTurn()` and `void onDefenderTurn()` methods for Earthlings. Recall that Earthlings get one Stamina bar each turn and get one Ki bar on their own turn.

4.1.2 Frieza

Think about if/how you will override the `void onAttackerTurn()` and `void onDefenderTurn()` methods for Frieza's Race. Recall that Frieza's Race get one Stamina bar each turn.

4.1.3 Majin

Think about if/how you will override the `void onAttackerTurn()` and `void onDefenderTurn()` methods for Majins. Recall that Majins get one Stamina bar on their opponent's turn.

4.1.4 Namekian

Think about if/how you will override the `void onAttackerTurn()` and `void onDefenderTurn()` methods for Namekian's. Recall that Namekian get one Stamina bar and 50 health points each turn.

4.1.5 Saiyan

Think about if/how you will override the `void onAttackerTurn()` and `void onDefenderTurn()` methods for Saiyans. Recall that Saiyans get one Stamina bar each turn. If transformed they also lose one Ki bar each turn. In case the Ki reached zero, their stamina also becomes zero and their transformation state ends (meaning they become no longer transformed).

4.2 NonPlayableFighter

Think about if/how you will override the `void onAttackerTurn()` method for non-playable fighters. Recall that non-playable fighters get one stamina bar each turn.

5 Cell

1. Should fire events handled by `CellListener`.
2. `void onStep()`: Handles the actions required upon moving one step within the map. Think whether this method should be implemented here or in the subclasses.

5.1 EmptyCell

Think about the needed functionality of the `void onStep()` method in case of an empty cell.

5.2 FoeCell

Think about the needed functionality of the `void onStep()` method in case of a cell containing a foe. The method should call the method for notifying the appropriate listeners.

5.3 CollectibleCell

Think about the needed functionality of the `void onStep()` method in case of a cell containing a collectible. The method should call the method for notifying the appropriate listeners.

6 World

1. Should fire events handled by `WorldListener`. The `World` class can be seen as an intermediate class that forwards events from `Cell` objects to the `Game`.
2. The class should implement the `CellListener` interface.
3. `void resetPlayerPosition()`: Resets the players position to starting cell (9, 9).
4. `void moveUp()`: Moves the player up by one step. Handle the case where this position is outside the map boundaries.
5. `void moveDown()`: Moves the player down by one step. Handle the case where this position is outside the map boundaries.
6. `void moveRight()`: Moves the player right by one step. Handle the case where this position is outside the map boundaries.
7. `void moveLeft()`: Moves the player left by one step. Handle the case where this position is outside the map boundaries.

7 Attack

1. `int getAppliedDamage(BattleOpponent attacker)`: Calculates the damage applied on the defender (attacked fighter) by the `attacker`. You should take into consideration the physical and blast damage of the `attacker`. The applied damage gets deducted from the health points of the defender. Think whether this method should be implemented here or in the subclasses.
2. `void onUse(BattleOpponent attacker, BattleOpponent defender, boolean defenderBlocking)`: Performs the action of using the current attack and all the necessary restrictions and considerations based on the game description. Think about which part of the functionality of this method to implement here and which to implement in the subclasses. In case of a transformed Saiyan, the damage should be increased by 25 percent. In case the `defender` is blocking the attack, then for every 100 damage points he/she will lose 1 stamina bar in order to block these 100 points until the current stamina is zero. In this case the `defender` will lose health equal to the unblocked amount.

7.1 Physical Attack

Think about the needed functionality of the `void onUse(BattleOpponent attacker, BattleOpponent defender, boolean defenderBlocking)` method for physical attacks. Recall that physical attacks do not require any Ki bars but charges one Ki bar to the user instead. The method should then rely on its counterpart in the super class to perform the attack.

7.2 Super Attack

Think about the needed functionality of the `void onUse(BattleOpponent attacker, BattleOpponent defender, boolean defenderBlocking)` method for super attacks. Recall that super attacks require and consume 1 Ki bar. Handle the case of insufficient Ki. The method should then rely on its counterpart in the super class to perform the attack.

The special case of a `MaximumCharge` attack only provides the fighter with 3 additional Ki bars.

7.3 Ultimate Attack

Think about the needed functionality of the `void onUse(BattleOpponent attacker, BattleOpponent defender, boolean defenderBlocking)` method for ultimate attacks. Recall that ultimate attacks require and consume 3 Ki bars. Handle the case of insufficient Ki. The method should then rely on its counterpart in the super class to perform the attack.

The special case of a `SuperSaiyan` attack should only be applied to Saiyan fighters and sets their transformed value to true.

8 Dragon

`DragonWish[] getWishes()`: Returns an array of wishes that dragon can offer. This list includes:

1. Senzu beans, if any.
2. Ability points, if any.
3. One random super attack, if any.
4. One random ultimate attack, if any.

This method will be used in further milestones.

9 Battle

1. Should fire events handled by `BattleListener`.
2. `ArrayList<Attack> getAssignedAttacks()`: Returns the current attacker's attacks as an array list containing:
   - An instance of PhysicalAttack
   - All super attacks that are assigned to the attacker.
   - All ultimate attacks that are assigned to the attacker.

   Hint: You can use the `addAll()` method of java (check http://goo.gl/UPjyTk).
3. In the `Battle` constructor the transformation state of the fighter should be reset in case the fighter was a Saiyan transformed from a previous battle.
4. `void attack(Attack attack)`: Performs an attack by calling the appropriate methods in the `Attack` class. After any attack the turn ends. The method should also notify the appropriate listeners because this will be needed for future milestones.
5. `void block()`: This method performs the block action, causing the opponent's attack in the next turn only to be blocked according to the block rules specified in the game description. After using block option the turn ends. The method should also notify the appropriate listeners because this will be needed for future milestones.
6. `void use(Player player, Collectible collectible)`: Applies the action of the chosen `collectible` on `player`. Senzu beans restore the health points and stamina of the active fighter of the player. Handle the case where the player doesn't have enough senzu beans. After using a collectible the turn ends. Make sure to update the player's collectibles after using any one. The method should also notify the appropriate listeners because this will be needed for future milestones.
7. `BattleOpponent getDefender()`: Returns the battle party that is currently the defender.
8. `void play()`: Simulates the playing of the foe by randomly selecting either block or a random foe-assigned attack.
9. `void start()`: The method should also notify the appropriate listeners that the battle is about to start because this will be needed for future milestones.
10. `void endTurn()`: Think about the checks needed when ending a turn. You should check if the battle already has a winner; if so call the appropriate methods. The `onAttackerTurn`, `onDefenderTurn` and `switchTurn` methods should be called here. Take care that the effect of `onAttackerTurn` and `onDefenderTurn` should occur at the beginning of a turn; think about the order of the method calls. Reset the flags for blocking players and finally switch the turn. The method should also notify the appropriate listeners because this will be needed for future milestones.
11. `void switchTurn()`: Sets the current attacker to be the defender and the current defender to be the attacker.

10 Game

1. The class should implement the `PlayerListener`, `WorldListener` and `BattleListener`.
2. Should fire the events that will be handled by the `GameListener` in the future milestones.
3. In this milestone we should extend the map creation by changing the range of the foe's levels that are added to the map according to the fighter's level. This is decided based on the specific `Database-Foes-RangeN.csv` file to be loaded. The "N" in the suffix is between 1 and 5. 1 Range covers 10 levels. The Range N depends on max level of fighters, i.e. if max level $<= 10$ then $N = 1$, if max level $<= 20$ then $N = 2$, etc. If max level > 50, also use $N = 5$ since it's the last range.

4. When the ENDED battle event is received, think about performing all the actions that should be done upon winning a battle:
   - The appropriate xp points should be added
   - The foe's skills should be unlocked i.e. added to the list of super and ultimate attacks belonging to the player.
   - In case of a strong foe, a new map should be unlocked (created) and the `exploredMaps` should be incremented by one.
     Hint: Before calling `generateMap` make sure the cell values of the old map do not interfere with the generation of the new map.
   - For the purposes of this milestone, nothing should happen in case the non-playable fighter wins.