

# Humans vs Monsters

Sofia Galante



UNIVERSITÀ DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

<b>1</b>	<b>Introduzione e requisiti</b>	<b>4</b>
1.1	Il progetto . . . . .	4
1.2	Use Case Diagram . . . . .	5
<b>2</b>	<b>Progettazione</b>	<b>6</b>
2.1	Class Diagram . . . . .	6
2.2	Sequence Diagram . . . . .	7
2.3	Mockup . . . . .	9
<b>3</b>	<b>Implementazione</b>	<b>12</b>
3.1	Classi . . . . .	12
3.1.1	Character . . . . .	12
3.1.2	RealCharacter e le sue implementazioni . . . . .	12
3.1.3	ProxyCharacter . . . . .	14
3.1.4	CharactersByID . . . . .	14
3.1.5	Network . . . . .	15
3.1.6	Observer . . . . .	15
3.1.7	Game . . . . .	16
3.1.8	PriorityQueueComparator . . . . .	18
3.1.9	StatusBar . . . . .	19
3.1.10	ObjectOrAbility e le sue implementazioni . . . . .	19
3.1.11	Menu, MenuItem . . . . .	21
3.1.12	Action e le sue implementazioni . . . . .	22
3.1.13	GameMap . . . . .	22
3.1.14	GameMapPosition . . . . .	23
3.2	Dettagli su pattern utilizzati . . . . .	25
3.2.1	Singleton . . . . .	25
3.2.1.1	Dettagli . . . . .	25
3.2.1.2	Utilizzo e motivazioni . . . . .	25
3.2.2	Proxy . . . . .	26
3.2.2.1	Dettagli . . . . .	26

3.2.2.2	Utilizzo e motivazioni . . . . .	27
3.2.3	Observer . . . . .	28
3.2.3.1	Dettagli . . . . .	28
3.2.3.2	Utilizzo e motivazioni . . . . .	29
3.2.4	Command . . . . .	31
3.2.4.1	Dettagli . . . . .	31
3.2.4.2	Utilizzo e motivazioni . . . . .	31

## **4 Unit Test svolti 33**

4.1	GameTest() . . . . .	33
4.2	GameMapTest() . . . . .	35
4.3	Test su oggetti e abilità . . . . .	38

# 1 Introduzione e requisiti

## 1.1 Il progetto

**Humans vs Monsters** è un videogioco strategico a turni in cui due giocatori (connessi in remoto) combattono l'uno contro l'altro in una mappa a griglia (20x15). Uno è a capo di un esercito di umani, l'altro di una squadra di mostri. Entrambe le fazioni sono composte da 3 personaggi.

La guerra si svolge in una mappa a griglia. I personaggi di ogni giocatore si trovano inizialmente su due angoli opposti della mappa.

Il gioco è diviso in round. All'inizio di un round, il programma genera l'ordine di azione dei personaggi in base alla loro velocità attuale.

L'ordine di gioco, i punti vita (HP) e i punti magia (MP) del personaggio corrente sono mostrati da una status-bar.

Durante il turno di uno dei propri personaggi, il giocatore può eseguire le seguenti azioni al massimo una volta:

- Muoversi (ogni personaggio si muove in modo diverso). Non ci si può spostare in una casella occupata da un altro personaggio.
- Eseguire un attacco su un nemico in una casella adiacente alla sua.
- Utilizzare un oggetto (se è umano) o un'abilità (se è un mostro) su se stesso o su un alleato in una casella adiacente alla sua. Oggetti e abilità di ogni personaggio sono preimpostati.
- Passare il turno.

Vince chi per primo sconfigge l'intero plotone nemico.

Gli oggetti e le abilità disponibili nel gioco sono:

- **Pozione:** un oggetto che recupera 20HP.
- **Ali della Velocità:** un oggetto che aumenta la velocità di 3.
- **Lozione fortificante:** un oggetto che aumenta la difesa di 4.
- **Cura:** un'abilità che fa recuperare un numero di HP pari al 20% degli HP di chi la utilizza e costa 10MP.
- **VelocitàSU:** un'abilità che aumenta la velocità di un numero pari al 20% della difesa di chi la utilizza e costa 15MP.

- **ForzaSU**: un'abilità che aumenta la forza di un numero pari al 20% della difesa di chi la utilizza e costa 15 MP.

I personaggi umani sono:

- **L'arciere**, che si muove in diagonale di due caselle e possiede *tre Pozioni* e *un paio di Ali della Velocità*.
- **Il guerriero**, che si muove dritto di due caselle o una in diagonale e possiede *due paia di Ali della Velocità* e *due Lozioni fortificanti*.
- **Il mago**, che si muove a L e possiede *una Pozione* e *tre Lozioni fortificanti*.

I mostri sono:

- **Il goblin**, che si muove dritto di quattro caselle e possiede le abilità *ForzaSU* e *VelocitàSU*.
- **Il drago**, che si muove di una o tre caselle in diagonale e possiede le abilità *ForzaSU* e *Cura*.
- **Lo slime**, che si muove dritto di una o tre caselle e possiede le abilità *Cura* e *VelocitàSU*.

## 1.2 Use Case Diagram

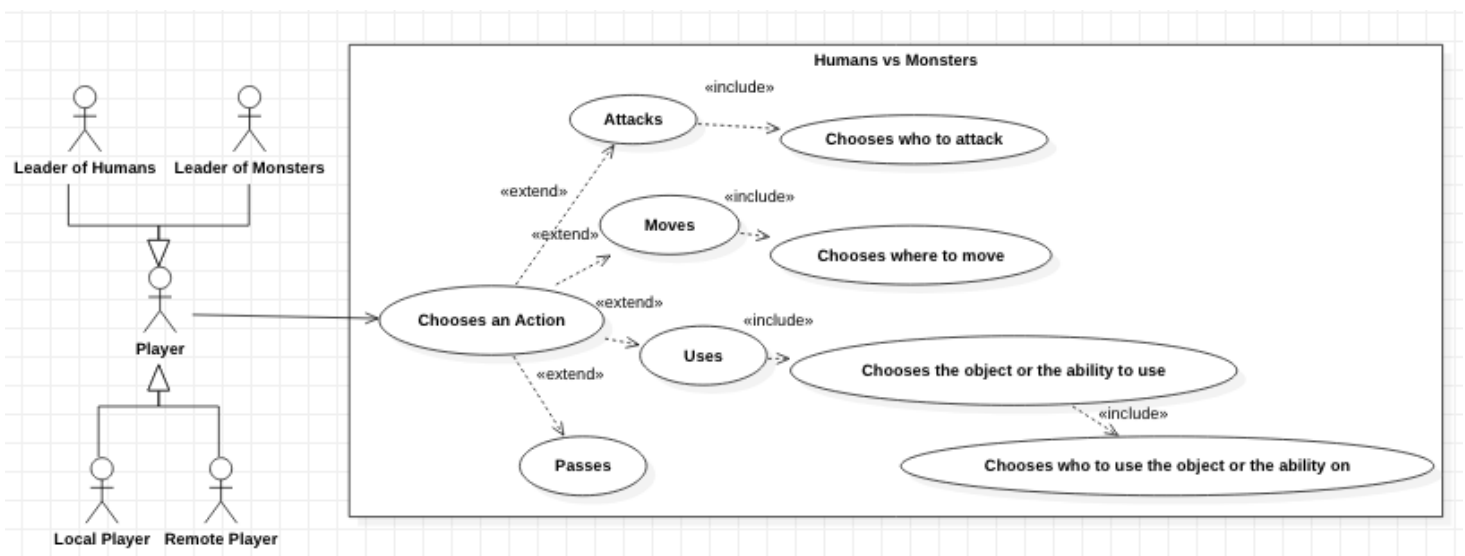


Figure 1: Use Case Diagram.

## 2 Progettazione

In questa sezione vengono riportati il Class Diagram, due possibili Sequence Diagram e alcuni Mockup per capire come è fatta l'interfaccia utente del gioco.

### 2.1 Class Diagram

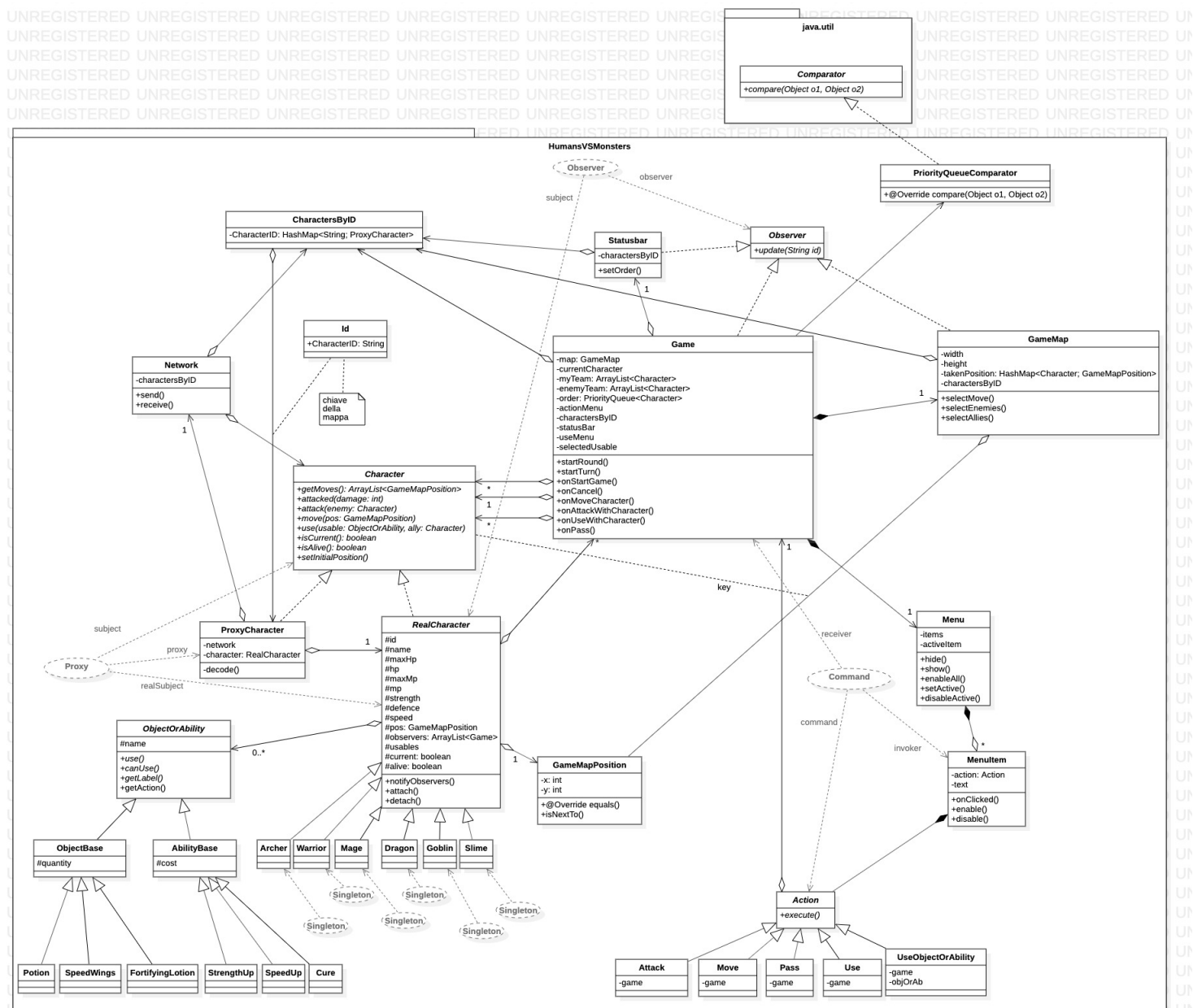


Figure 2: Class Diagram.

## 2.2 Sequence Diagram

Nei seguenti *Sequence Diagram* vengono rappresentati cosa succede quando il gioco viene avviato e cosa accade durante l'esecuzione di una delle possibili azioni (nel nostro caso "Move"). Il FrontEnd rappresenta l'interfaccia di gioco (non implementata).

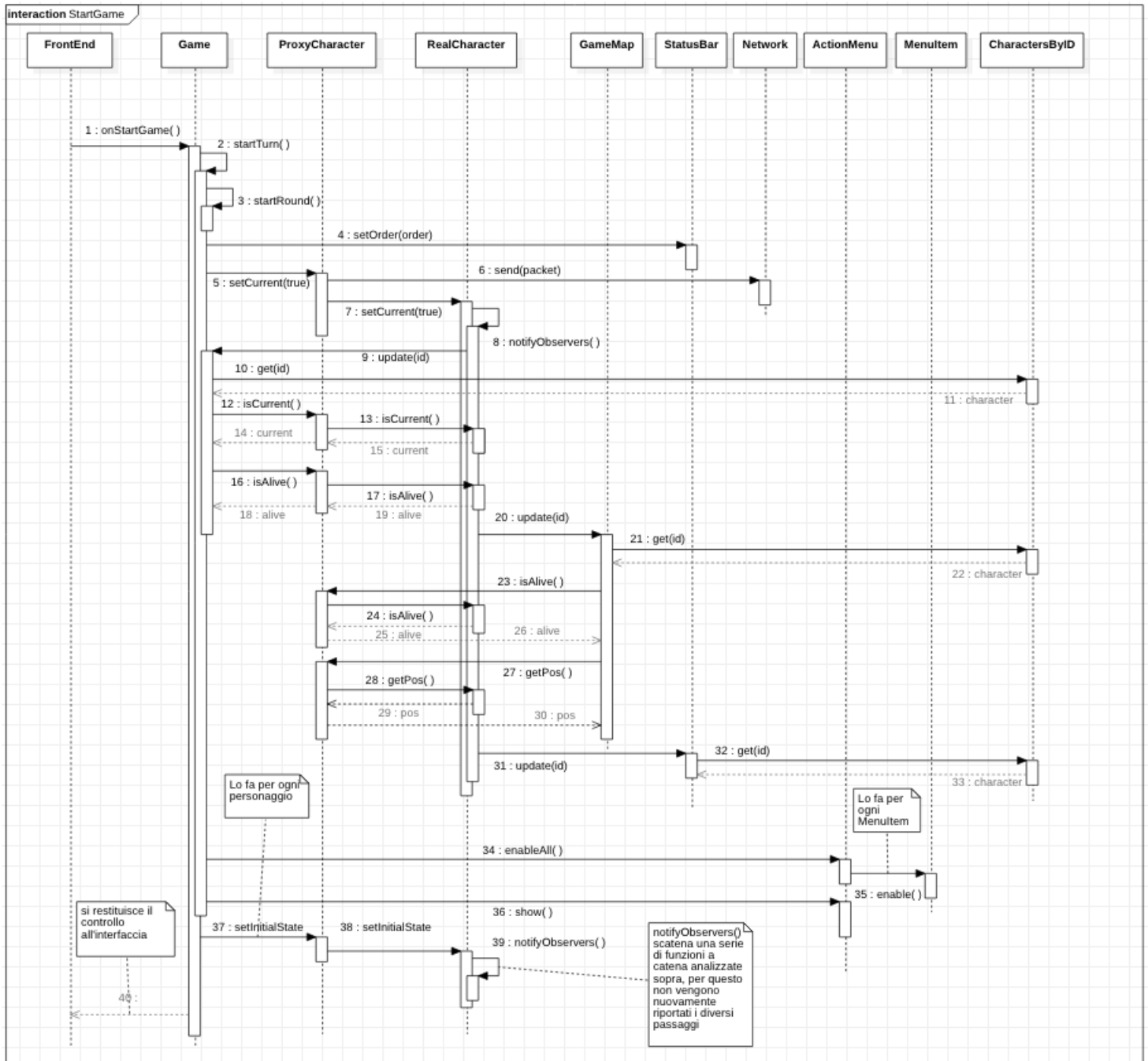


Figure 3: Sequence Diagram per l'inizio di una partita.

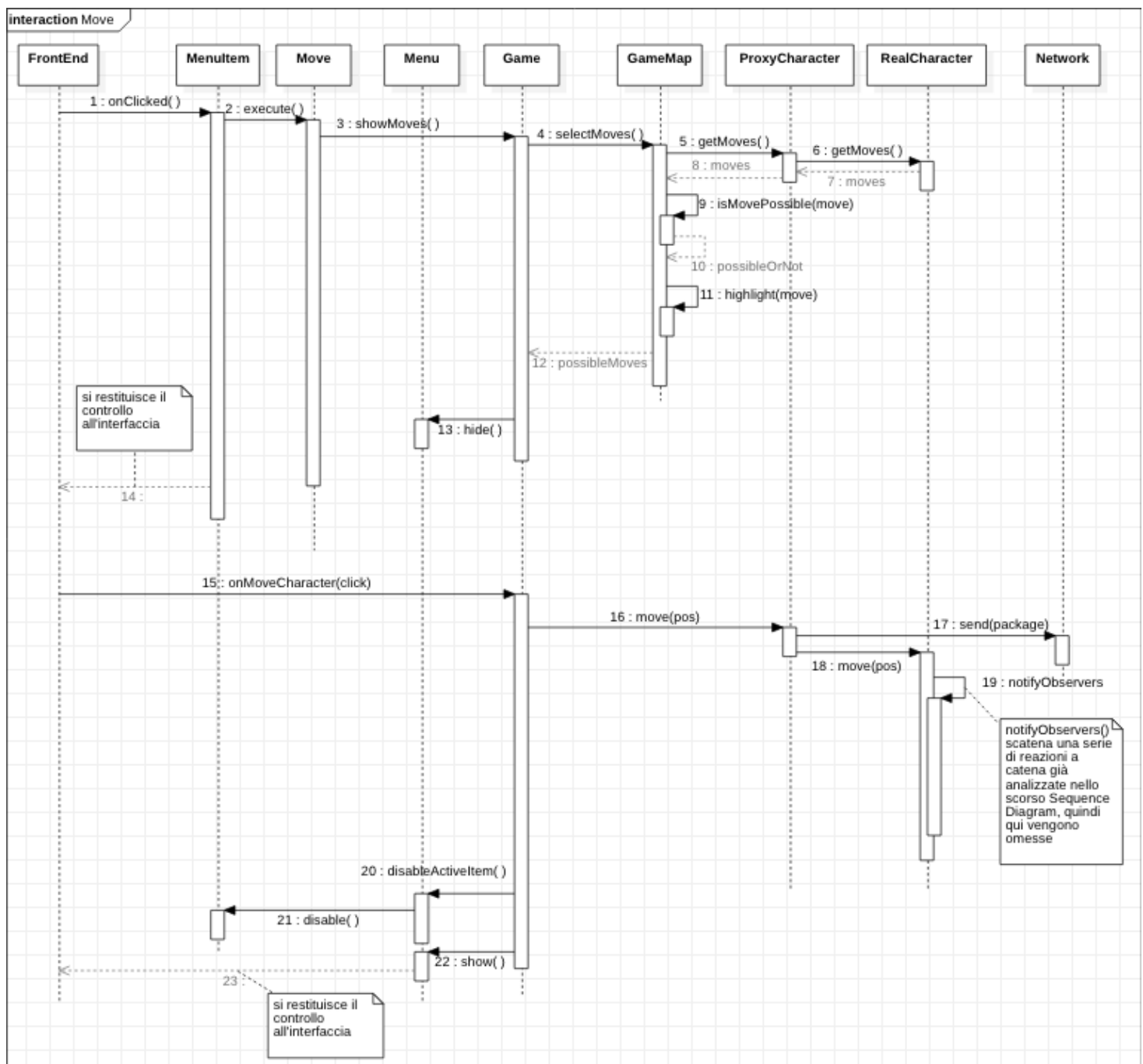


Figure 4: Sequence Diagram per il comando "Move".



## 2.3 Mockup

Nota: nei seguenti Mockup si vede la partita dal punto di vista dal giocatore che guida l'esercito di umani.

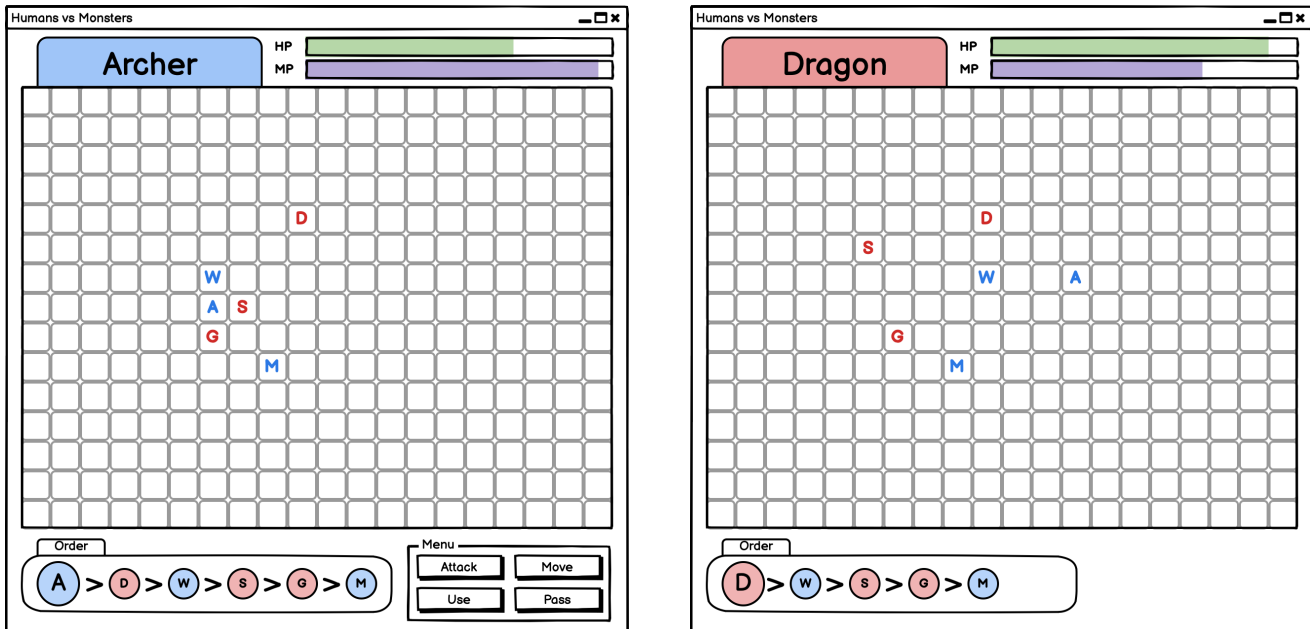


Figure 5: Differenza tra il turno del giocatore e quello del suo avversario.

Il giocatore vede il menù in basso a destra solo se è il turno di uno dei suoi personaggi. Da queste due figure si può inoltre notare come la barra dell'ordine di gioco si svuoti ogni volta che un personaggio ha passato il proprio turno; quando sarà vuota, inizierà un nuovo round e l'ordine verrà ricalcolato tenendo conto delle possibili modifiche alla statistica "velocità".

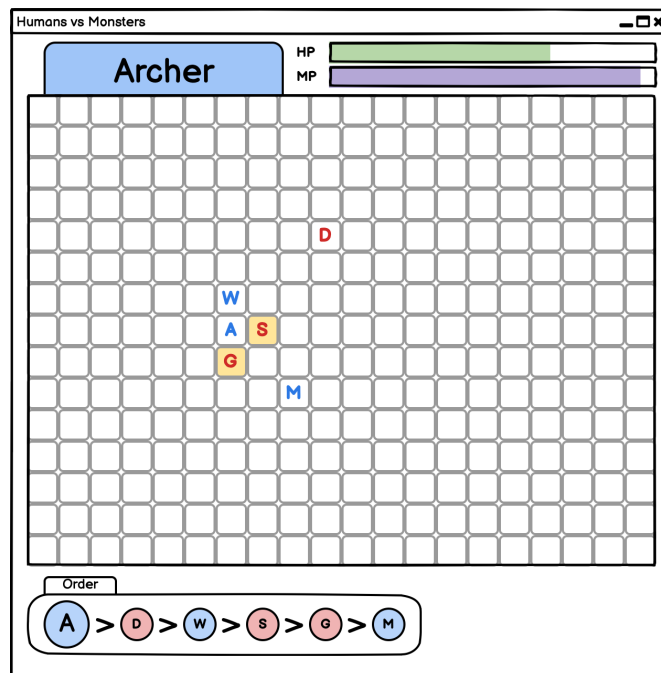


Figure 6: Il comando "Attack".

Quando si decide di attaccare, le caselle in cui sono posizionati i nemici che è possibile attaccare si illuminano.

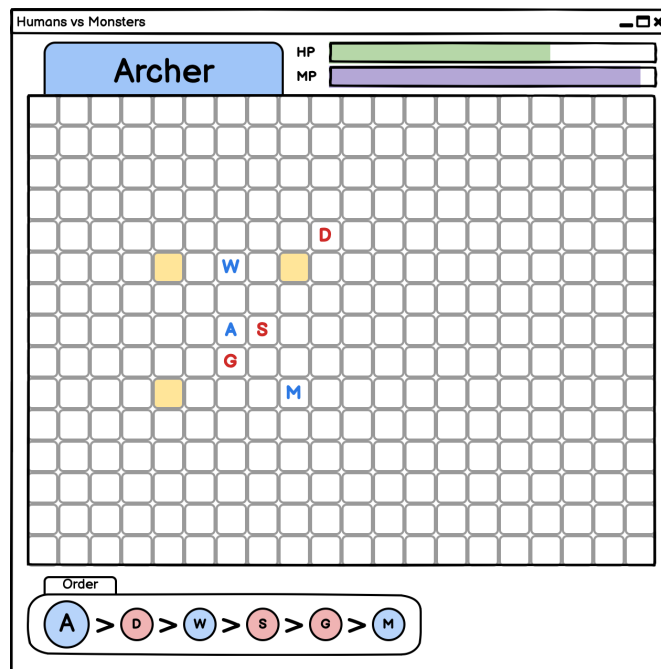


Figure 7: Il comando "Move".

Se si decide di spostarsi, il gioco selezionerà le caselle in cui è possibile muoversi.

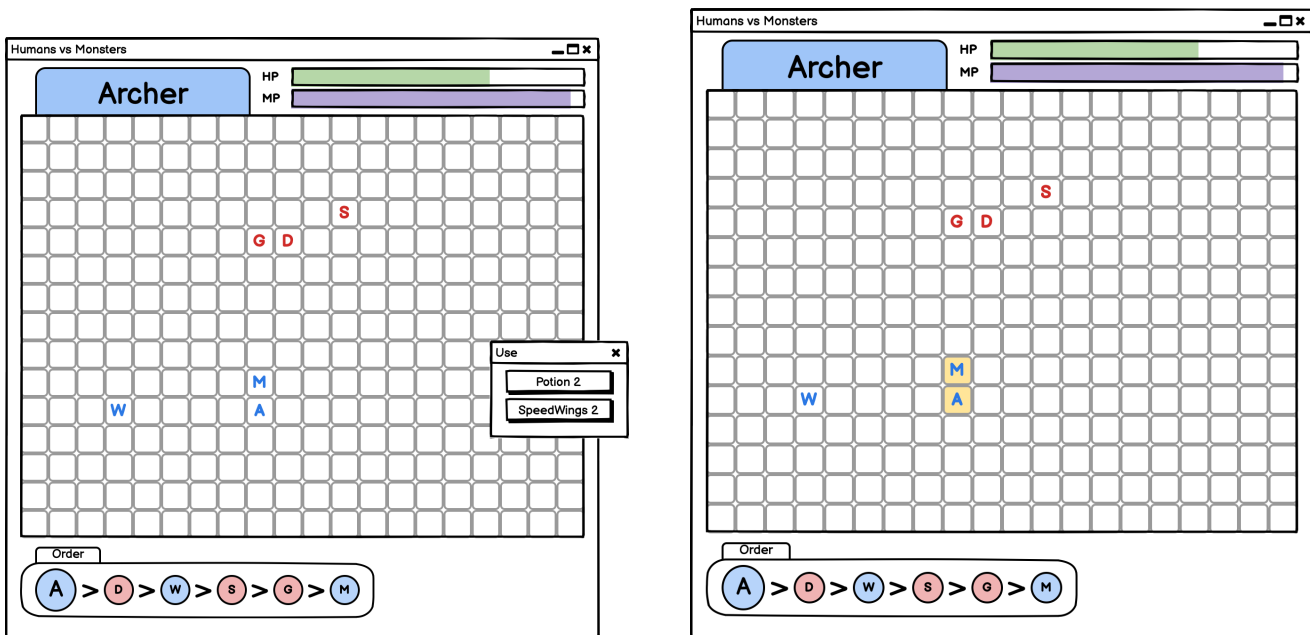


Figure 8: L'utilizzo di un'abilità o un oggetto

Premendo il tasto *Use* comparirà a schermo la lista degli oggetti o delle abilità del personaggio. Una volta scelto ciò che si vuole utilizzare, il gioco colorerà le caselle su cui è possibile agire (cioè quelle occupate dal personaggio e dai suoi alleati a lui adiacenti).

Ogni volta che si sceglie un'azione il menù viene nascosto, come si è potuto vedere dalle figure precedenti; esso torna visibile quando l'azione è stata completata.

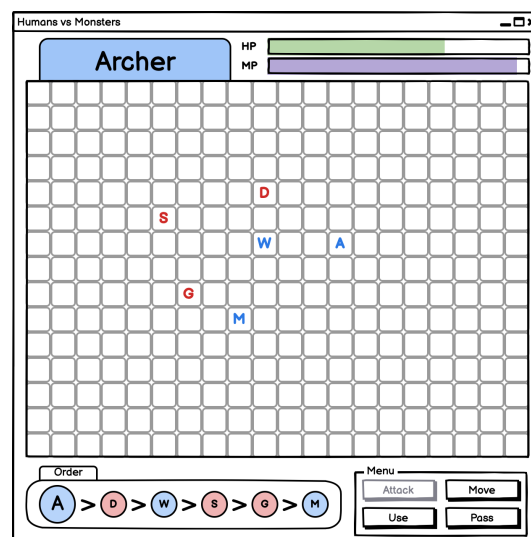


Figure 9: Ciò che accade dopo aver eseguito un comando ("Attack" in questo caso).

Dopo aver eseguito un'azione il pulsante corrispondente viene disabilitato.

## 3 Implementazione

### 3.1 Classi

In questa sezione verranno presentate le classi che compongono il progetto, mostrando quali di queste fanno parte di un Design Pattern. Per dettagli implementativi sui pattern e le motivazioni per cui si sono utilizzati si rimanda alla sezione 3.2.

#### 3.1.1 Character

**Character** è l'interfaccia da cui vengono implementati tutti i personaggi del gioco.

```
23 interface Character {
    public ArrayList<GameMapPosition> getMoves();
    public void attacked(int damage);
    public void attack(Character enemy);
    public void move(GameMapPosition pos);
    public ArrayList<ObjectOrAbility> getObjectsOrAbilities();
    public void use(ObjectOrAbility usable, Character ally);
    public void setInitialPosition();
}
```

Figure 10: Interfaccia Character.

#### 3.1.2 RealCharacter e le sue implementazioni

**RealCharacter** è una delle due implementazioni di **Character**.

**RealCharacter** è una *classe astratta* che contiene tutti gli attributi di un personaggio e implementa tutti i metodi dell'interfaccia **Character** tranne i metodi *getMoves()* e *setInitialPosition()*, in quanto questi dipendono dal singolo personaggio e non sono comuni a tutti.

```
15 abstract class RealCharacter implements Character{
16     protected String id;
17     protected String name;
18     protected int maxHp;
19     protected int maxMp;
20     protected int hp;
21     protected int mp;
22     protected int strength;
23     protected int defence;
24     protected int speed;
25     protected GameMapPosition pos;
26     protected ArrayList<Observer> observers;
27     protected ArrayList<ObjectOrAbility> usables;
28     protected boolean current;
29     protected boolean alive;
30
31     RealCharacter(){
32         observers = new ArrayList();
33         alive = true;
34         current = false;
35     }
36
37     @Override public void attacked(int damage){
38         hp -= damage;
39         if(hp <= 0){
40             hp = 0;
41             alive = false;
42             notifyObservers();
43         }
44     }
45     @Override public void attack(Character enemy){
46         int damage = (this.strength - enemy.getDefence())/enemy.getMaxHp();
47         enemy.attacked(damage);
48     }
49     @Override public void move(GameMapPosition pos){
50         this.pos = pos;
51         notifyObservers();
52     }
53     @Override public ArrayList<ObjectOrAbility> getObjectsOrAbilities(){
54         return usables;
55     }
56     @Override public void use(ObjectOrAbility usable, Character ally){
57         usable.use(own: this, receiver: ally);
58     }
59 }
```

Figure 11: Classe astratta RealCharacter: a sinistra i suoi attributi e il costruttore, a destra i metodi implementati (escludendo getter e setter).

Questa classe fa parte di un *pattern Proxy* e di un *pattern Observer*.

Le sei implementazioni del **RealCharacter** (**Archer**, **Warrior**, **Mage**, **Dragon**, **Slime** e **Goblin**) rappresentano i sei personaggi in gioco. L'implementazione di queste sei classi differisce unicamente per i metodi *getMoves()* e *setInitialPosition()* e nei diversi valori assegnati agli attributi.

```

14 class Archer extends RealCharacter{
15     static private Archer instance;
16
17     private Archer(){
18         id = "H01";
19         name = "Archer";
20         maxHp = 200;
21         hp = maxHp;
22         maxMp = 90;
23         mp = maxMp;
24         strength = 20;
25         defence = 10;
26         speed = 15;
27
28         usables = new ArrayList();
29         usables.add(new Potion( quantity:3));
30         usables.add(new SpeedWings( cost:1));
31     }
32     static public Archer instantiate(){
33         if (instance == null){
34             instance = new Archer();
35         }
36         return instance;
37     }
38
39     @Override public ArrayList<GameMapPosition> getMoves(){
40         ArrayList<GameMapPosition> moves = new ArrayList();
41         moves.add(new GameMapPosition(pos.getX()+2, pos.getY()+2));
42         moves.add(new GameMapPosition(pos.getX()+2, pos.getY()-2));
43         moves.add(new GameMapPosition(pos.getX()-2, pos.getY()+2));
44         moves.add(new GameMapPosition(pos.getX()-2, pos.getY()-2));
45
46         return moves;
47     }
48
49     @Override public void setInitialPosition(){
50         pos = new GameMapPosition( x:0, y:12);
51         notifyObservers();
52     }
53 }
54
55 class Dragon extends RealCharacter{
56     static private Dragon instance;
57
58     private Dragon(){
59         id = "M03";
60         name = "Dragon";
61         maxHp = 150;
62         hp = maxHp;
63         maxMp = 120;
64         mp = maxMp;
65         strength = 30;
66         defence = 20;
67         speed = 25;
68
69         usables = new ArrayList();
70         usables.add(new StrengthUP());
71         usables.add(new Cure());
72     }
73     static public Dragon instantiate(){
74         if (instance == null){
75             instance = new Dragon();
76         }
77         return instance;
78     }
79
80     @Override public ArrayList<GameMapPosition> getMoves(){
81         ArrayList<GameMapPosition> moves = new ArrayList();
82
83         moves.add(new GameMapPosition(pos.getX()+3, pos.getY()+3));
84         moves.add(new GameMapPosition(pos.getX()+3, pos.getY()-3));
85         moves.add(new GameMapPosition(pos.getX()+1, pos.getY()-1));
86         moves.add(new GameMapPosition(pos.getX()+1, pos.getY()+1));
87         moves.add(new GameMapPosition(pos.getX()-1, pos.getY()+1));
88         moves.add(new GameMapPosition(pos.getX()-1, pos.getY()-1));
89         moves.add(new GameMapPosition(pos.getX()-3, pos.getY()+3));
90         moves.add(new GameMapPosition(pos.getX()-3, pos.getY()-3));
91
92         return moves;
93     }
94
95     @Override public void setInitialPosition(){
96         pos = new GameMapPosition( x:19, y:0);
97         notifyObservers();
98     }
99 }

```

Figure 12: Due classi personaggio per esempio.

Ognuna di queste classi è un *pattern Singleton*.

### 3.1.3 ProxyCharacter

**ProxyCharacter** è la classe proxy che implementa ogni metodo dell'interfaccia **Character**. Questa classe possiede come attributi il **RealCharacter** di cui è il proxy e il **Network** a cui deve inviare le modifiche che il **RealCharacter** subisce. **ProxyCharacter** è un proxy "intelligente": esso infatti invoca il **Network** solo quando necessario, mentre le altre volte accede semplicemente al **RealCharacter** corrispondente.

Il **ProxyCharacter** è anche il responsabile della codifica e decodifica del packet da inviare e ricevere dalla rete.

```
107 @Override public void move(GameMapPosition pos){
108     String packet = character.getId()+" movesTo "+pos.getX()+" "+pos.getY();
109     network.send(packet);
110     character.move(pos);
111 }
```

```
92 @Override public void setInitialPosition(){
93     character.setInitialPosition();
94 }
```

Figure 13: Due metodi della classe ProxyCharacter per esempio: a sinistra un metodo in cui viene invocato il Network, a destra un metodo in cui si accede solo al RealCharacter.

Questa classe fa parte di un *pattern Proxy*.

### 3.1.4 CharactersByID

La classe **CharactersByID** rappresenta una mappa globale che lega ogni **ProxyCharacter** al suo id.

La mappa è stata implementata con una HashMap e viene utilizzata da quattro diverse classi: il **Game**, il **Network**, la **StatusBar** e la **GameMap**.

```
14 class CharactersByID {
15     private HashMap<String, ProxyCharacter> characterByID;
16
17     public CharactersByID(){
18         characterByID = new HashMap();
19     }
20
21     public void add(ProxyCharacter c){
22         characterByID.put( key:c.getId(), value: c);
23     }
24
25     public ProxyCharacter get(String id){
26         return characterByID.get( key:id);
27     }
28 }
```

Figure 14: Classe CharactersByID.

### 3.1.5 Network

Il **Network** viene utilizzato dal **ProxyCharacter** per inviare e ricevere messaggi dalla rete.

Il **Network** possiede un'istanza di tipo **CharactersByID** grazie alla quale può inviare il packet al **ProxyCharacter** giusto (si noti che ogni packet inizia con l'id del personaggio a cui si riferisce).

```
13 class Network{
14     private CharactersByID characterByID;
15
16     public Network(CharactersByID characterByID){
17         this.characterByID = characterByID;
18     }
19
20     public void send(String packet){
21         //invia informazioni attraverso la rete...
22     }
23
24     public void receive(String packet){
25         //riceve informazioni dalla rete...
26         //utilizza la tabella per inviare il packet al Proxy corrispondente
27         String id = packet.substring(beginIndex:0, endIndex:3);
28         characterByID.get(id).decode(packet);
29     }
30 }
```

Figure 15: Classe Network.

### 3.1.6 Observer

E' un'interfaccia che viene utilizzata nel *pattern Observer*. Le sue implementazioni sono la classe **Game**, la classe **StatusBar** e la classe **GameMap**.

```
14 interface Observer {
    public void update(String characterID);
}
```

Figure 16: Interfaccia Observer

### 3.1.7 Game

La classe **Game** è il cuore dell'intero progetto. Essa coordina la partita, permettendo alle diverse classi di interagire l'una con l'altra.

```
15 class Game implements Observer{
16     private GameMap map;
17     private Character currentCharacter;
18     private ArrayList<Character> myTeam;
19     private ArrayList<Character> enemyTeam;
20     private PriorityQueue<Character> order;
21     private Menu actionMenu;
22     private Menu useMenu;
23     private ObjectOrAbility selectedUsable;
24     private CharactersByID charactersByID;
25     private StatusBar statusBar;
26
27     public Game(ArrayList<Character> myTeam, ArrayList<Character> enemyTeam, CharactersByID characterByID, StatusBar statusBar){
28         map = new GameMap(charactersByID);
29         actionMenu = new Menu( title: "Menu");
30         useMenu = null;
31         selectedUsable = null;
32
33         actionMenu.addItem(new MenuItem( menu: actionMenu, text: "Attack", new Attack( game: this)));
34         actionMenu.addItem(new MenuItem( menu: actionMenu, text: "Move", new Move( game: this)));
35         actionMenu.addItem(new MenuItem( menu: actionMenu, text: "Use", new Use( game: this)));
36         actionMenu.addItem(new MenuItem( menu: actionMenu, text: "Pass", new Pass( game: this)));
37
38         this.myTeam = myTeam;
39         this.enemyTeam = enemyTeam;
40         this.charactersByID = characterByID;
41         this.statusBar = statusBar;
42
43         order = new PriorityQueue(new PriorityQueueComparator());
44     }
```

Figure 17: La classe Game.

Questa classe svolge le seguenti azioni:

- tiene traccia del personaggio corrente e dello stato generale del gioco;
- calcola l'ordine dei turni all'inizio di ogni Round e lo salva nella PriorityQueue order;
- implementa sia metodi per calcolare le possibili mosse che il giocatore può compiere e le mostra a schermo, sia metodi per ricevere in input la risposta del giocatore;
- si occupa di mostrare e nascondere il **Menu** e di abilitare e disabilitare i diversi **MenuItem**;
- fa partire e concludere la partita.

I metodi che iniziano con "on" (*onStartGame()*, *onMoveCharacter()*, *onAttack-WithCharacter()*, *onUseWithCharacter()*, *onPass()*, *onCancel()* e *onCancelUse-Menu()*) sono i metodi che vengono eseguiti alla fine di ogni processo di scelta



di una determinata azione, dopo che il giocatore ha dato un input attraverso l'interfaccia grafica.

```

92 | public void onStartGame(){
93 |     startTurn();
94 |     for(int i=0; i<myTeam.size(); i++)
95 |         myTeam.get(index:i).setInitialPosition();
96 |     for(int i=0; i<enemyTeam.size(); i++)
97 |         enemyTeam.get(index:i).setInitialPosition();
98 | }
99 |
100 |
101 | public void onCancel(){
102 |     if(useMenu != null){
103 |         selectedUsable = null;
104 |         useMenu.setActive(item:null);
105 |         useMenu.show();
106 |     }
107 |     else{
108 |         actionMenu.setActive(item:null);
109 |         actionMenu.show();
110 |     }
111 | }
112 |
113 | public void onCancelUseMenu(){
114 |     useMenu.hide();
115 |     useMenu = null;
116 |     onCancel();
117 | }
118 |
119 | public void onMoveCharacter(GameMapPosition click){
120 |     currentCharacter.move(pos:click);
121 |     actionMenu.disableActiveItem();
122 |     actionMenu.show();
123 | }
124 |
125 | public void onAttackWithCharacter(GameMapPosition click){
126 |     Character enemy = map.getCharacter(pos:click);
127 |     currentCharacter.attack(enemy);
128 |     actionMenu.disableActiveItem();
129 |     actionMenu.show();
130 | }
131 |
132 | public void onUseWithCharacter(GameMapPosition click){
133 |     Character ally = map.getCharacter(pos:click);
134 |     currentCharacter.use(usable:selectedUsable, ally);
135 |     selectedUsable = null;
136 |     useMenu = null;
137 |     actionMenu.disableActiveItem();
138 |     actionMenu.show();
139 | }
140 |
141 | public void onPass(){
142 |     currentCharacter.setCurrent(current:false);
143 | }

```

Figure 18: I metodi che iniziano con "on".

Invece, i metodi che iniziano con "show" (*showMoves()*, *showEnemies()*, *showAllies()* e *showObjectsOrAbilities()*) sono i metodi che si occupano di calcolare le possibili mosse del giocatore e mostrarle a schermo.

```

145 | public void showMoves(){
146 |
147 |     ArrayList<GameMapPosition> possibleMoves = map.selectMoves(c:currentCharacter);
148 |     actionMenu.hide();
149 |     //mostra le mosse possibili a schermo
150 | }
151 |
152 | public void showEnemies(){
153 |     ArrayList<GameMapPosition> enemiesPos = map.selectEnemies(c:currentCharacter, enemies:enemyTeam);
154 |     actionMenu.hide();
155 |     //mostra i nemici che puoi attaccare
156 | }
157 |
158 | public void showAllies(ObjectOrAbility usable){
159 |     selectedUsable = usable;
160 |     ArrayList<GameMapPosition> alliesPos = map.selectAllies(c:currentCharacter, allies:myTeam);
161 |     useMenu.hide();
162 |     //mostra gli alleati su cui si può usare un oggetto/abilità
163 | }
164 |
165 | public void showObjectsOrAbilities(){
166 |     ArrayList<ObjectOrAbility> usables = currentCharacter.getObjectsOrAbilities();
167 |     useMenu = new Menu(title:"Use");
168 |     for (int i=0; i<usables.size(); i++){
169 |         ObjectOrAbility item = usables.get(index:i);
170 |         if(item.canUse(c:currentCharacter))
171 |             useMenu.addItem(new MenuItem(menu:useMenu, text:item.getLabel(), action:item.getAction(g:this)));
172 |     }
173 |     actionMenu.hide();
174 |     useMenu.show();
175 | }

```

Figure 19: I metodi che iniziano con "show".

Infine, ci sono i due metodi che servono per cambiare round e turno durante la partita (*startRound()* e *startTurn()*).

```

73  □ public void startRound(){
74      for (int i = 0; i<myTeam.size(); i++)
75          order.add( e:myTeam.get( index:i));
76      for (int i = 0; i<enemyTeam.size(); i++)
77          order.add( e:enemyTeam.get( index:i));
78  }
79
80  □ public void startTurn(){
81      if (order.isEmpty())
82          startRound();
83      currentCharacter = order.remove();
84      statusBar.setOrder(order);
85      currentCharacter.setCurrent( current:true);
86      if(myTeam.contains( o:currentCharacter)){
87          actionMenu.enableAll();
88          actionMenu.show();
89      }
90  }

```

Figure 20: I metodi per il cambio di round e turno.

Questa classe fa parte di un *pattern Observer* e di un *pattern Command*.

### 3.1.8 PriorityQueueComparator

**PriorityQueueComparator** è un'implementazione dell'interfaccia **Comparator** presente nel package *java.util*. Questo comparatore viene utilizzato dalla *PriorityQueue* del Game e ordina i personaggi in base alla loro velocità e, in caso di pareggio, in ordine alfabetico.

```

14  class PriorityQueueComparator implements Comparator{
15      @Override public int compare(Object o1, Object o2){
16          Character c1 = (Character) o1;
17          Character c2 = (Character) o2;
18
19          if(c1.getSpeed() < c2.getSpeed())
20              return 1;
21          else if (c1.getSpeed() > c2.getSpeed())
22              return -1;
23          else
24              return (c1.getId().compareTo( anotherString: c2.getId()));
25      }
26  }

```

Figure 21: Classe PriorityQueueComparator.

### 3.1.9 StatusBar

La **StatusBar** ha come scopo quello di mostrare a schermo i dati del personaggio corrente e l'ordine dei turni di gioco.

```
14 class StatusBar implements Observer{
15     CharactersByID characterByID;
16
17     public StatusBar(CharactersByID characterByID){
18         this.characterByID = characterByID;
19     }
20
21     @Override public void update(String id){
22         Character c = characterByID.get(id);
23         //se il personaggio è quello corrente, lo mostra a schermo, altrimenti lo ignora
24         if(c.isCurrent()){
25             //mostra il nome del personaggio, hp e mp.
26         }
27     }
28
29     public void setOrder(Queue<Character> order){
30         //mostra a schermo l'ordine di gioco
31     }
32 }
```

Figure 22: Classe StatusBar.

Questa classe fa parte di un *pattern Observer*.

### 3.1.10 ObjectOrAbility e le sue implementazioni

La classe astratta **ObjectOrAbility** rappresenta un oggetto o un'abilità del gioco e definisce tutti i metodi che questi devono implementare.

```
13 abstract class ObjectOrAbility {
14     protected String name;
15     public abstract void use(Character owner, Character receiver);
16     public abstract boolean canUse(Character c);
17
18     public abstract String getLabel();
19
20     public String getName(){
21         return name;
22     }
23     public Action getAction(Game g){
24         return new UseObjectOrAbility( game:g, objOrAb:this);
25     }
26 }
```

Figure 23: Classe astratta ObjectOrAbility.

Da questa classe derivano altre due classi astratte, **ObjectBase** e **AbilityBase**, che implementano i due metodi (*getLabel()* e *canUse()*) comuni per tutti gli oggetti e abilità.

```

13  abstract class ObjectBase extends ObjectOrAbility{
14      protected int quantity;
15
16      @Override public String getLabel(){
17          return name + " x" + quantity;
18      }
19
20      @Override public boolean canUse(Character c){
21          return quantity>0;
22      }

```

```

13  abstract class AbilityBase extends ObjectOrAbility{
14      protected int cost;
15
16      @Override public String getLabel(){
17          return name + " " + cost + " MP";
18      }
19
20      @Override public boolean canUse(Character c){
21          return cost<=c.getMp();
22      }

```

Figure 24: Classi astratte ObjectBase e AbilityBase.

Infine, da **ObjectBase** vengono derivate le classi di tutti gli oggetti del gioco (**Potion**, **FortifyingLotion** e **SpeedWings**) e da **AbilityBase** le classi delle abilità (**SpeedUP**, **Cure** e **StrengthUP**). Ognuna di queste classi finali implementa il metodo *use()*.

```

12  class FortifyingLotion extends ObjectBase{
13      FortifyingLotion(int quantity){
14          this.quantity = quantity;
15          this.name = "Fortifying Lotion";
16      }
17
18      @Override public void use(Character owner, Character receiver){
19          receiver.setDefence(receiver.getDefence()+4);
20          quantity -= 1;
21      }
22  }

```

```

12  class Cure extends AbilityBase{
13      public Cure(){
14          this.cost = 10;
15          this.name = "Cure";
16      }
17
18      @Override public void use(Character owner, Character receiver){
19          receiver.setHp(receiver.getHp()+owner.getHp()*20/100);
20          owner.setMp(owner.getMp()-cost);
21      }
22  }

```

Figure 25: Una classe di un oggetto concreto (FortifyingLotion) e di un'abilità concreta (Cure).

### 3.1.11 Menu, MenuItem

La classe **MenuItem** rappresenta un "tasto" all'interno di un **Menu** (che è semplicemente una collezione di **MenuItem**).

Il **Menu** ha un metodo per apparire su schermo e uno per essere nascosto. Possiede inoltre un metodo per riabilitare tutti i **MenuItem** che contiene. Infine, salva nel suo stato il **MenuItem** attualmente "attivo", cioè quello che il giocatore ha cliccato e che ha fatto iniziare l'azione corrente.

Il **MenuItem** ha metodi per disabilitare e abilitare se stesso all'interno del **Menu**.

```
14 class Menu{
15     private String title;
16     private ArrayList<MenuItem> items;
17     private MenuItem activeItem;
18
19     public Menu(String title){
20         items = new ArrayList();
21         this.title = title;
22     }
23
24     public void addItem(MenuItem item){
25         items.add( item);
26     }
27
28     public void hide(){
29         //nasconde il Menu
30     }
31     public void show(){
32         //mostra il Menu
33     }
34     public void enableAll(){
35         for(int i=0; i<items.size(); i++)
36             items.get( index:i).enable();
37     }
38
39     public void setActive(MenuItem item){
40         activeItem = item;
41     }
42
43     public void disableActiveItem(){
44         activeItem.disable();
45         activeItem = null;
46     }
47 }
```

```
12 class MenuItem{
13     private String text;
14     private Action action;
15     private Menu menu;
16
17     public MenuItem(Menu menu, String text, Action action){
18         this.text = text;
19         this.action = action;
20         this.menu = menu;
21     }
22
23     public void onClicked(){
24         menu.setActive( item:this);
25         action.execute();
26     }
27
28     public void disable(){
29         //disabilità il bottone
30     }
31     public void enable(){
32         //abilità il bottone
33     }
34 }
```

Figure 26: Classi Menu e MenuItem.

**MenuItem** fa parte di un *pattern Command*.

### 3.1.12 Action e le sue implementazioni

L'interfaccia **Action** rappresenta una possibile azione che il giocatore può far svolgere ad un personaggio. Le sue diverse implementazioni (**Attack**, **Move**, **Use**, **UseObjectOrAbility** e **Pass**) specificano l'azione da svolgere, andando a implementare in modo differente il metodo *execute()*.

```
12 interface Action {  
13     public void execute();  
15 }
```

Figure 27: Interfaccia Action

Ogni implementazione di **Action** possiede un riferimento al **Game**, così da poter chiamare il metodo corrispondente alla loro azione. Un'implementazione particolare di **Action** è **UseObjectOrAbility** che rappresenta il tasto per utilizzare un particolare oggetto o abilità di un personaggio e che quindi, oltre al **Game**, possiede anche un riferimento all'oggetto o all'abilità a lui associato.

```
12 class Attack implements Action {  
13     private Game game;  
14  
15     public Attack(Game game){  
16         this.game = game;  
17     }  
18  
19     @Override public void execute(){  
20         game.showEnemies();  
21     }  
22 }
```

```
12 class UseObjectOrAbility implements Action{  
13     private Game game;  
14     private ObjectOrAbility objOrAb;  
15  
16     UseObjectOrAbility(Game game, ObjectOrAbility objOrAb){  
17         this.game = game;  
18         this.objOrAb = objOrAb;  
19     }  
20  
21     @Override public void execute(){  
22         game.showAllies(usable=objOrAb);  
23     }  
24  
25 }
```

Figure 28: Due implementazioni di Action: Attack e UseObjectOrAbility.

**Action** e le sue implementazioni fanno parte di un *pattern Command*.

### 3.1.13 GameMap

La classe **GameMap** rappresenta la mappa di gioco e nel suo stato mantiene tutte le posizioni occupate da ogni personaggio.

I metodi di questa classe vengono chiamati per ogni azione compiuta dal giocatore (tranne il **Pass**) e servono a selezionare le possibili caselle su cui il giocatore può agire. In particolare:

- *selectMoves()* seleziona le caselle in cui il giocatore può muoversi;
- *selectEnemies()* seleziona le caselle occupate dai nemici che possono essere attaccati (i.e. che sono adiacenti al personaggio che compie l'attacco);

- *selectAllies()* seleziona le caselle occupate dagli alleati adiacenti e dal personaggio stesso, indicando così su chi si può usare un oggetto/un'abilità.

```
50 public ArrayList<GameMapPosition> selectMoves(Character c){
51     ArrayList<GameMapPosition> possibleMoves = c.getMoves();
52     for(int i=0; i < possibleMoves.size(); i++){
53         if (isMovePossible( move: possibleMoves.get( index:i)))
54             highlight( position:possibleMoves.get( index:i));
55         else{
56             possibleMoves.remove( index:i);
57             i--;
58         }
59     }
60     return possibleMoves;
61 }
62
63 public ArrayList<GameMapPosition> selectEnemies(Character c, ArrayList<Character> enemies){
64     ArrayList<GameMapPosi> enemiesPos = new ArrayList();
65
66     for (int i=0; i<enemies.size(); i++){
67         Character enemy = enemies.get( index:i);
68         if (c.getPos().isNextTo( pos:enemy.getPos())){
69             highlight( position:enemy.getPos());
70             enemiesPos.add( e:enemy.getPos());
71         }
72     }
73     return enemiesPos;
74 }
75
76 public ArrayList<GameMapPosition> selectAllies(Character c, ArrayList<Character> allies){
77     ArrayList<GameMapPosition> alliesPos = new ArrayList();
78
79     for (int i=0; i<allies.size(); i++){
80         Character ally = allies.get( index:i);
81         if (c.getPos().isNextTo( pos:ally.getPos())){
82             highlight( position:ally.getPos());
83             alliesPos.add( e:ally.getPos());
84         }
85     }
86     alliesPos.add( e:c.getPos());
87     return alliesPos;
88 }
```

Figure 29: i metodi di selezione della GameMap

Questa classe fa parte di un *pattern Observer*.

### 3.1.14 GameMapPosition

La classe **GameMapPosition**, come dice il nome, rappresenta una posizione all'interno della mappa di gioco.

Questa classe fa l'override del metodo *equals()* della classe *Object* del package *java.lang*. Questo permette di poter osservare se due posizioni sono uguali confrontando le due coordinate.

Ha anche un metodo per capire se due posizioni sono adiacenti tra loro (*isNextTo()*).

```

22  @Override public boolean equals(Object o){
23      if (this == o)
24          return true;
25
26      if (!(o instanceof GameMapPosition)) {
27          return false;
28      }
29
30      GameMapPosition pos = (GameMapPosition) o;
31
32      if(pos.x==this.x && pos.y==this.y)
33          return true;
34
35      return false;
36  }

```

Figure 30: Il metodo equals() di GameMapPosition.

```

50  public boolean isNextTo(GameMapPosition pos){
51      int dX = Math.abs(pos.x - this.x);
52      int dY = Math.abs(pos.y - this.y);
53      if ((dX == 1 && dY == 0) || (dX == 0 && dY == 1))
54          return true;
55      return false;
56  }

```

Figure 31: Il metodo isNextTo() di GameMapPosition.



## 3.2 Dettagli su pattern utilizzati

### 3.2.1 Singleton

#### 3.2.1.1 Dettagli

Il **pattern Singleton** è un *Creational Pattern* che ha come intento quello di assicurare che una classe abbia una sola istanza e fornisce l'accesso globale alla stessa.

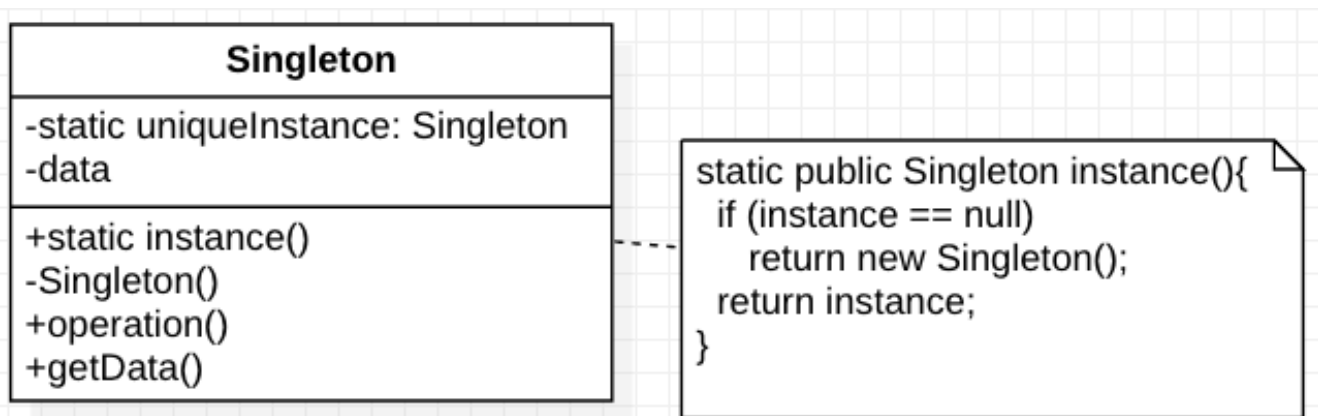


Figure 32: Class Diagram del pattern Singleton

#### 3.2.1.2 Utilizzo e motivazioni

Nel progetto, le sei classi dei personaggi (**Archer**, **Warrior**, **Mage**, **Dragon**, **Slime** e **Goblin**) sono sei diversi *pattern Singleton*. Questo pattern è stato utilizzato in quanto durante ogni partita può esistere una singola istanza di ogni personaggio.

## 3.2.2 Proxy

### 3.2.2.1 Dettagli

Il **pattern Proxy** è uno *Structural Pattern* che ha come scopo quello di fornire un surrogato di un altro oggetto per controllare l'accesso su quest'ultimo.

Gli elementi di un *pattern Proxy* sono:

- il **Subject** che è una classe astratta o un'interfaccia dell'oggetto;
- il **RealSubject** che è l'oggetto di cui si deve fornire un surrogato;
- il **Proxy** che mantiene un riferimento al *RealSubject* di cui è il surrogato e controlla l'accesso a esso.

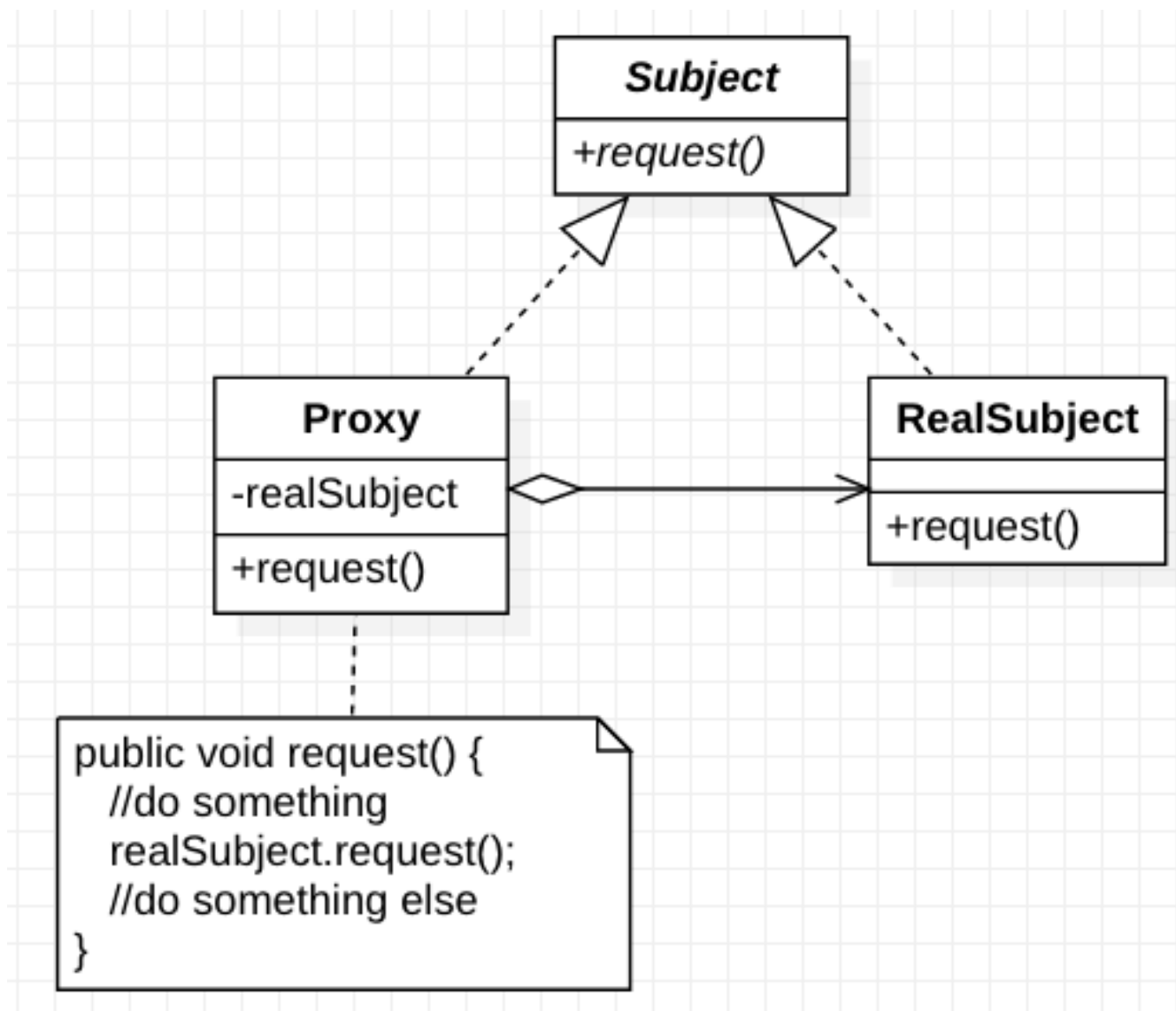


Figure 33: Class Diagram del pattern Proxy

### 3.2.2.2 Utilizzo e motivazioni

**Character**, **ProxyCharacter** e **RealCharacter** sono le tre componenti (*subject*, *proxy* e *realSubject*) di un *pattern Proxy*.

Questo pattern è stato utilizzato per permettere al gioco del giocatore locale di sincronizzarsi con il gioco del giocatore remoto ogni volta che lo stato di un personaggio viene modificato. Per fare ciò, il lavoro compiuto dal **ProxyCharacter** è diviso in tre fasi:

1. riceve le azioni che il **Game** vuole svolgere sul **RealCharacter**, facendo quindi da tramite tra questi due elementi;
2. se l'azione che il **Game** ha eseguito modifica lo stato del **RealCharacter**, il **ProxyCharacter** la codifica e invia il packet al **Network**, così da inviarlo tramite la rete al giocatore remoto;
3. infine, quando il **Network** riceve un packet dalla rete, il **ProxyCharacter** lo decodifica e applica le azioni corrispondenti sul **RealCharacter**, attuando la sincronizzazione.

### 3.2.3 Observer

#### 3.2.3.1 Dettagli

Il **pattern Observer** è un *Behavioral Pattern* che definisce una dipendenza uno-a-molti tra degli oggetti così che quando un oggetto modifica il suo stato, tutti gli oggetti che dipendono da esso ricevono una notifica e si aggiornano di conseguenza. I partecipanti a questo pattern sono:

- il **Subject** che è una classe astratta o un'interfaccia che conosce i suoi *Observer* e fornisce metodi per notificarli e aggiungerli o eliminarli dalla lista;
- i **ConcreteSubject** che sono ciò che viene osservato realmente dai *ConcreteObserver*;
- l'interfaccia **Observer** che definisce il metodo per fare l'aggiornamento;
- i **ConcreteObserver** che implementano il metodo per fare l'aggiornamento fornito dall'interfaccia *Observer* e mantengono un riferimento al **ConcreteSubject** che stanno osservando;

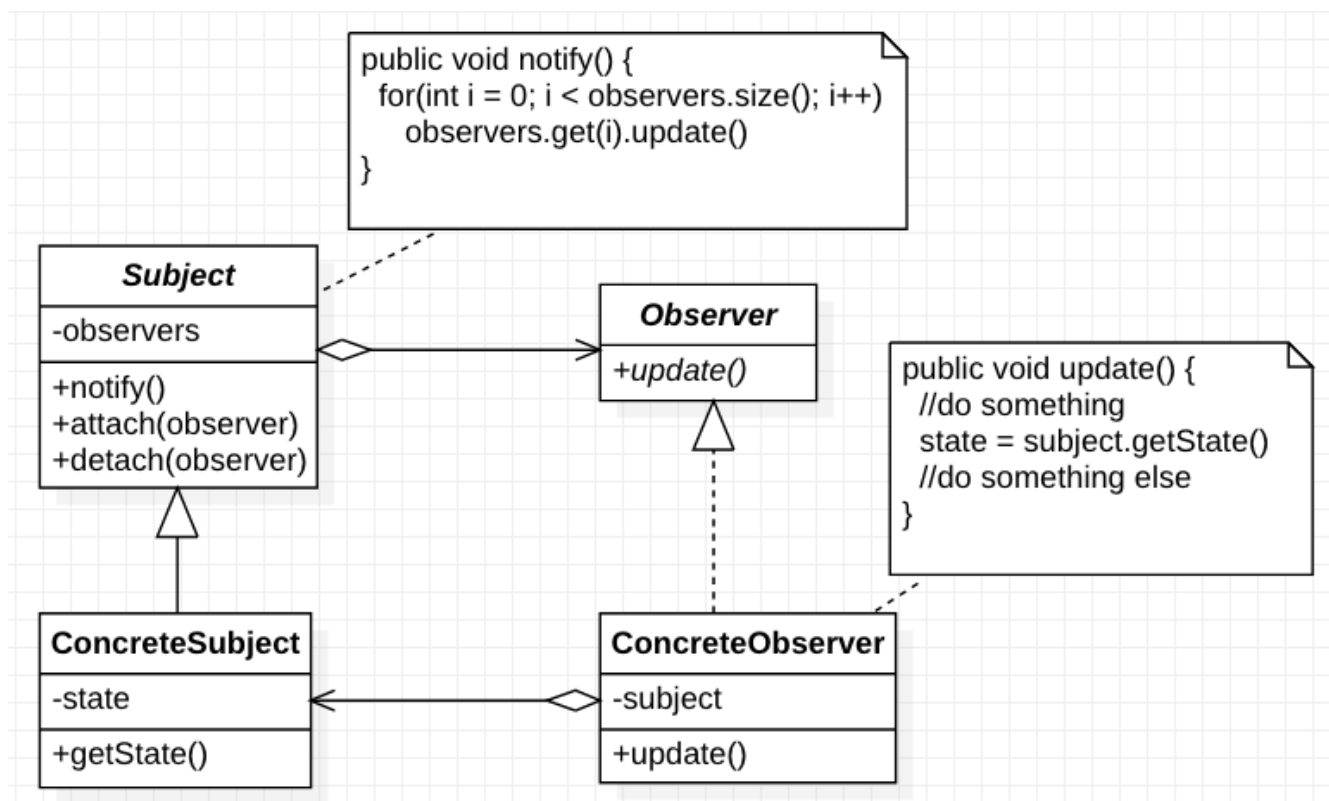


Figure 34: Class Diagram del pattern Observer

### 3.2.3.2 Utilizzo e motivazioni

**RealCharacter**, **Archer**, **Warrior**, **Mage**, **Dragon**, **Slime**, **Goblin**, **Observer**, **Game**, **StatusBar** e **GameMap** sono le componenti di un *patter Observer* di tipo *push*.

In particolare:

- **RealCharacter** è il subject;
- **Archer**, **Warrior**, **Mage**, **Dragon**, **Slime** e **Goblin** sono i concreteSubject;
- **Observer** è l'observer;
- **Game**, **StatusBar** e **GameMap** sono tre concreteObserver di **RealCharacter**.

Questo pattern è stato utilizzato perché il **Game**, la **StatusBar** e la **GameMap** devono essere costantemente aggiornati sullo stato di ogni **RealCharacter**.

Inoltre è qui che entra in gioco la mappa **CharactersByID**: tutte le classi Observer ricevono infatti l'id del **RealCharacter** nel metodo *update()* e lo utilizzano per osservare in realtà il **ProxyCharacter** corrispondente.

A differenza di come è stato descritto il pattern nella sezione *Dettagli*, quindi, i concreteObserver non hanno un riferimento diretto alle classi concreteSubject, ma ad un loro proxy.

```

48  @Override public void update(String characterID){
49      Character c = charactersByID.get( id: characterID);
50
51      if(!c.isCurrent() && c==currentCharacter){
52          currentCharacter = null;
53          startTurn();
54      }
55
56      if(!c.isAlive()){
57          if(myTeam.contains( o:c)){
58              myTeam.remove( o:c);
59              order.remove( o:c);
60              statusBar.setOrder(order);
61          }
62          if(enemyTeam.contains( o:c)){
63              enemyTeam.remove( o:c);
64              order.remove( o:c);
65              statusBar.setOrder(order);
66          }
67      }
68
69      if(myTeam.isEmpty() || enemyTeam.isEmpty()){
70          //chiede al FrontEnd di uscire dal programma
71      }

```

Figure 35: La funzione update() nella classe Game.

## 3.2.4 Command

### 3.2.4.1 Dettagli

Il patter **Command** è un *Behavioral Pattern* che incapsula una richiesta in un oggetto. I partecipanti a questo pattern sono:

- il **Command** che è un'interfaccia che dichiara il metodo *execute()*;
- il **Receiver** che deve svolgere l'azione richiesta;
- i **ConcreteCommand** che implementano l'interfaccia *Command* e possiedono un riferimento al *Receiver* per svolgere l'azione corrispondente;
- gli **Invoker** che chiedono ai *ConcreteCommand* di eseguire la richiesta;
- il **Client** che crea un oggetto *ConcreteCommand* e seleziona il suo *Receiver*.

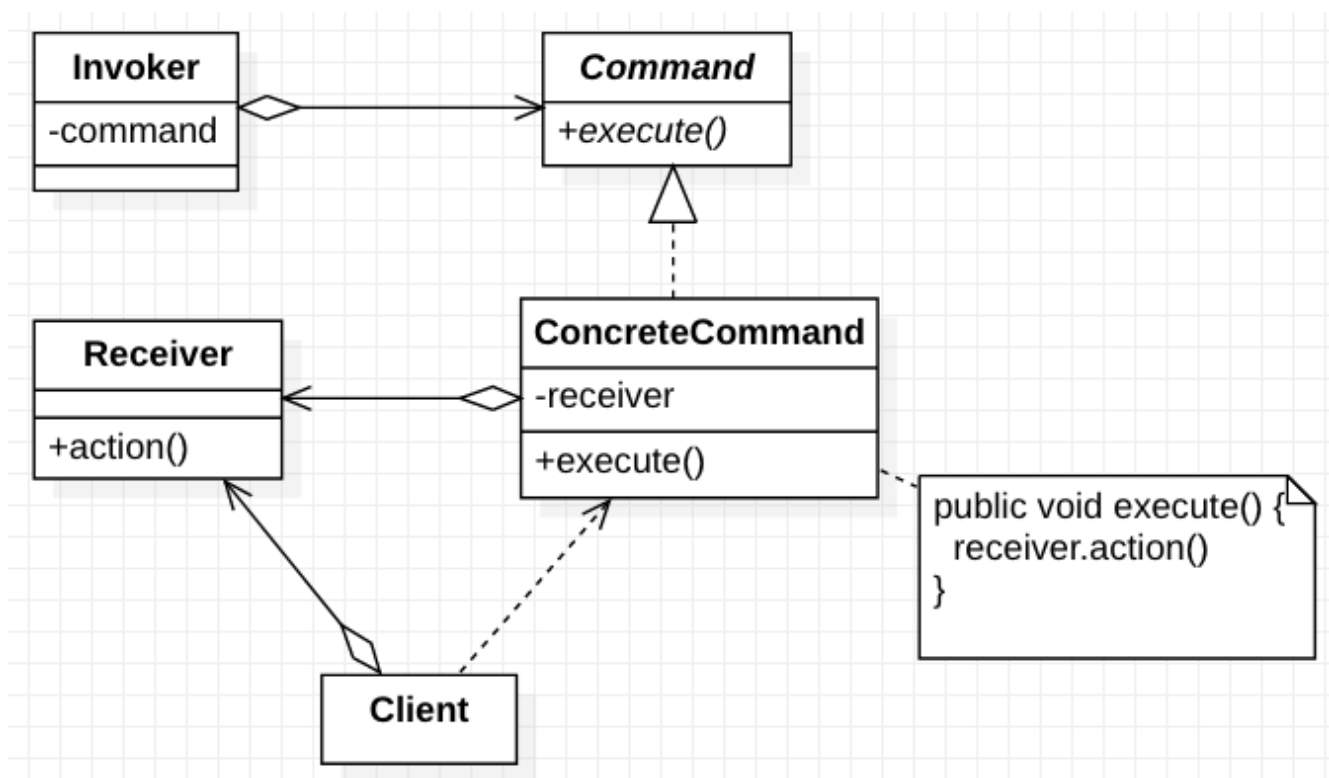


Figure 36: Class Diagram del pattern Command

### 3.2.4.2 Utilizzo e motivazioni

Ad ogni **MenuItem** è associata una differente **Action**, che si attiva quando il **MenuItem** viene "cliccato" (*onClicked()*). Quando un'azione viene eseguita (i.e.

viene eseguito il metodo *execute()*, si chiama l'azione corrispondente nella classe **Game**.

Questa è l'implementazione di un *pattern Command*, dove il **MenuItem** rappresenta l'elemento invoker, il **Game** è il receiver, l'interfaccia **Action** l'elemento command e le sue implementazioni sono i *concreteCommand*.

Si noti inoltre che nel nostro caso non vi è un client e che la creazione dei *concreteCommand* viene svolta direttamente dal receiver.

Questo pattern è stato utilizzato in quanto era utile poter scindere la classe del **MenuItem** ("il tasto fisico") dalla classe **Action** ("l'azione corrispondente al tasto"), permettendo maggiore flessibilità nella creazione di diversi tasti del **Menu**.



## 4 Unit Test svolti

In questa sezione vengono analizzati gli **Unit Test** svolti.

Per ogni classe di test si sono create delle *classi mock* per poter svolgere più facilmente i test.

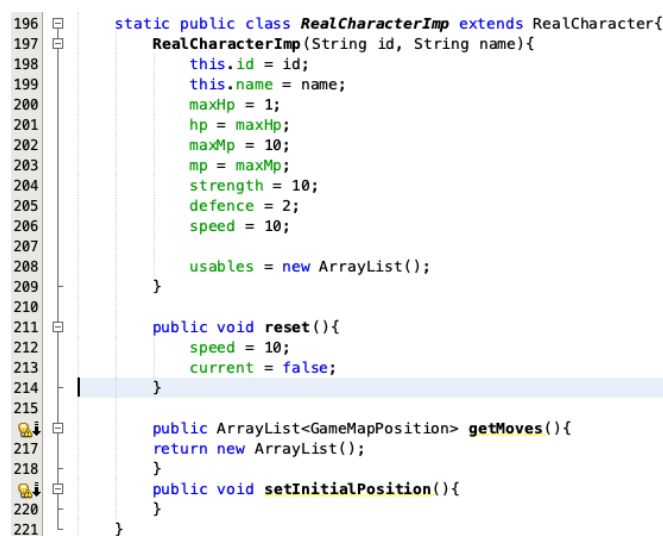
### 4.1 GameTest()

Per la classe **Game** sono stati testati i metodi di *update()* e di *onPass()*.

Si è deciso di testare solo questi due metodi e non anche gli altri in quanto:

- il metodo *update()* è fondamentale per il funzionamento dell'intero programma;
- i metodi *onStartGame()*, *startTurn()*, *startRound()* vengono testati attraverso il metodo *onPass()*;
- testando il metodo *onPass()* si testa anche il funzionamento della classe **PriorityQueueComparator**. A tal fine si sono svolti due test, per controllare se vengono soddisfatti entrambi i criteri di ordinamento (dal più veloce al più lento e, in caso di pareggio, in ordine alfabetico);
- i restanti metodi sono banali e la loro complessità dipende da metodi presenti in **GameMap**, che sono stati testati nella classe di test apposita.

All'inizio di ogni test i personaggi vengono resettati con una funzione di *reset()* implementato nella classe mock.



```
196 static public class RealCharacterImp extends RealCharacter{
197     RealCharacterImp(String id, String name){
198         this.id = id;
199         this.name = name;
200         maxHp = 1;
201         hp = maxHp;
202         maxMp = 10;
203         mp = maxMp;
204         strength = 10;
205         defence = 2;
206         speed = 10;
207
208         usables = new ArrayList();
209     }
210
211     public void reset(){
212         speed = 10;
213         current = false;
214     }
215
216     public ArrayList<GameMapPosition> getMoves(){
217         return new ArrayList();
218     }
219     public void setInitialPosition(){
220     }
221 }
```

Figure 37: La classe mock utilizzata, è stato definito il metodo *reset()* per resettare l'oggetto.

```

113
114
115
116
117
118
119
120
121
122
123
124
125
@Test
public void testUpdate() {
    System.out.println("update");
    assertTrue(condition: myTeam.contains( o: pH1));
    m1.attack( enemy: h1);
    assertFalse( condition: h1.isAlive());
    assertFalse( condition: myTeam.contains( o: pH1));

    assertTrue( condition: enemyTeam.contains( o: pM1));
    h2.attack( enemy: m1);
    assertFalse( condition: m1.isAlive());
    assertFalse( condition: enemyTeam.contains( o: pM1));
}

```

Figure 38: Il test del metodo update().

```

130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
@Test
public void test10nPass() {
    System.out.println("onPass: checkOrderQueue(differentSpeed)");
    h1.setSpeed( speed: 1);
    h2.setSpeed( speed: 2);
    h3.setSpeed( speed: 3);
    m1.setSpeed( speed: 4);
    m2.setSpeed( speed: 5);
    m3.setSpeed( speed: 6);

    assertFalse( condition: h1.isCurrent());
    assertFalse( condition: h2.isCurrent());
    assertFalse( condition: h3.isCurrent());
    assertFalse( condition: m1.isCurrent());
    assertFalse( condition: m2.isCurrent());
    assertFalse( condition: m3.isCurrent());

    g.onStartGame();
    assertTrue( condition: m3.isCurrent());
    g.onPass();
    assertFalse( condition: m3.isCurrent());
    assertTrue( condition: m2.isCurrent());
    g.onPass();
    assertFalse( condition: m2.isCurrent());
    assertTrue( condition: m1.isCurrent());
    g.onPass();
    assertFalse( condition: m1.isCurrent());
    assertTrue( condition: h3.isCurrent());
    g.onPass();
    assertFalse( condition: h3.isCurrent());
    assertTrue( condition: h2.isCurrent());
    g.onPass();
    assertFalse( condition: h2.isCurrent());
    assertTrue( condition: h1.isCurrent());
}

166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
@Test
public void test20nPass() {
    System.out.println("onPass: checkOrderQueue(sameSpeed)");

    assertFalse( condition: h1.isCurrent());
    assertFalse( condition: h2.isCurrent());
    assertFalse( condition: h3.isCurrent());
    assertFalse( condition: m1.isCurrent());
    assertFalse( condition: m2.isCurrent());
    assertFalse( condition: m3.isCurrent());

    g.onStartGame();
    assertTrue( condition: h1.isCurrent());
    g.onPass();
    assertFalse( condition: h1.isCurrent());
    assertTrue( condition: h2.isCurrent());
    g.onPass();
    assertFalse( condition: h2.isCurrent());
    assertTrue( condition: h3.isCurrent());
    g.onPass();
    assertFalse( condition: h3.isCurrent());
    assertTrue( condition: m1.isCurrent());
    g.onPass();
    assertFalse( condition: m1.isCurrent());
    assertTrue( condition: m2.isCurrent());
    g.onPass();
    assertFalse( condition: m2.isCurrent());
    assertTrue( condition: m3.isCurrent());
}

```

Figure 39: I due test per onPass(), in un caso i personaggi hanno velocità diversa, nell'altro hanno la stessa.

## 4.2 GameMapTest()

Per la classe **GameMap** sono stati testati tutti i metodi che devono selezionare le caselle su cui il giocatore può compiere una mossa. Si sono svolti questi test perché questi metodi rappresentano la parte più complessa per ogni azione compiuta dal giocatore (tranne l'azione "Pass" che è stata testata nel **Game**). Inoltre si nota che non si è svolto un test per il metodo *update()* perché quest'ultimo viene indirettamente controllato durante ogni test svolto.

In particolare si sono svolti i seguenti test:

- tre test per il metodo *selectMoves()*, per controllare se le mosse disponibili rispettano i vincoli di "un solo personaggio per casella" e "non uscire fuori dalla mappa". Questo è un test indiretto anche del metodo *isMovePossible()*;
- due test per il metodo *selectEnemies()*, per controllare sia il caso in cui non fosse possibile selezionare alcun nemico, sia quello in cui ci fossero sia nemici che alleati;
- due test per il metodo *selectAllies()*, per controllare sia il caso in cui non fosse possibile selezionare alcun alleato, sia quello in cui ci fossero sia nemici che alleati.

All'inizio di ogni test le posizioni dei personaggi vengono resettate.

```

202 static public class RealCharacterImp extends RealCharacter{
203     RealCharacterImp(String id, String name){
204         this.id = id;
205         this.name = name;
206         maxHp = 1;
207         hp = maxHp;
208         maxMp = 10;
209         mp = maxMp;
210         strength = 10;
211         defence = 2;
212         speed = 10;
213
214         usables = new ArrayList();
215     }
216
217     public ArrayList<GameMapPosition> getMoves(){
218         ArrayList<GameMapPosition> moves = new ArrayList();
219         return moves;
220     }
221
222     public void setInitialPosition(GameMapPosition pos){
223         this.pos = pos;
224         setInitialPosition();
225     }
226
227     public void setInitialPosition(){
228         notifyObservers();
229     }

```

Figure 40: La classe mock utilizzata, è stato fatto un overload del metodo setInitialPosition() per poter settare da fuori la posizione iniziale.

```

93 @Test
94 public void test1SelectMoves() {
95     System.out.println( x: "selectMoves: noObstacles");
96
97     h1.move(new GameMapPosition( x: 11, y: 8));
98
99     ArrayList<GameMapPosition> moves = map.selectMoves( c: h1);
100     ArrayList<GameMapPosition> expMoves = h1.getMoves();
101     assertEquals( expected: expMoves, actual: moves);
102 }
103
104 @Test
105 public void test2SelectMoves() {
106     System.out.println( x: "selectMoves: withObstacles");
107
108     h1.move(new GameMapPosition( x: 11, y: 8));
109
110     h2.move(new GameMapPosition( x: 13, y: 8));
111     m1.move(new GameMapPosition( x: 11, y: 10));
112     m2.move(new GameMapPosition( x: 10, y: 7));
113
114     ArrayList<GameMapPosition> moves = map.selectMoves( c: h1);
115     ArrayList<GameMapPosition> expMoves = h1.getMoves();
116     GameMapPosition h2Pos = h2.getPos();
117     GameMapPosition m1Pos = m1.getPos();
118     GameMapPosition m2Pos = m2.getPos();
119
120     expMoves.remove( o: h2Pos);
121     expMoves.remove( o: h2Pos);
122     expMoves.remove( o: m1Pos);
123
124     assertEquals( expected: expMoves, actual: moves);
125 }

```

```

127 @Test
128 public void test3SelectMoves(){
129     System.out.println( x: "selectMoves: outOfMap");
130     h1.move(new GameMapPosition( x: 19, y: 13));
131
132     ArrayList<GameMapPosition> moves = map.selectMoves( c: h1);
133     ArrayList<GameMapPosition> expMoves = h1.getMoves();
134
135     expMoves.remove(new GameMapPosition( x: 20, y: 12));
136     expMoves.remove(new GameMapPosition( x: 21, y: 13));
137     expMoves.remove(new GameMapPosition( x: 20, y: 14));
138     expMoves.remove(new GameMapPosition( x: 19, y: 15));
139
140     assertEquals( expected: expMoves, actual: moves);
141 }

```

Figure 41: I tre test per selectMove().

```

146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170

```

```

@Test
public void test1SelectEnemies() {
    System.out.println("selectEnemies: noEnemiesOrAllies");
    h1.move(new GameMapPosition( x:19, y:13));

    ArrayList<GameMapPosition> enemiesPos = map.selectEnemies( c:h1, enemies:Monsters);
    assertTrue( condition:enemiesPos.isEmpty());
}

@Test
public void test2SelectEnemies() {
    System.out.println("selectEnemies: withEnemiesAndAllies");
    h1.move(new GameMapPosition( x:19, y:13));

    h2.move(new GameMapPosition( x:19, y:12));
    m1.move(new GameMapPosition( x:18, y:13));
    m2.move(new GameMapPosition( x:19, y:14));

    ArrayList<GameMapPosition> enemiesPos = map.selectEnemies( c:h1, enemies:Monsters);
    ArrayList<GameMapPosition> expEnemiesPos = new ArrayList();
    expEnemiesPos.add( e:m1.getPos());
    expEnemiesPos.add( e:m2.getPos());
    assertTrue(enemiesPos.size()==expEnemiesPos.size());
    assertEquals( expected:expEnemiesPos, actual:enemiesPos);
}

```

Figure 42: I due test per selectEnemies()

```

175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200

```

```

@Test
public void test1SelectAllies() {
    System.out.println("selectAllies: noEnemiesOrAllies");
    h1.move(new GameMapPosition( x:19, y:13));

    ArrayList<GameMapPosition> alliesPos = map.selectAllies( c:h1, allies:Humans);
    assertTrue( condition:alliesPos.isEmpty());
}

@Test
public void test2SelectAllies() {
    System.out.println("selectAllies: withEnemiesAndAllies");
    h1.move(new GameMapPosition( x:19, y:13));

    h2.move(new GameMapPosition( x:19, y:12));
    m1.move(new GameMapPosition( x:18, y:13));
    m2.move(new GameMapPosition( x:19, y:14));

    ArrayList<GameMapPosition> alliesPos = map.selectAllies( c:h1, allies:Humans);
    ArrayList<GameMapPosition> expAlliesPos = new ArrayList();
    expAlliesPos.add( e:h2.getPos());
    expAlliesPos.add( e:h1.getPos());

    assertTrue(expAlliesPos.size() == alliesPos.size());
    assertEquals( expected:expAlliesPos, actual:alliesPos);
}

```

Figure 43: I due test per selectAllies()

## 4.3 Test su oggetti e abilità

Si sono svolti infine dei test su ogni oggetto e abilità per controllarne il corretto funzionamento.

Per ogni classe è stato svolto almeno un test del metodo *use()* (due nel caso di **Cure** e **Potion** per controllare che gli HP dopo l'utilizzo non aumentassero oltre il limite massimo).

Si noti che, grazie a questi test, si è testato indirettamente anche il metodo *canUse()* sia di **ObjectBase** che di **AbilityBase**.

```
47 public class Human extends RealCharacter {
48     public Human(ObjectOrAbility obj){
49         id = "H00";
50         name = "Human";
51         maxHp = 30;
52         hp = maxHp;
53         maxMp = 10;
54         mp = maxMp;
55         strength = 10;
56         defence = 10;
57         speed = 10;
58         pos = new GameMapPosition( x:0, y:0);
59
60         usables = new ArrayList();
61         usables.add( e:obj);
62     }
63
64     public ArrayList<GameMapPosition> getMoves(){
65         ArrayList<GameMapPosition> moves = new ArrayList();
66         return moves;
67     }
68     public void setInitialPosition(){
69
70     }
71 }
```

```
34 @Test
35 public void testUse() {
36     System.out.println( x:"use");
37     FortifyingLotion lotion= new FortifyingLotion( quantity:1);
38     Human human = new Human( obj:lotion);
39     assertTrue( condition:lotion.canUse( c:human));
40     human.use( usable:lotion, ally:human);
41     assertEquals( expected:14, actual:human.getDefence());
42
43     assertFalse( condition:lotion.canUse( c:human));
44 }
```

Figure 44: LotionTest(): la classe mock utilizzata e il test di use().

```
61 public class Monster extends RealCharacter {
62     public Monster(int hp, ObjectOrAbility ability){
63         id = "M00";
64         name = "Monster";
65         maxHp = 20;
66         this.hp = hp;
67         maxMp = 10;
68         mp = maxMp;
69         strength = 10;
70         defence = 10;
71         speed = 10;
72         pos = new GameMapPosition( x:0, y:0);
73
74         usables = new ArrayList();
75         usables.add( e:ability);
76     }
77
78     public ArrayList<GameMapPosition> getMoves(){
79         ArrayList<GameMapPosition> moves = new ArrayList();
80         return moves;
81     }
82     public void setInitialPosition(){
83
84     }
85 }
```

```
34 @Test
35 public void test1Use() {
36     System.out.println( x:"use: notOverMaxHp");
37     Cure cure = new Cure();
38     Monster monster = new Monster( hp:10, ability:cure);
39     assertTrue( condition:cure.canUse( c:monster));
40     monster.use( usable:cure, ally:monster);
41     assertEquals( expected:10+monster.getHp()*20/100, actual:monster.getHp());
42
43     assertEquals(monster.getMaxMp()-10, actual:monster.getMp());
44     assertFalse( condition:cure.canUse( c:monster));
45 }
46
47 @Test
48 public void test2Use() {
49     System.out.println( x:"use: overMaxHp");
50     Cure cure = new Cure();
51     Monster monster = new Monster( hp:19, ability:cure);
52     assertTrue( condition:cure.canUse( c:monster));
53     monster.use( usable:cure, ally:monster);
54     assertEquals( expected:monster.getMaxMp(), actual:monster.getHp());
55
56     assertEquals(monster.getMaxMp()-10, actual:monster.getMp());
57     assertFalse( condition:cure.canUse( c:monster));
58 }
```

Figure 45: CureTest(): la classe mock utilizzata e i due test di use().