# Rappresentazione di discendenze

Marco Benelli

Maggio 2020

## 1 Definizione del problema

Vogliamo creare un'applicazione per tenere conto di parentele fra persone ed essere in grado di visualizzare tutti i discendenti di una persona. In particolare:

- nella rappresentazione di ciascuna persona dobbiamo mettere il nome, il cognome e, opzionalmente, anche luoghi e date di nascita e morte;
- il modello deve permettere che una persona si sia sposata più di una volta e da ciascun matrimonio abbia avuto un certo numero di figli.

## 2 Progettazione e implementazione

### 2.1 Le classi Person e Marriage

Per tenere conto delle parentele fra le persone abbiamo bisogno di una classe `Person` e una classe `Marriage` perché vogliamo tenere conto dei figli soltanto in un oggetto (quindi non in entrambi i genitori).

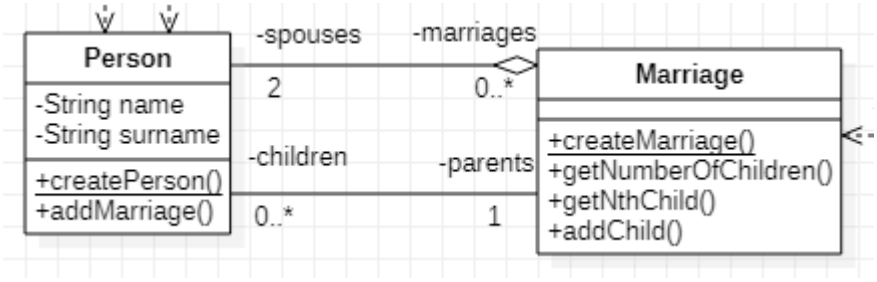


Figure 1: Il class diagram delle classi Person e Marriage

Vengono usati dei metodi statici per la creazione di oggetti di queste classi per fare cose di questo tipo:

```
1 public class Person {
2     public static Person createPerson(
```

```
3               String name, String surname, Marriage parents) {
4           Person p = new Person(name, surname, parents);
5           parents.addChild(p);
6           return p;
7       }
8   }
```

In particolare, la linea 5 dentro al costruttore diventerebbe `parents.addChild(this)`, ma non possiamo passare a `parents` un `Person` non ancora del tutto inizializzato.

Visto che alcune delle persone potrebbero essere ancora vive, i valori riferiti alla morte della persona potrebbero essere indeterminati. Lasciamo quindi a `null` i campi relativi alla morte se la persona è ancora viva, ma nel getter ritorno un tipo `Optional` per obbligare il cliente che lo usa a tenere conto del caso in cui la persona sia ancora viva.[1]

## 2.2 L'interfaccia PersonRepresentator e le sue implementazioni

Per rappresentare ogni persona come una stringa viene usato un `PersonRepresentator` che implementa lo schema del pattern composite. Questo per avere un modo di rappresentare ogni campo di ciascuna persona in maniera indipendente. Le leaves del pattern composite sono omesse dal class diagram perché nel codice vengono usate sempre classi anonime.

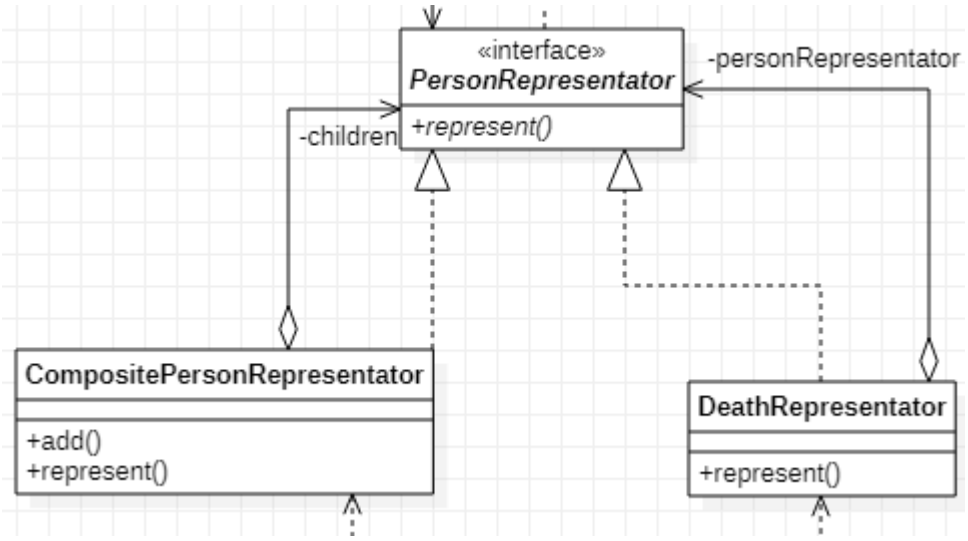


Figure 2: L'interfaccia PersonRepresentator e le sue implementazioni

La classe `DeathRepresentator` è usata per poter tenere conto del fatto che non tutte le persone sono morte. Segue il suo codice.

```java
public class DeathRepresentator implements PersonRepresentator {

    private final PersonRepresentator personRepresentator;

    public DeathRepresentator(PersonRepresentator personRepresentator) {
        this.personRepresentator = personRepresentator;
    }

    @Override
    public String represent(Person p) {
        if (p.getDeathDate().isPresent()) {
            return personRepresentator.represent(p);
        } else {
            return "";
        }
    }

}
```

### 2.3 La classe PersonPrinter

La classe `PersonPrinter` utilizza un `CompositePersonRepresentator` per stampare le persone e le loro relazioni. Un esempio di rappresentazione di una famiglia è il seguente.

```
  / Persona
--|
| \ Prima moglie
|
|---/ Primo figlio dal primo matrimonio
| --|
| | \ Nuora1
| |
| |---- Nipote1
| |
| |---- Nipote2
|
|---- Secondo figlio dal primo matrimonio
--|
| \ Seconda moglie
|
|---/ Primo figlio dal secondo matrimonio
  --|
  | \ Nuora2
  |
```

```
|-- Nipote3
```

La classe `PersonPrinter` è riportata nel seguito. Il metodo che fa la maggior parte del lavoro per quanto riguarda la rappresentazione delle parentele è `printDescendantsRecursive`.

```
1  public class PersonPrinter {
2
3      private final PersonRepresentator representator;
4
5      public PersonPrinter(PersonRepresentator representator) {
6          this.representator = representator;
7      }
8
9      private void print(Person person) {
10         System.out.print(representator.represent(person));
11     }
12
13     public void printDescendants(Person person) {
14         printDescendantsRecursive(person, new ArrayList<>());
15     }
16
17     private void printLines(ArrayList<Boolean> list) {
18         list.forEach(e -> System.out.print(" " + (e ? " " : "|")));
19     }
20
21     private void printDescendantsRecursive(Person person, ArrayList<Boolean> list) {
22         //printing first person
23         System.out.print("---");
24         System.out.print(person.getNumberOfMarriages() == 0 ? "-" : "/");
25         print(person);
26         for (int j = 0; j < person.getNumberOfMarriages(); j++) {
27             //printing spouse
28             Marriage marriage = person.getNthMarriage(j);
29             System.out.print("\n");
30             printLines(list);
31             System.out.print(" "
32                     + (marriage.getNumberOfChildren() == 0 ? "  " : "--") + "|\n");
33             list.add(marriage.getNumberOfChildren() == 0);
34             printLines(list);
35             System.out.print(" \\"); // "\\" prints as "\"
36             print(marriage.getOther(person));
37             for (int i = 0; i < marriage.getNumberOfChildren(); i++) {
38                 //printing child
39                 System.out.print("\n");
40                 printLines(list);
```

```
                System.out.print("\n");
                printLines(list);
                if (i == marriage.getNumberOfChildren() - 1) {
                    list.set(list.size() - 1, true);
                }
                printDescendantsRecursive(marriage.getNthChild(i), list);
            }
            list.remove(list.size() - 1);
        }
    }

}
```

## 2.4 La classe RepresentatorBuilder

Per creare tutti gli oggetti per rappresentare le persone viene usata una classe apposita chiamata `RepresentatorBuilder` con un metodo statico per mettere in vita gli oggetti che implementano la interfaccia `PersonRepresentator`. Il metodo `CreateRepresentator` riceve come input un booleano per scegliere se nella rappresentazione di una persona devono essere presenti i luoghi di nascita e di morte; inoltre riceve anche un parametro di un tipo enum per sapere se indicare o no le date di nascita e di morte e, in caso affermativo, se indicare la data completa o solo l'anno.

```
public class RepresentatorBuilder {

    public enum DateFormat {
        NONE, YEAR, FULL;
    }

    public static PersonRepresentator createRepresentator(
            DateFormat date, boolean place) {
        CompositePersonRepresentator representator =
                new CompositePersonRepresentator(
                new CompositePersonRepresentator(
                Person::getName, p -> " ", Person::getSurname));
        if (date != DateFormat.NONE || place) {
            representator.add(p -> " (");
            if (place) {
                representator.add(Person::getBirthPlace);
            }
            if (date != DateFormat.NONE && place) {
                representator.add(p -> " ");
            }
            if (date != DateFormat.NONE) {
                PersonRepresentator pr = null;
```

```
                switch (date) {
                    case YEAR:
                        pr = p -> p.getBirthDate().format(
                                DateTimeFormatter.ofPattern("yyyy"));
                        break;
                    case FULL:
                        pr = p -> p.getBirthDate().toString();
                }
                representator.add(pr);
            }

            CompositePersonRepresentator cd =
                    new CompositePersonRepresentator();
            cd.add(p -> " - ");
            if (place) {
                cd.add(p -> p.getDeathPlace().get());
            }
            if (date != DateFormat.NONE && place) {
                cd.add(p -> " ");
            }
            if (date != DateFormat.NONE) {
                PersonRepresentator pr = null;
                switch (date) {
                    case YEAR:
                        pr = p -> p.getDeathDate().get().format(
                                DateTimeFormatter.ofPattern("yyyy"));
                        break;
                    case FULL:
                        pr = p -> p.getDeathDate().get().toString();
                }
                cd.add(pr);
            }
            representator.add(new DeathRepresentator(cd));

            representator.add(p -> ")");
        }
        return representator;
    }
}
```

## 2.5 Class diagram

Segue il class diagram completo delle classi, alcune delle quali sono state anticipate nelle sezioni precedenti.

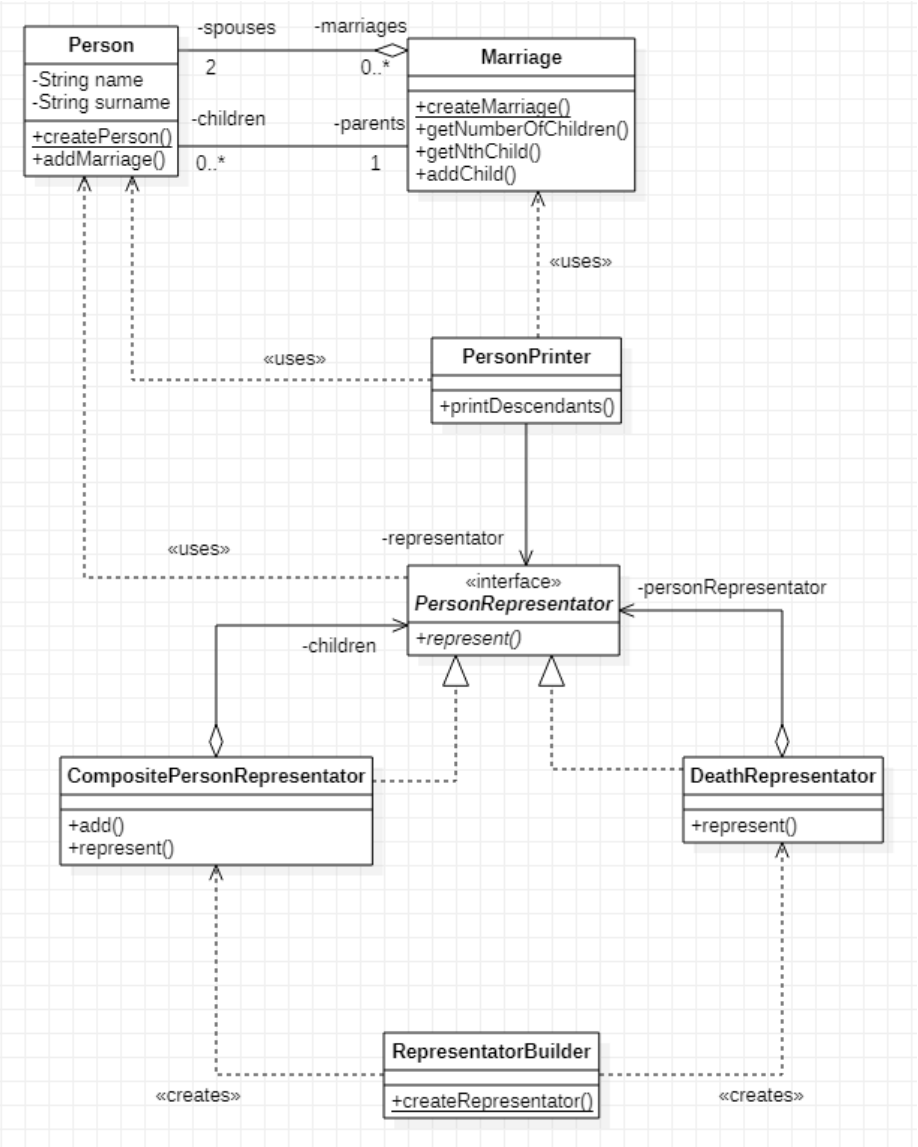


Figure 3: Il class diagram

## 2.6 Use case diagram

Nelle figure sono rappresentati gli use case diagram. Il diagramma usa tre attori diversi: uno per creare la rete di persone, uno per generare il rappresentatore tramite il builder e uno per stampare in output la rappresentazione dei discendenti delle varie persone. Ovviamente i tre attori possono essere lo stesso.

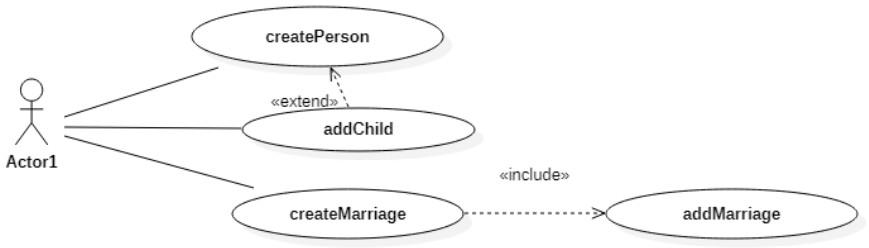


Figure 4: Use case diagram per la creazione delle persone e delle parentele

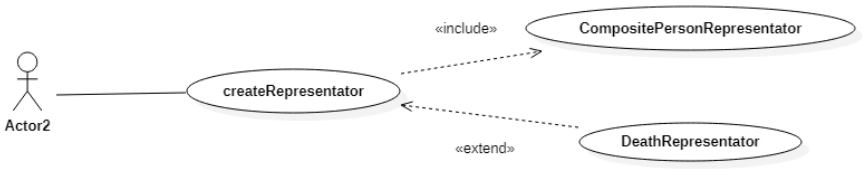


Figure 5: Use case diagram per la creazione dei rappresentatori


Figure 6: Use case diagram per la scrittura dei discendenti

# 3 Testing

Per i test vengono usate le funzioni main di alcune classi[2], in particolare le classi `Person`, `RepresentatorBuilder` e `PersonPrinter`.

Nella classe `Person`, vengono testati molti dei metodi di `Person` e `Marriage`, con particolare attenzione a i metodi che servono al collegamento fra istanze tramite riferimenti.

```
String name = "Name";
String surname = "Surname";
Person person = createPerson(name, surname);
if (!name.contentEquals(person.name)) {
        System.out.println("test failed: name mismatching");
}
if (!surname.contentEquals(person.surname)) {
        System.out.println("test failed: surname mismatching");
```

```
}

Person wife = createPerson("wife_name", "wife_surname");
Marriage marriage = Marriage.createMarriage(person, wife);
if (person.getNumberOfMarriages() != 1) {
        System.out.println("test failed: person not having 1 marriage");
}
if (wife.getNumberOfMarriages() != 1) {
        System.out.println("test failed: wife not having 1 marriage");
}
if (person.getNthMarriage(0) != marriage) {
        System.out.println("test failed: person's marriage mismatching");
}
if (wife.getNthMarriage(0) != marriage) {
        System.out.println("test failed: wife's marriage mismatching");
}
if (marriage.getOther(person) != wife) {
        System.out.println("test failed: wife not in marriage");
}
if (marriage.getOther(wife) != person) {
        System.out.println("test failed: person not in marriage");
}
if (marriage.getNumberOfChildren() != 0) {
        System.out.println("test failed: marriage not having 0 children");
}
try {
    marriage.getOther(Person.createPerson("", "");
    System.out.println("test failed: getOther did not throw exception");
} catch (RuntimeException e) {}

Person son = createPerson("son_name", "son_surname", marriage);
if (son.getParents() != marriage) {
        System.out.println("test failed: son's parents mismatching");
}
if (marriage.getNumberOfChildren() != 1) {
        System.out.println("test failed: marriage not having 1 children");
}
if (marriage.getNthChild(0) != son) {
        System.out.println("test failed: son mismatching");
}

Person son2 = createPerson("son2_name", "son2_surname");
marriage.addChild(son2);
if (son2.getParents() != marriage) {
        System.out.println("test failed: son2's parents mismatching");
}
```

```
55  if (marriage.getNumberOfChildren() != 2) {
56          System.out.println("test failed: marriage not having 2 children");
57  }
58  if (marriage.getNthChild(1) != son2) {
59          System.out.println("test failed: son2 mismatching");
60  }
```

L'object diagram relativo a questi test è riportato in figura 7.

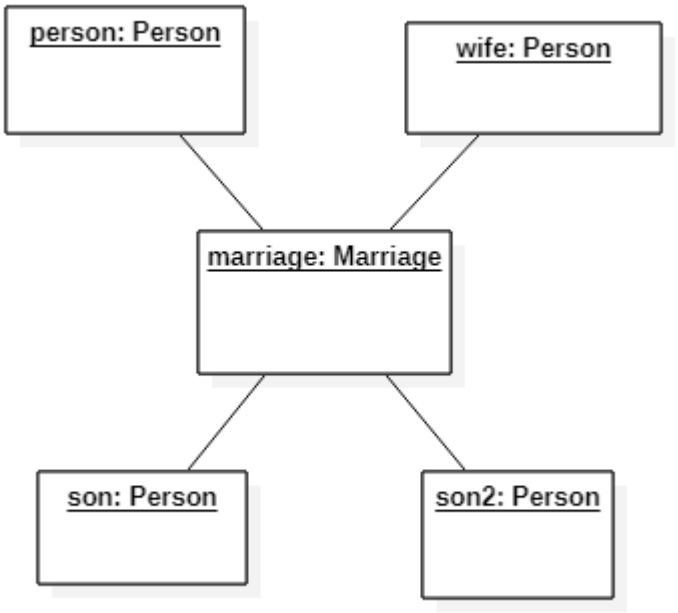


Figure 7: Object diagram relativo ai test su Person

I test su `RepresentatorBuilder` controllano tutti i 6 casi possibili per la rappresentazione di persone, sia con persone vive che con persone morte. Segue il relativo codice.

```
1  Person alive = Person.createPerson("name", "surname");
2  alive.setBirthDate(LocalDate.of(2000, 1, 1));
3  alive.setBirthPlace("birth_place");
4  Person dead = Person.createPerson("name", "surname");
5  dead.setBirthDate(LocalDate.of(2000, 1, 1));
6  dead.setBirthPlace("birth_place");
7  dead.setDeathDate(LocalDate.of(2001, 1, 1));
8  dead.setDeathPlace("death_place");
9
```

```
10  PersonRepresentator pr00 = createRepresentator(DateFormat.NONE, false);
11  if (!pr00.represent(alive).contentEquals("name surname")) {
12          System.out.println("test failed: alive NONE false");
13          System.out.println("  got: " + pr00.represent(alive));
14  }
15  if (!pr00.represent(dead).contentEquals("name surname")) {
16          System.out.println("test failed: dead NONE false");
17          System.out.println("  got: " + pr00.represent(dead));
18  }
19
20  PersonRepresentator pr01 = createRepresentator(DateFormat.NONE, true);
21  if (!pr01.represent(alive).contentEquals(
22          "name surname (birth_place)")) {
23          System.out.println("test failed: alive NONE true");
24          System.out.println("  got: " + pr01.represent(alive));
25  }
26  if (!pr01.represent(dead).contentEquals(
27          "name surname (birth_place - death_place)")) {
28          System.out.println("test failed: dead NONE true");
29          System.out.println("  got: " + pr01.represent(dead));
30  }
31
32  PersonRepresentator pr10 = createRepresentator(DateFormat.YEAR, false);
33  if (!pr10.represent(alive).contentEquals(
34          "name surname (2000)")) {
35          System.out.println("test failed: alive YEAR false");
36          System.out.println("  got: " + pr10.represent(alive));
37  }
38  if (!pr10.represent(dead).contentEquals(
39          "name surname (2000 - 2001)")) {
40          System.out.println("test failed: dead YEAR false");
41          System.out.println("  got: " + pr10.represent(dead));
42  }
43
44  PersonRepresentator pr11 = createRepresentator(DateFormat.YEAR, true);
45  if (!pr11.represent(alive).contentEquals(
46          "name surname (birth_place 2000)")) {
47          System.out.println("test failed: alive YEAR true");
48          System.out.println("  got: " + pr11.represent(alive));
49  }
50  if (!pr11.represent(dead).contentEquals(
51          "name surname (birth_place 2000 - death_place 2001)")) {
52          System.out.println("test failed: dead YEAR true");
53          System.out.println("  got: " + pr11.represent(dead));
54  }
55
```

```
56 PersonRepresentator pr20 = createRepresentator(DateFormat.FULL, false);
57 if (!pr20.represent(alive).contentEquals(
58         "name surname (2000-01-01)")) {
59         System.out.println("test failed: alive FULL false");
60         System.out.println("  got: " + pr20.represent(alive));
61 }
62 if (!pr20.represent(dead).contentEquals(
63         "name surname (2000-01-01 - 2001-01-01)")) {
64         System.out.println("test failed: dead FULL false");
65         System.out.println("  got: " + pr20.represent(dead));
66 }
67
68 PersonRepresentator pr21 = createRepresentator(DateFormat.FULL, true);
69 if (!pr21.represent(alive).contentEquals(
70         "name surname (birth_place 2000-01-01)")) {
71         System.out.println("test failed: alive FULL true");
72         System.out.println("  got: " + pr21.represent(alive));
73 }
74 if (!pr21.represent(dead).contentEquals(
75         "name surname (birth_place 2000-01-01 - death_place 2001-01-01)")) {
76         System.out.println("test failed: dead FULL true");
77         System.out.println("  got: " + pr21.represent(dead));
78 }
```

Infine, per verificare la correttezza della classe `PersonPrinter`, si usa un test strutturato secondo il diagramma di figura 8. Il risultato del test deve essere quello riportato nel seguito (viene usato un ramo della famiglia reale britannica come esempio).

```
---/Charles Mountbatten-Windsor
 --|
 | \Diana Spencer
 |
 |----William Mountbatten-Windsor
 |
 |----Harry Mountbatten-Windsor
   |
   \Camilla Parker Bowles
```

# References


[1] Java Practices. Return optional not null, . URL `http://www.javapractices.com/topic/TopicAction.do?Id=279`.

[2] Java Practices. Test using main method, . URL `http://www.javapractices.com/topic/TopicAction.do?Id=174`.

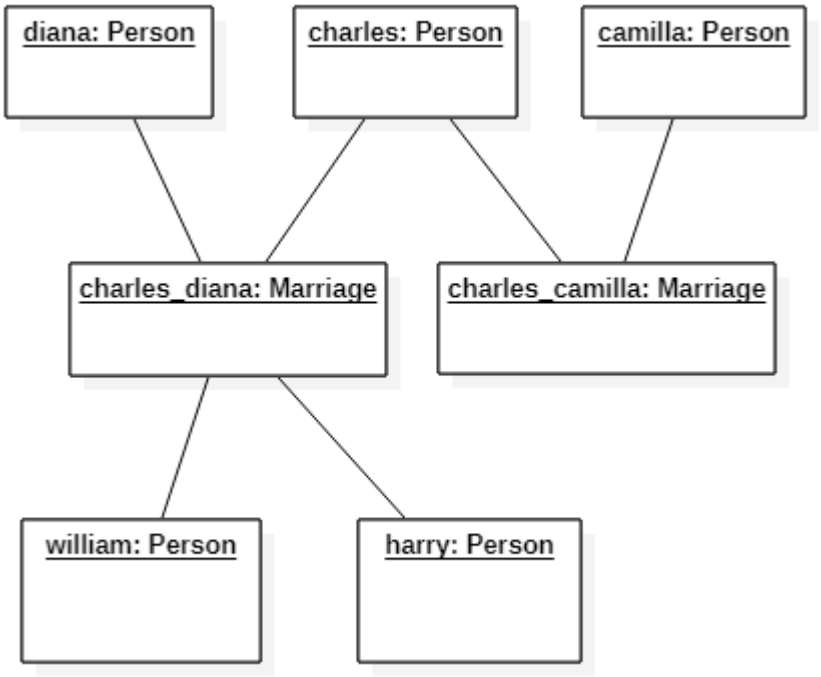


Figure 8: Object diagram relativo ai test su PersonPrinter