

Elaborato di Ingegneria del Software

PGgenerator

Tommaso Vannozzi
matricola 6356306

1 Settembre 2021



UNIVERSITÀ
DEGLI STUDI
FIRENZE

1 Intento e descrizione pratica

L'intento di questo elaborato è quello di creare un generatore di personaggi di giochi di ruolo. Tramite l'utilizzo di Design Pattern differenti è stato possibile creare famiglie di prodotti eterogenei, personalizzabili e unici.

2 Design Pattern

Design pattern in informatica e specialmente nell'ambito dell'ingegneria del software, è un concetto che può essere definito "una soluzione progettuale generale ad un problema ricorrente". Si tratta di una descrizione o modello logico da applicare per la risoluzione di un problema che può presentarsi in diverse situazioni durante le fasi di progettazione e sviluppo del software, ancor prima della definizione dell'algoritmo risolutivo della parte computazionale. È un approccio spesso efficace nel contenere o ridurre il debito tecnico.

2.1 Abstract Factory

Presenta un'interfaccia per la creazione di famiglie di prodotti, in modo tale che il Client che le utilizza non abbia conoscenza delle loro classi concrete. Definisce quindi un'interfaccia per creare istanze di classi dipendenti tra loro senza conoscenza della loro classe effettiva. I partecipanti sono:

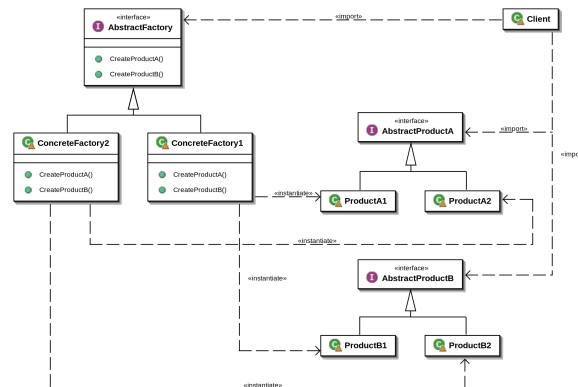


Figura 1: ABSTRACT FACTORY

- **AbstractFactory**: Dichiara un'interfaccia per le operazioni che creano oggetti Product astratti.
- **ConcreteFactory**: Implementa le operazioni di creare oggetti Product concreti.
- **AbstractProduct**: Dichiara un'interfaccia per ogni tipo di oggetto Product.
- **ConcreteProduct**: Definisce un oggetto Product per essere creato dal corrispondente ConcreteFactory. Implementa l'interfaccia AbstractProduct.
- **Client**: Usa solamente le interfacce dichiarate dalle classi AbstractFactory e AbstractProduct.

2.2 Singleton

Il Singleton rappresenta un tipo particolare di classe che garantisce che soltanto un'unica istanza della classe stessa possa essere creata all'interno di un programma. Per ottenere questo è necessario avvalersi di un costruttore privato ed utilizzare un metodo statico che consenta di accedere all'unica istanza della classe.

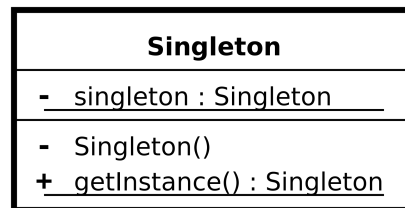


Figura 2: SINGLETON

2.3 Observer

Il pattern Observer permette di definire una dipendenza uno a molti tra oggetti, in modo tale che se un oggetto cambia il suo stato interno, ciascuno degli oggetti dipendenti da esso viene notificato e aggiornato automaticamente. Esso trova applicazione nei casi in cui diversi oggetti devono conoscere lo stato di un soggetto, di cui vengono “osservati” i cambiamenti.

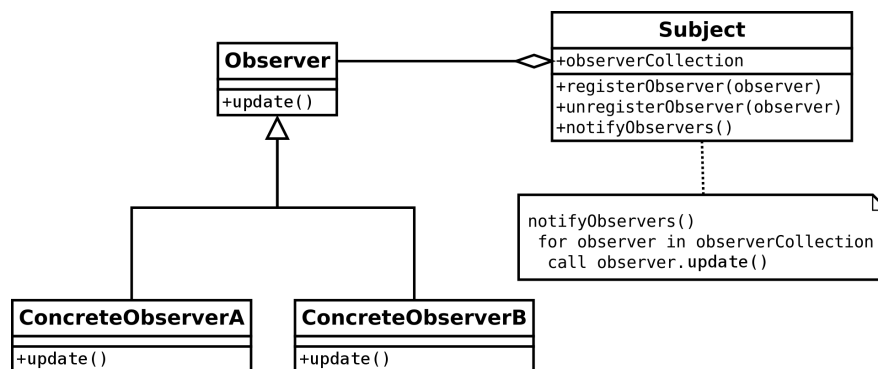


Figura 3: OBSERVER

- **Subject:** Ha conoscenza dei propri Observer e fornisce operazioni per l'aggiunta, la cancellazione e la notifica agli Observer.
- **ConcreteSubject:** Mantiene lo stato del soggetto osservato e notifica agli observer in caso di un cambio di stato e chiama le operazioni di notifica ereditate dal Subject, quando devono essere informati i ConcreteObserver.
- **Observer:** Specifica un'interfaccia per la notifica di eventi agli oggetti interessati in un subject.
- **ConcreteObserver:** Specifica un'interfaccia per la notifica di eventi agli oggetti interessati in un subject.

2.4 Decorator

Aggiunge dinamicamente responsabilità aggiuntive ad un oggetto. In questo modo si possono estendere le funzionalità d'oggetti particolari senza coinvolgere complete classi. Permette quindi di aggiungere o modificare dinamicamente il comportamento di un oggetto. attraverso ripetute composizioni.

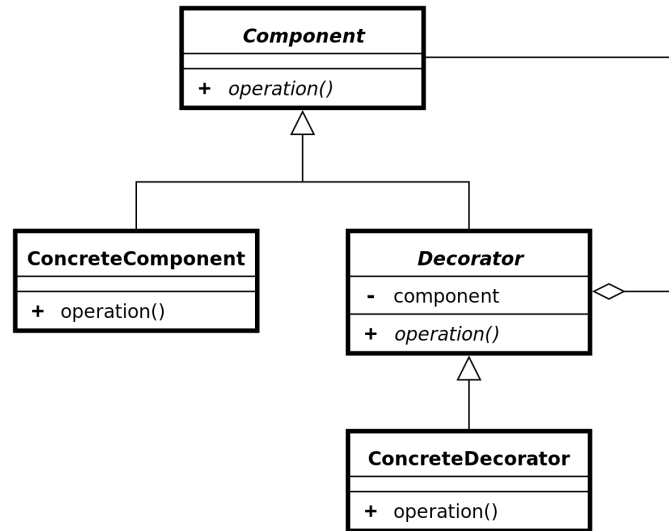


Figura 4: Decorator

- **AbstractComponent**: Specifica l'interfaccia degli oggetti che possono avere delle responsabilità aggiunte dinamicamente.
- **ConcreteComponent**: Implementa l'oggetto nel quale si possono aggiungere nuove responsabilità.
- **AbstractDecorator**: Possiede un riferimento all'oggetto `Component` e specifica un'interfaccia concordante con l'interfaccia `Component`.
- **ConcreteDecorator**: Aggiunge nuove responsabilità al `Component`.

3 Implementazione e Descrizione delle classi

3.1 AbstractFactory

L'implementazione del DesignPattern AbstractFactory prevede la creazione di 9 classi che sono:

- **PGFactory**: Interfaccia che delega la creazione degli oggetti astratti ConcreteHumanFactory e ConcreteElfFactory.

```
1      import character.AbstractCharacter;
2      import pet.AbstractPet;
3
4      //Definizione della AbstractFactory
5      public interface PGFactory {
6          public abstract AbstractCharacter createCharacter();
7          public abstract AbstractPet createPet();
8      }
```

- **ConcreteHumanFactory**: Implementa le operazioni per creare oggetti concreti. Nel nostro caso abbiamo due metodi pubblici specificheranno i prodotti facendo l'override dei singoli factory method presenti nell'interfaccia.

```
1
2      import character.AbstractCharacter;
3      import character.ConcreteHuman;
4      import pet.AbstractPet;
5      import pet.ConcreteHorse;
6
7      //Definizione della ConcreteFactory per gli Umani
8      public class ConcreteHumanFactory implements PGFactory {
9
10         @Override
11         public AbstractCharacter createCharacter() {
12             return new ConcreteHuman();
13         }
14
15         @Override
16         public AbstractPet createPet() {
17             return new ConcreteHorse();
18         }
19     }
20
```

- **ConcreteElfFactory**:Analogo al precedente.

```
1      import character.AbstractCharacter;
2      import character.ConcreteElf;
3      import pet.AbstractPet;
4      import pet.ConcreteFairy;
5
6
7      //Definizione della ConcreteFactory per gli Elfi
8      public class ConcreteElfFactory implements PGFactory {
9
10         @Override
11         public AbstractCharacter createCharacter() {
12             return new ConcreteElf();
13         }
14
15         @Override
16         public AbstractPet createPet() {
17             return new ConcreteFairy();
18         }
19
20     }
```

- **AbstractPet**: Interfaccia utilizzata per la creazione di oggetti di tipo Pet.

```
1      package pet;
2
3      //Definizione della interfaccia per la creazione dei Pet
4      public interface AbstractPet {
5          public int getHP();
6          public int getSpeed();
7          public String getRace();
8          public int getStat();
9
10     }
```

- **ConcreteFairy** La classe concreteFairy andrà ad implementare l'interfaccia di AbstractPet. Risulta essere il nostro concreteProduct. Qui verranno implementati i metodi ed inizializzati gli attributi della classe.

```
1 package pet;
2
3 //Implementazione della interfaccia AbstracPet
4 public class ConcreteFairy implements AbstractPet{
5
6     private final int hp = 10;
7     private final int speed = 25;
8     private final String race = "Fairy";
9
10    public int getHP() {
11        return this.hp;
12    }
13
14    public int getSpeed() {
15        return this.speed;
16    }
17
18    public String getRace() {
19        return this.race;
20    }
21
22    public int getStat() {
23        return this.hp + this.speed;
24    }
25 }
```

- **ConcreteHorse**: Analogo al precedente.

```
1 //Implementazione della interfaccia AbstracPet
2 public class ConcreteHorse implements AbstractPet {
3
4     private final int hp = 20;
5     private final int speed = 15;
6     private final String race = "Horse";
7
8     public int getHP() {
9         return this.hp;
10    }
11
12    public int getSpeed() {
13        return this.speed;
14    }
15
16    public String getRace() {
17        return this.race;
18    }
19
20    public int getStat() {
21        return this.hp + this.speed;
22    }
23 }
```

- **AbstractCharacter:** Interfaccia utilizzata per la creazione di oggetti di tipo Character. Inoltre ha la funzione di Abstract Component per il Design Pattern Decorator.

```
1
2 package character;
3
4 //Interfaccia per la creazione dei Character
5 public interface AbstractCharacter{
6
7     public int getHP();
8     public int getStat();
9     public int getAttack();
10    public int getSpeed();
11    public int getArmor();
12    public String getRace();
13
14    // public ArrayList<String> getEquipments();
15 }
```

- **ConcreteElf:** La classe ConcreteElf andrà ad implementare l'interfaccia AbstractCharacter. E' il secondo ConcreteProduct che abbiamo. Qui verranno implementati i metodi e inizializzati gli attributi della classe.

```
1 public class ConcreteElf implements AbstractCharacter {
2
3     private final int hp = 15;
4     private final int attack = 12;
5     private final int speed = 20;
6     private final int armor = 10;
7     private final String race = "Elf";
8
9     public int getHP() {
10         return this.hp;
11     }
12
13     public int getAttack() {
14         return this.attack;
15     }
16
17     public int getSpeed() {
18         return this.speed;
19     }
20
21     public int getArmor() {
22         return this.armor;
23     }
24
25     public int getStat() {
26         return this.armor + this.attack + this.hp + this.speed;
27     }
28
29     public String getRace() {
30         return this.race;
31     }
32 }
```


- **ConcreteHuman:** Analogo al precedente.

```
1 package character;
2
3 public class ConcreteHuman implements AbstractCharacter {
4
5     private final int hp = 25;
6     private final int attack = 12;
7     private final int speed = 20;
8     private final int armor = 10;
9     private final String race = "Human";
10
11     public int getHP() {
12         return this.hp;
13     }
14
15     public int getAttack() {
16         return this.attack;
17     }
18
19     public int getSpeed() {
20         return this.speed;
21     }
22
23     public int getArmor() {
24         return this.armor;
25     }
26
27     public int getStat() {
28         return this.armor + this.attack + this.hp + this.speed;
29     }
30
31     public String getRace() {
32         return this.race;
33     }
34
35 }
```

3.2 Decorator

L'implementazione del DesignPattern Decorator prevede la creazione di altre classi che sono:

- **EquipmentCharacter:** Svolge il ruolo di abstractDecorator. Possiede un riferimento all'oggetto AbstractCharacter e ne specifica l'interfaccia concordante.

```
1 package character.DecoratorCharacter;
2
3 import character.AbstractCharacter;
4
5 //Creazione del AbstractDecorator
6 public abstract class EquipmentCharacter implements AbstractCharacter {
7
8     protected AbstractCharacter character;
9
10    public EquipmentCharacter(AbstractCharacter character) {
11        this.character=character;
12    }
13
14    public int getHP () {
15        return character.getHP ();
16    }
17
18    public int getAttack () {
19        return character.getAttack ();
20    }
21
22    public int getSpeed () {
23        return character.getSpeed ();
24    }
25
26    public int getArmor () {
27        return character.getArmor ();
28    }
29
30    public int getStat () {
31        return character.getStat ();
32    }
33
34    public String getRace () {
35        return character.getRace ();
36    }
37
38 }
```

- **ConcreteDecorator:** La classe seguente BootsFire è un esempio di concreteDecorator. E' una classe che aggiunge nuove funzionalità, in questo caso una modifica delle statistiche, al Component. Le altre classi rimanenti permettono di variare le statistiche del personaggio. Sono analoghe a BootsFire differenziate solo dal valore dei parametri.

```
1 package character.DecoratorCharacter.armor;
2 import character.AbstractCharacter;
3 import character.DecoratorCharacter.EquipmentCharacter;
4
5 public class BootsFire extends EquipmentCharacter {
6     private final int attack = 7;
7     private final int speed = 10;
8
9     public BootsFire (AbstractCharacter character){
10         super(character);
11     }
12
13     public int getHP () {
14         return super.getHP ();
15     }
16
17     public int getAttack () {
18         return super.getAttack () + this.attack;
19     }
20
21     public int getSpeed () {
22         return super.getSpeed () + this.speed;
23     }
24
25     public int getArmor () {
26         return super.getArmor ();
27     }
28
29     public int getStat () {
30         return super.getStat () + this.attack + this.speed;
31     }
32
33     public String getRace () {
34         return super.getRace ();
35     }
36 }
```

3.3 Singleton

L'implementazione del Singleton è stata effettuata su due classi che sono:

- **CharacterList**: Classe che permette la creazione di una lista di oggetti di tipo Character. All'interno vi è il metodo statico getObject che permetta la unica istanza della classe.

```
1 import observer.Subject;
2
3 public class CharacterList extends Subject{
4
5     private static CharacterList tmp = null;
6     private ArrayList<EquipmentCharacter> characters;
7
8     private CharacterList(){
9         characters = new ArrayList<>();
10    }
11
12    public static CharacterList getObject(){
13        if (tmp == null) {
14            tmp = new CharacterList();
15        }
16        return tmp;
17    }
18
19    public void addCharacter(EquipmentCharacter character){
20        this.characters.add(character);
21        this.notifyObservers();
22    }
23
24    public void remove(int index){
25        this.characters.remove(index);
26        this.notifyObservers();
27    }
28
29
30    public String toString(){
31
32        String s = "";
33
34        for(EquipmentCharacter tmp : this.characters){
35
36            s += "RIEPILOGO PERSONAGGIO:\n";
37            s += "\n";
38            s += "RAZZA: " + tmp.getRace() + "\n";
39            s += "HP: " + tmp.getHP() + "\n";
40            s += "SPEED: " + tmp.getSpeed() + "\n";
41            s += "ARMOR: " + tmp.getArmor() + "\n";
42            s += "ATTACK: " + tmp.getAttack() + "\n \n";
43            s += "STAT: " + tmp.getStat() + "\n";
44        };
45
46        return s;
47    }
48
49 }
```

- **PetList**: Classe analoga a **CharacterList**. In questo caso però abbiamo una lista di oggetti di tipo **Pet**.

```
1 import observer.Subject;
2 import pet.AbstractPet;
3
4 public class PetList extends Subject{
5
6     private static PetList tmp = null;
7     private ArrayList<AbstractPet> pets;
8
9     private PetList(){
10         pets = new ArrayList<>();
11     }
12
13     public static PetList getObject(){
14         if (tmp == null) {
15             tmp = new PetList();
16         }
17         return tmp;
18     }
19
20     public void addPet(AbstractPet pet){
21         this.pets.add(pet);
22         this.notifyObservers();
23     }
24
25     public void remove(int index){
26         this.pets.remove(index);
27         this.notifyObservers();
28     }
29
30     public String toString(){
31
32         String s = "";
33
34         for(AbstractPet tmp: this.pets){
35
36             s = "RIEPILOGO PET:\n";
37             s += "\n";
38             s += "RAZZA: " + tmp.getRace() + "\n";
39             s += "HP: " + tmp.getHP() + "\n";
40             s += "SPEED: " + tmp.getSpeed() + "\n";
41
42         };
43         return s;
44     }
45
46 }
```

3.4 Observer

L'implementazione dell' Observer prevede 3 classi che sono:

- **Subject:** Classe astratta che espone i metodi attach, detach e notifyObserver. Mantiene un ArrayList di osservatori a cui notifica cambiamenti di stato attraverso l'ultimo metodo sopracitato.

```
1 package observer;
2 import java.util.*;
3
4 public abstract class Subject {
5
6     private ArrayList<Observer> observers;
7
8     public Subject() {
9         this.observers = new ArrayList<>();
10    }
11
12    public void attach(Observer obs) {
13        this.observers.add(obs);
14    }
15
16    public void detach(Observer obs) {
17        this.observers.remove(obs);
18    }
19
20    public void notifyObservers() {
21        for(Observer obs : this.observers) {
22            obs.update(this);
23        }
24    }
25 }
26
27 }
```

- **Observer:** Interfaccia che definisce il metodo update. E' una classe astratta ed espone i metodi necessari per essere aggiornata dal Subject.

```
1 package observer;
2
3 public interface Observer {
4
5     public void update(Subject subject);
6
7 }
```

- **ConcreteObserver:** Implementa l'interfaccia Observer e il metodo necessario per aggiornare il Subject.

```
1 package observer;
2 public class ConcreteObserver implements Observer {
3
4     public void update(Subject subject){
5
6         System.out.println(subject);
7         System.out.println("Press To Show Next Character...");
8         new java.util.Scanner(System.in).nextLine();
9     }
10 }
```

4 Documentazione

4.1 UML Diagram

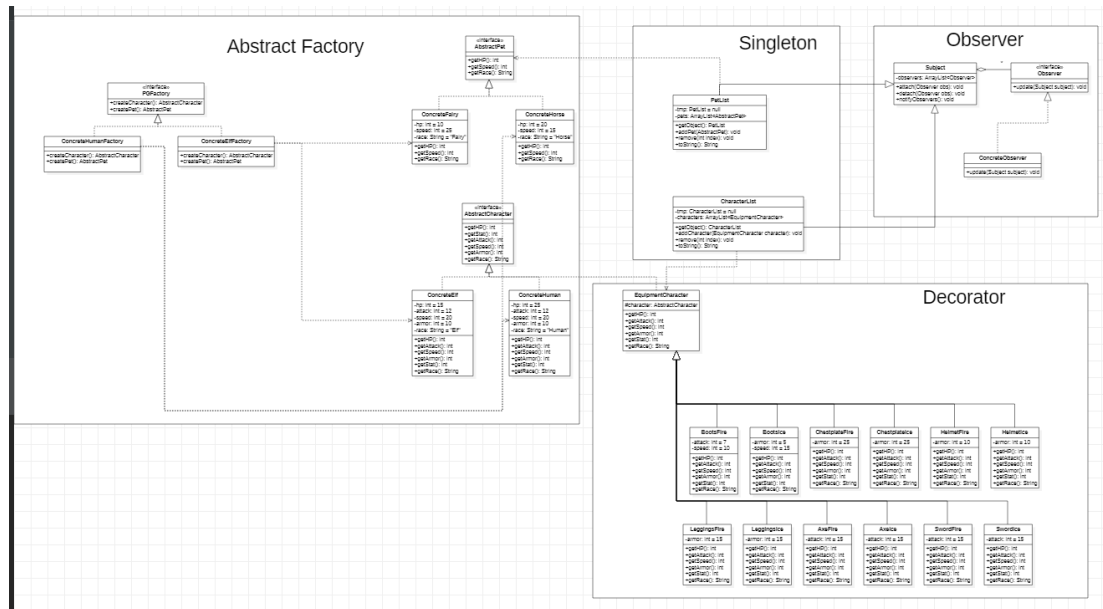


Figure 5: UML Diagram

4.2 Use Case Diagram

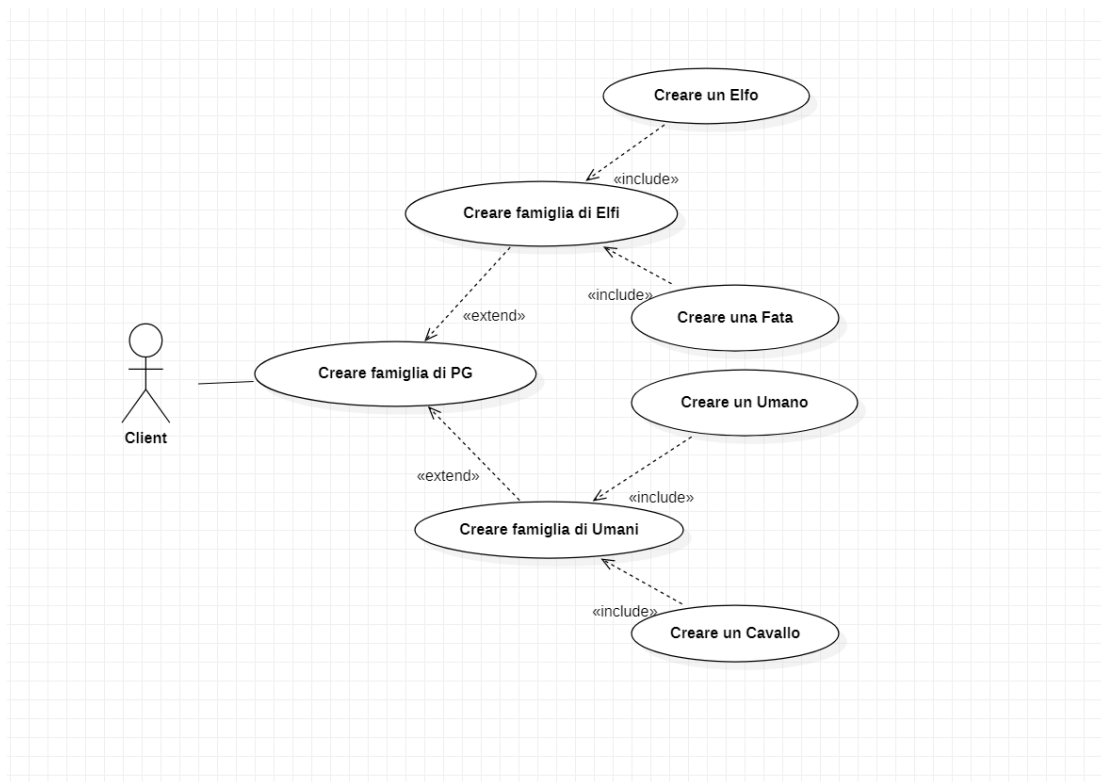


Figura 6: Use Case Diagram

4.3 Sequence Diagram

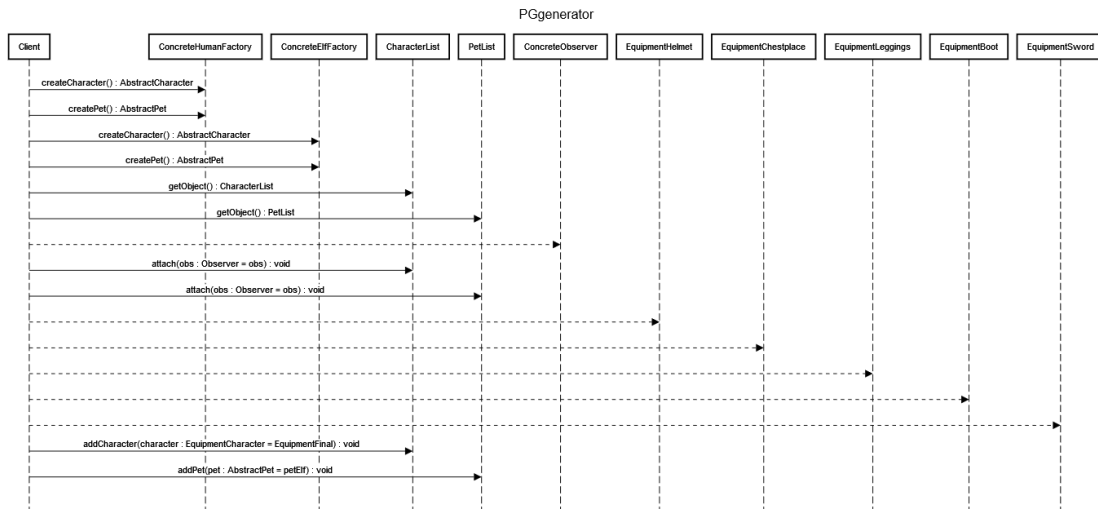


Figura 7: Sequence Diagram

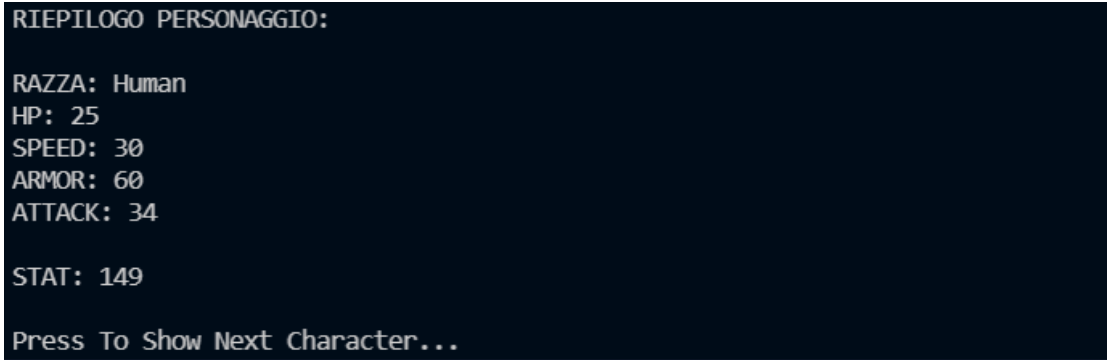
5 Test

5.1 PGFactory Test

```
1 public class PGFactoryTest {
2
3
4     @Test
5     public void test() {
6
7         //Istanza della famiglia Humam
8         PGFactory human = new ConcreteHumanFactory();
9         AbstractCharacter characterHuman = human.createCharacter();
10
11        //Test parametri Human
12        assertEquals(characterHuman.getHP(), 25);
13        assertEquals(characterHuman.getAttack(), 12);
14        assertEquals(characterHuman.getSpeed(), 20);
15        assertEquals(characterHuman.getArmor(), 10);
16        assertEquals(characterHuman.getStat(), 67);
17
18        //Creazione pet Human
19        AbstractPet petHuman = human.createPet();
20
21        //Test parametri Horse
22        assertEquals(petHuman.getHP(), 20);
23        assertEquals(petHuman.getSpeed(), 15);
24        assertEquals(petHuman.getStat(), 35);
25
26        //Costruzione Equipaggiamento del Character Human tramite decorator
27        EquipmentCharacter equipmentHelmet = new HelmetIce(characterHuman);
28        assertEquals(equipmentHelmet.getArmor(), 20);
29        assertEquals(equipmentHelmet.getStat(), 77);
30
31
32        EquipmentCharacter equipmentChestplace = new ChestplateIce(equipmentHelmet);
33        assertEquals(equipmentChestplace.getArmor(), 45);
34        assertEquals(equipmentChestplace.getStat(), 102);
35
36        EquipmentCharacter equipmentLegging = new LeggingsIce(equipmentChestplace);
37        assertEquals(equipmentLegging.getArmor(), 60);
38        assertEquals(equipmentLegging.getStat(), 117);
39
40        EquipmentCharacter equipmentBoot = new BootsFire(equipmentLegging);
41        assertEquals(equipmentBoot.getArmor(), 60);
42        assertEquals(equipmentBoot.getSpeed(), 30);
43        assertEquals(equipmentBoot.getStat(), 134);
44
45        EquipmentCharacter equipmentFinalHuman = new SwordFire(equipmentBoot);
46
47        assertEquals(equipmentFinalHuman.getStat(), 149);
48    }
49 }
50 }
```

6 Output

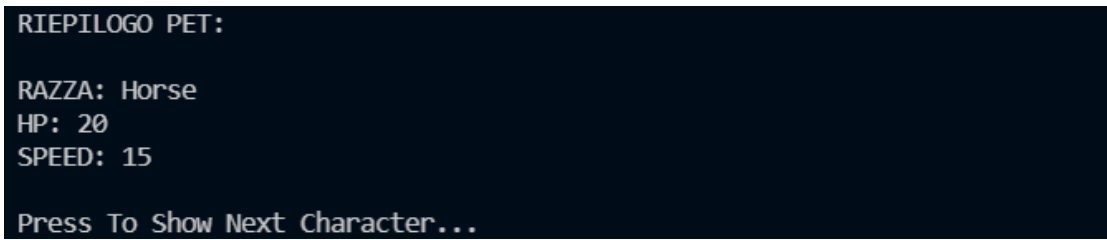
6.1 Human



```
RIEPILOGO PERSONAGGIO:  
  
RAZZA: Human  
HP: 25  
SPEED: 30  
ARMOR: 60  
ATTACK: 34  
  
STAT: 149  
  
Press To Show Next Character...
```

Figura 8: Human

6.2 Horse



```
RIEPILOGO PET:  
  
RAZZA: Horse  
HP: 20  
SPEED: 15  
  
Press To Show Next Character...
```

Figura 9: Horse

6.3 Elf

```
RIEPILOGO PERSONAGGIO:  
  
RAZZA: Elf  
HP: 15  
SPEED: 30  
ARMOR: 60  
ATTACK: 34  
  
STAT: 139  
  
Press To Show Next Character...
```

Figura 10: Elf

6.4 Fairy

```
RIEPILOGO PET:  
  
RAZZA: Fairy  
HP: 10  
SPEED: 25  
  
Press To Show Next Character...
```

Figura 11: Fairy