

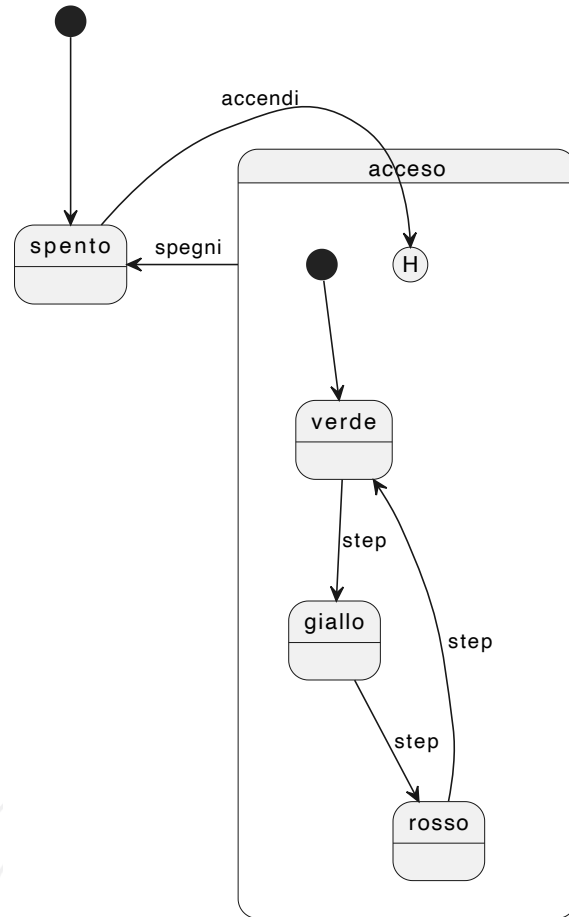
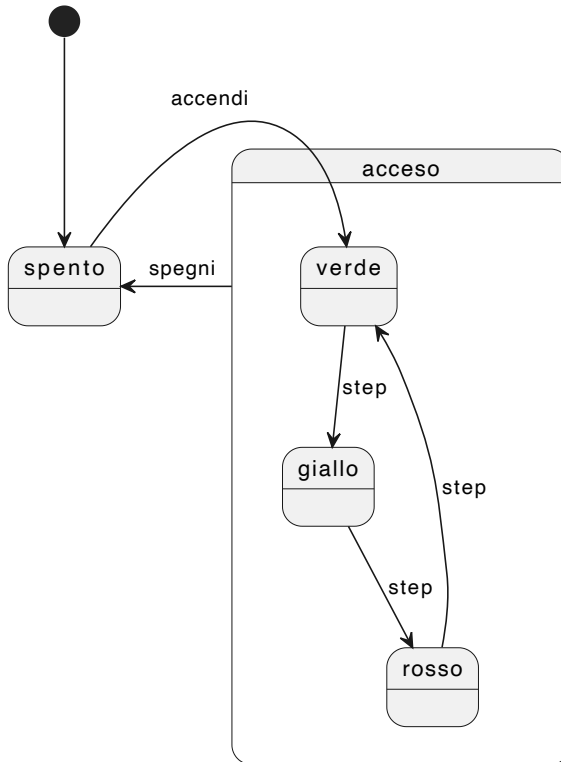
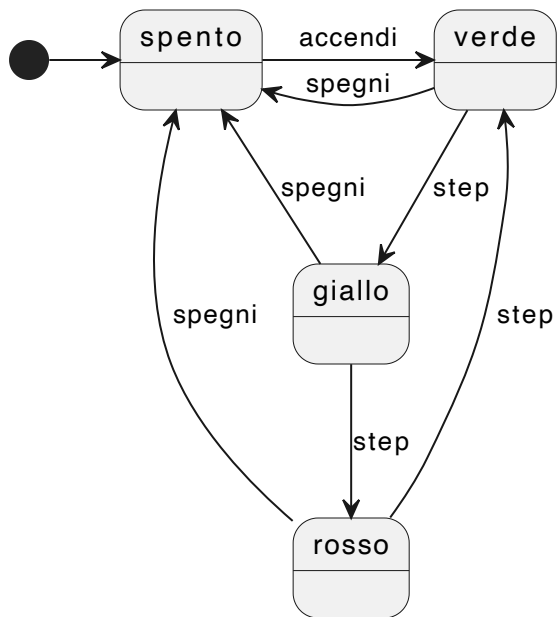


UNIVERSITÀ DEGLI STUDI
DI MILANO

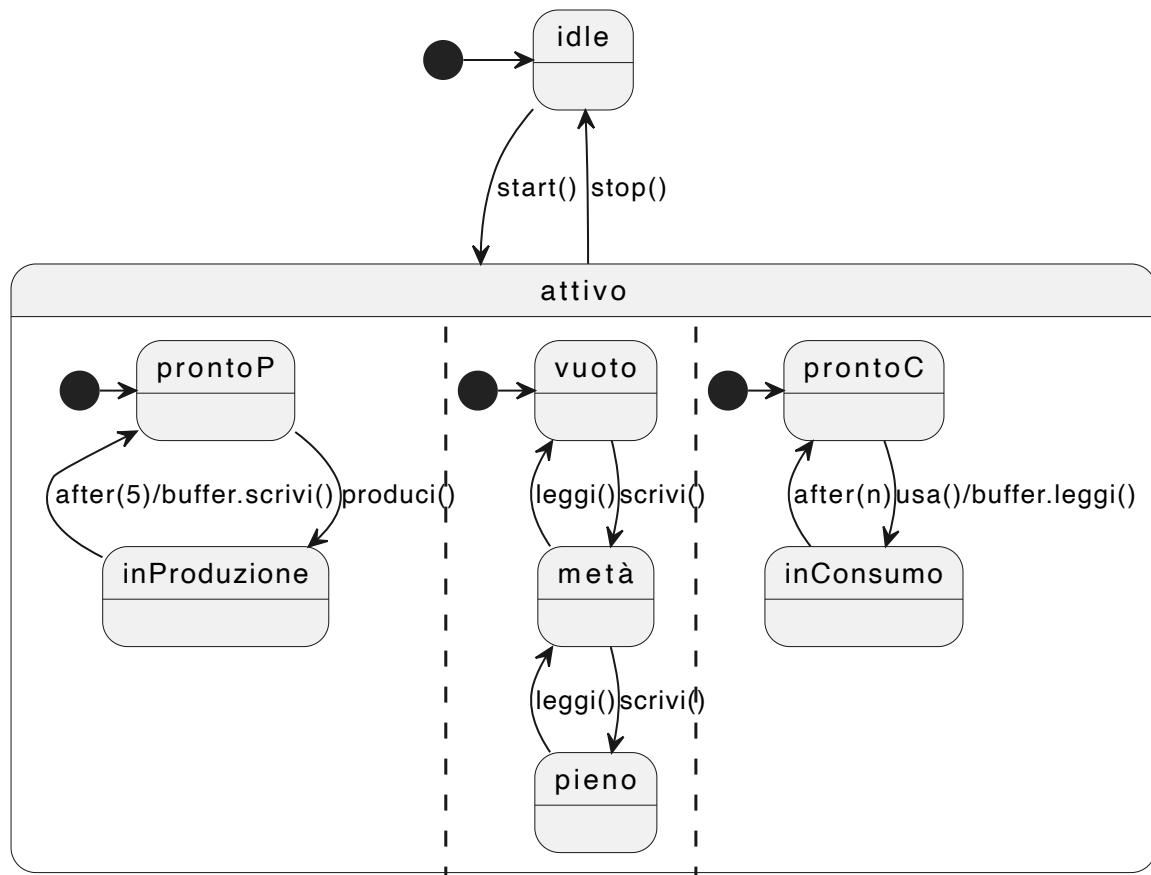
Ingegneria del software

Progettazione

Superstate



Concorrenza



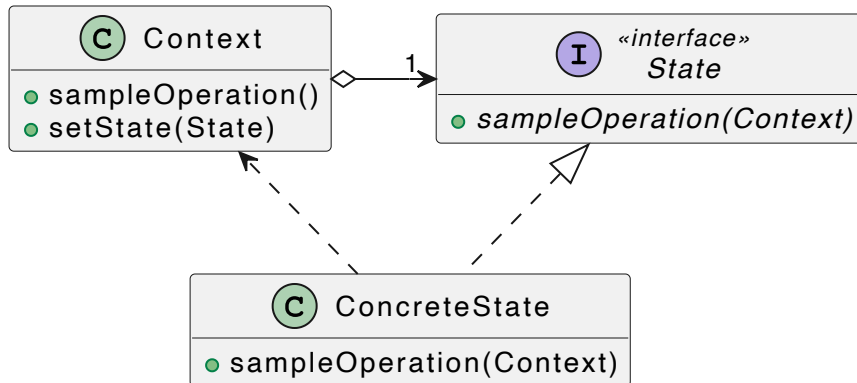
Torniamo a parlare di qualche pattern

- Segnalo oltre al libro Head First
 - il sito: <https://refactoring.guru>

Interessante sia per design patterns che per Refactoring actions

STATE pattern

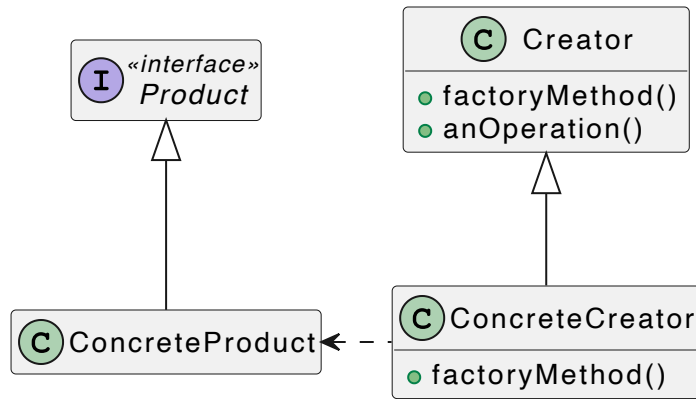
Permette di modellare cambiamenti di comportamento al cambiare dello stato dell'oggetto



```
public class Context {
    private State state;
    public void setState(@NotNull State s) {
        state = s;
    }
    public void sampleOperation() {
        state.sampleOperation(this);
    }
}
```

- Viene partizionato il comportamento nelle varie classi stato
- Viene reso esplicito (e atomico) il passaggio di stato
- Gli oggetti stato possono essere in alcuni casi condivisi

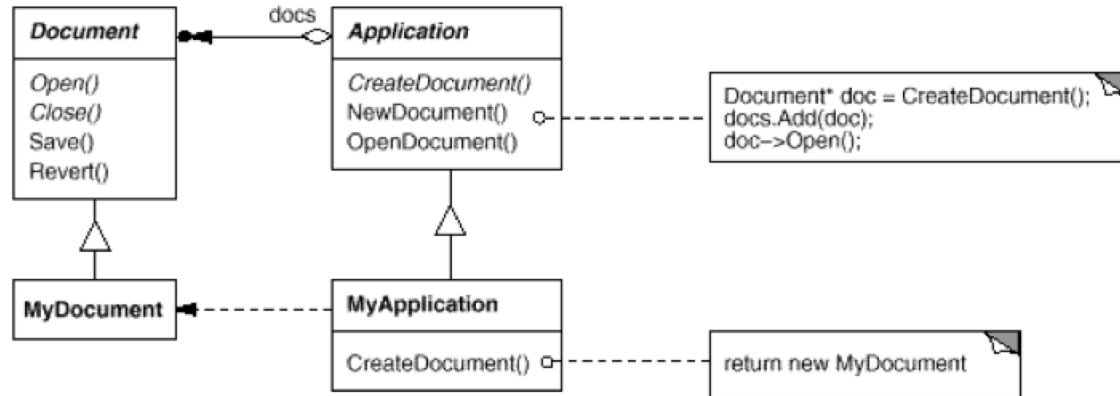
FACTORY METHOD pattern



Factory Method

Type: Creational

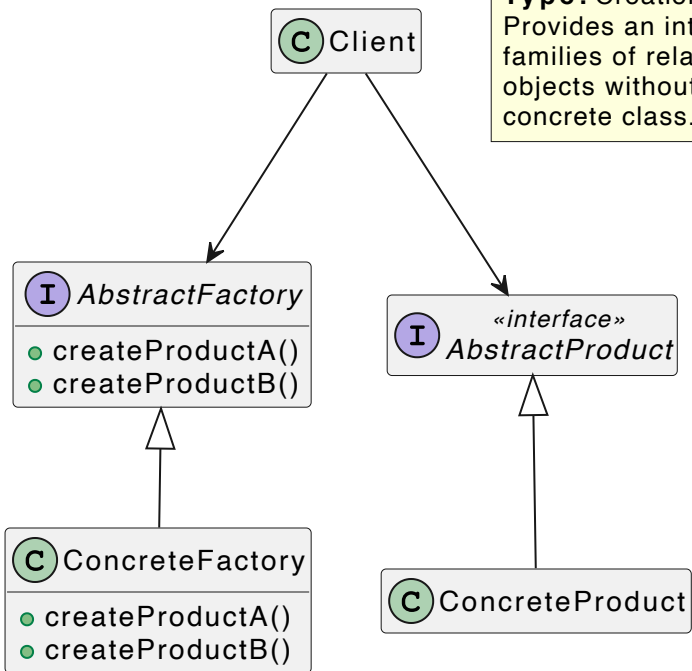
Definisce una interfaccia per creare un oggetto, ma lascia alle sottoclassi la scelta di cosa creare.



ABSTRACT FACTORY pattern

Abstract Factory

Type: Creational
Provides an interface for creating families of related or dependent objects without specifying their concrete class.



osservazioni su NULL OBJECT pattern

- abbiamo parlato di **oggetto...** non di **classe**

Cosa abbiamo scritto?

```
public class RandomStrategy implements Strategia {
    final private Strategia next;

    public RandomStrategy(Strategia next) {
        assert next != null : "se si usa NullObject Pattern non si ha mai strategia 'null'";
        this.next = next;
    }

    @Override
    public boolean chiediCarta() {
        if (Math.random() > 0.5)
            return true;
        return next.chiediCarta();
    }
}
```


Se non ci fosse Strategia *nulla*, avremmo dovuto scrivere:

```
@Override
public boolean chiediCarta() {
    if (Math.random() > 0.5)
        return true;
    if (next != null)
        return next.chiediCarta();
    else
        return false;
}
```



UNIVERSITÀ DEGLI STUDI
DI MILANO

Musica maestro Kata

- Implementare due diverse classi che rappresentano due diversi strumenti:

Trumpet e Horn. Entrambe le classi

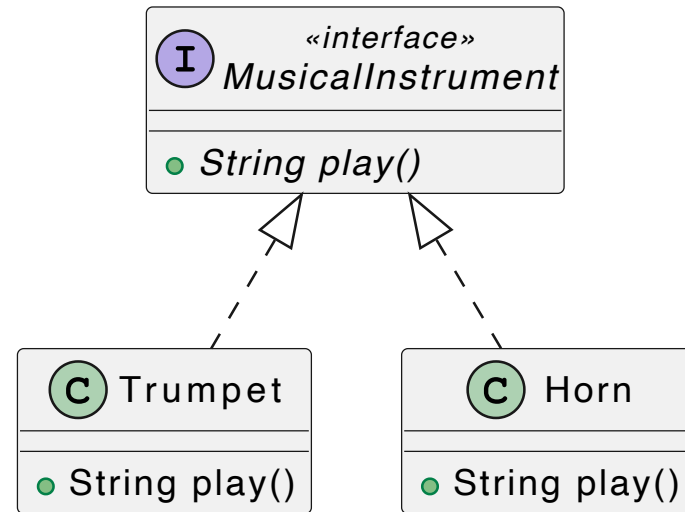
implementano l'interfaccia

MusicalInstrument e rispondono alla

chiamata del metodo

```
public String play();
```

- che restituisce nel primo caso la stringa **pepepe** e nel secondo la stringa **papapa**.



Implementare altre due classi `WaterGlass` e `IronRod`.

Esse, pur NON implementando l'interfaccia `MusicalInstrument` hanno un comportamento molto simile:

- `WaterGlass` possiede un metodo

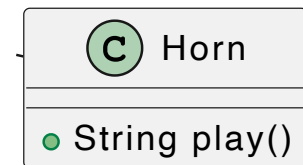
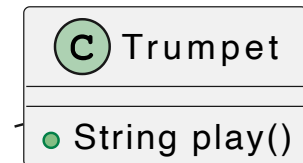
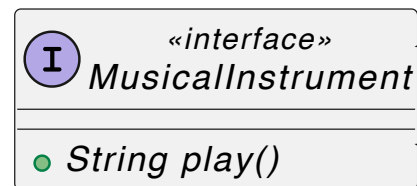
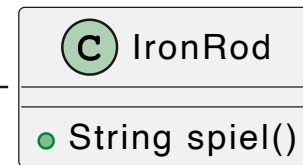
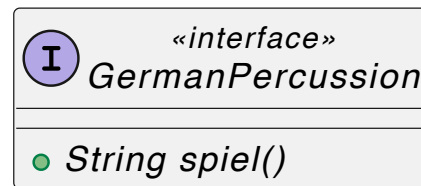
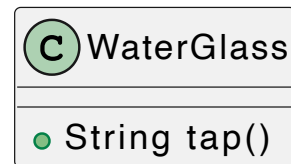
```
public String tap()
```

che restituisce `diding`

- `IronRod` invece aderisce alla interfaccia `GermanPercussion` e risponde quindi alla chiamata

```
public String spiel()
```

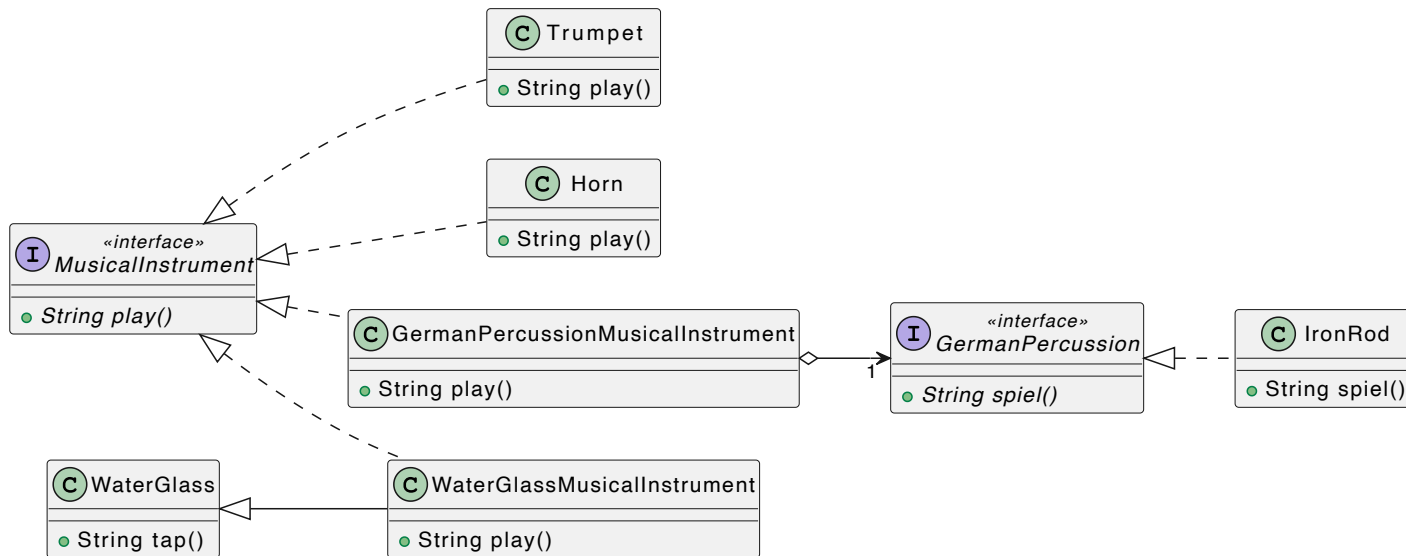
restituendo `tatang`



Aggiungere la possibilità di usare istanze della classe `WaterGlass` e delle classi aderenti alla interfaccia `GermanPercussion` (e in particolare perciò della classe `IronRod`) come oggetti di tipo `MusicalInstrument`

Tale obiettivo deve essere raggiunto usando il design pattern denominato adapter:

- `WaterGlassMusicalInstrument` che realizza un **CLASS ADAPTER**
- `GermanPercussionMusicalInstrument` che realizza invece un **OBJECT ADAPTER**.

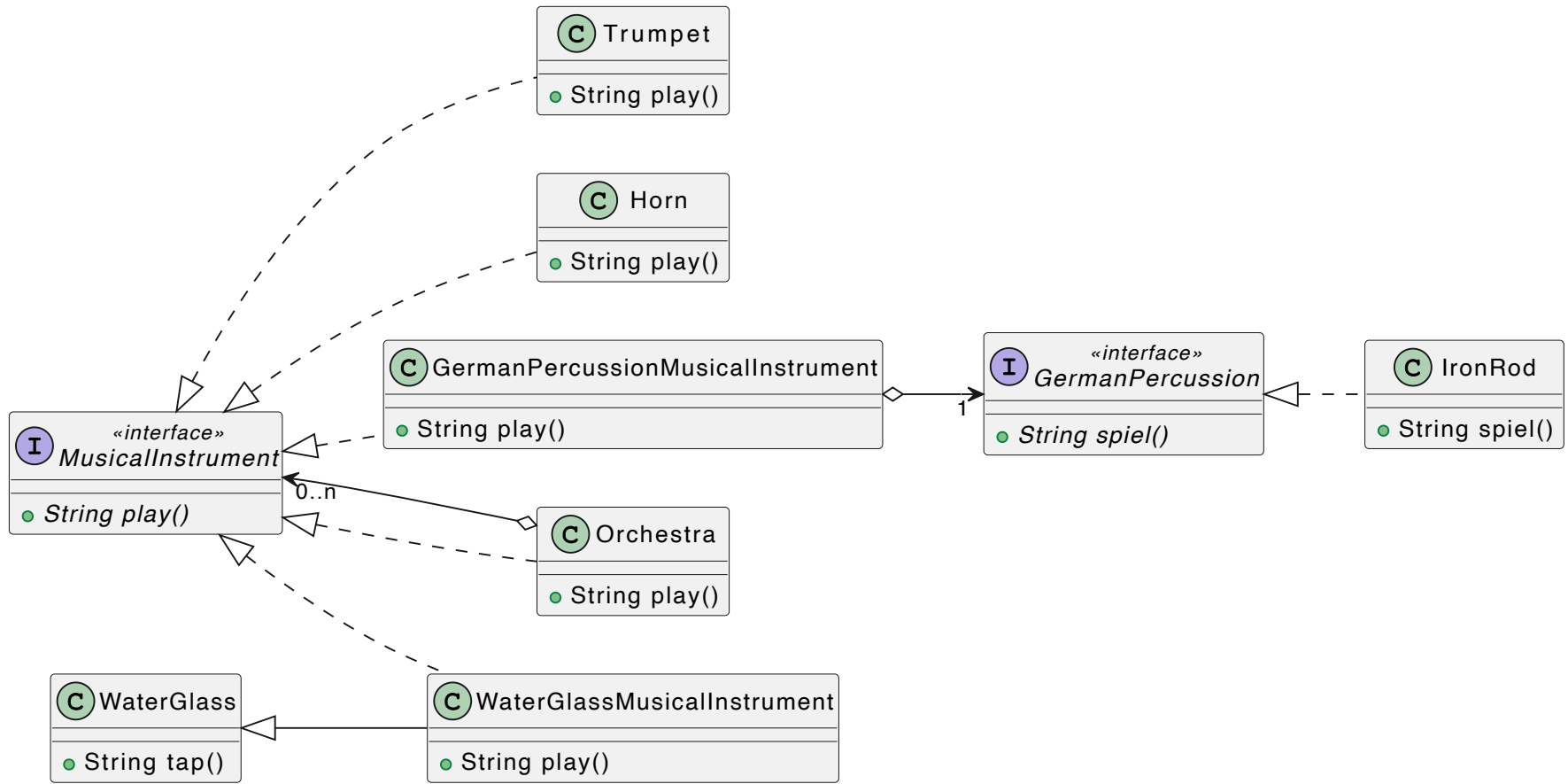


Creare una classe che rappresenti un'orchestra di `MusicalInstrument`.

Tale obiettivo deve essere raggiunto usando il design pattern `COMPOSITE`

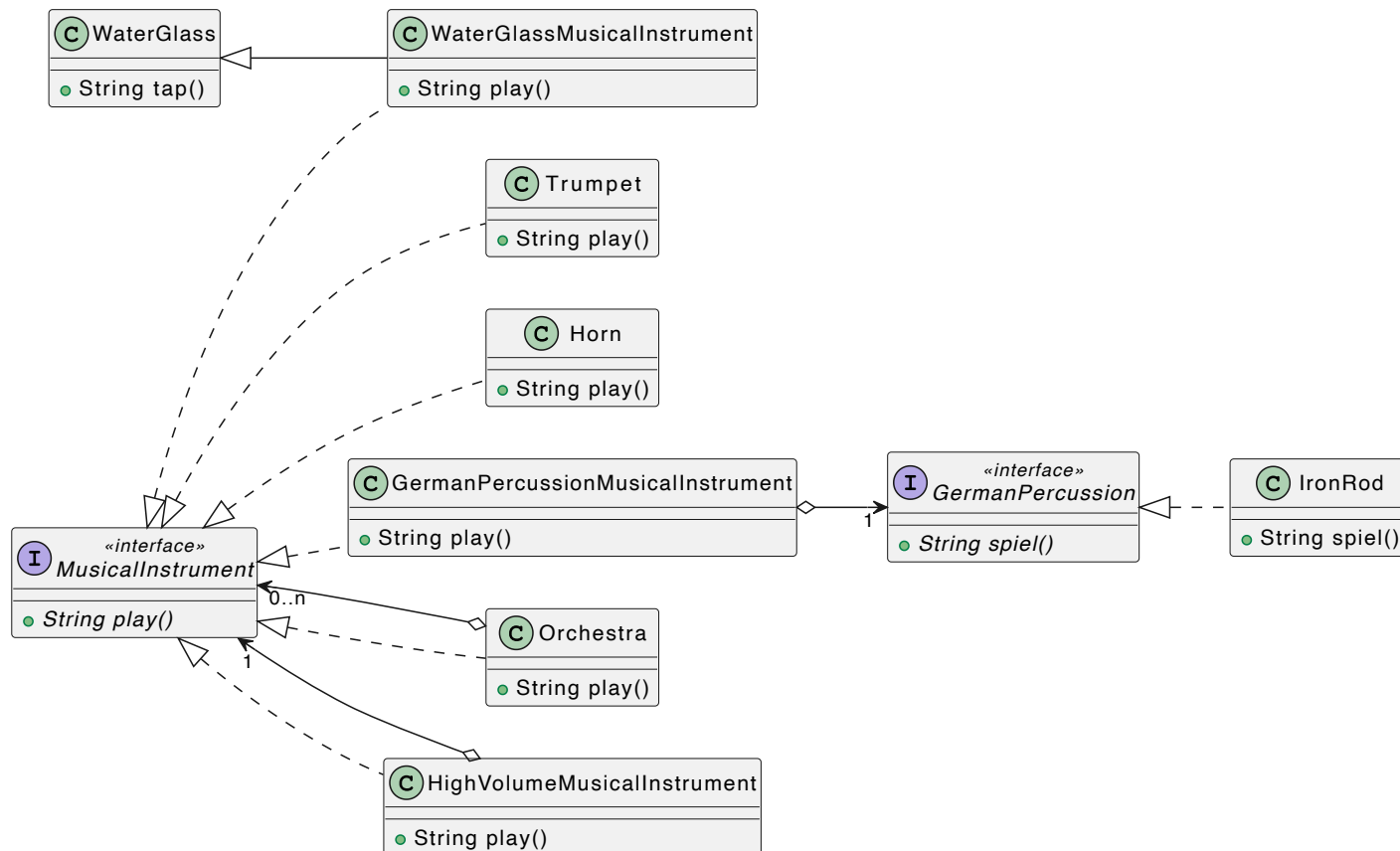
- È richiesta la realizzazione di una classe `Orchestra` che deve implementare l'interfaccia `MusicalInstrument` e deve rispondere all'invocazione del metodo `String play()` demandando la chiamata agli oggetti `MusicalInstrument` aggregati.
- Le istanze della classe `Orchestra` devono consentire l'aggiunta di elementi tramite l'invocazione del metodo

```
public void add(MusicalInstrument instrument)
```



Aggiungere la possibilità di far suonare più forte (stringa in maiuscolo) un oggetto di tipo `MusicalInstrument`.

Tale obiettivo deve essere raggiunto usando il design pattern `Decorator`; più in dettaglio, è richiesta la realizzazione di una classe `HighVolumeMusicalInstrument` che implementa l'interfaccia `MusicalInstrument` e decora un'istanza di `MusicalInstrument` in modo che tutti gli strumenti decorati quando richiesto suonino "forte".



- Aggiungere la possibilità di far suonare *più lentamente* (raddoppiando le vocali) un oggetto di tipo `MusicalInstrument`.
- Anche questo obiettivo deve essere raggiunto usando il design pattern `Decorator` creando una classe `SlowTempoMusicalInstrument`
- Dopo averlo realizzato provare a fattorizzare le parti comuni ai due decorator tramite pattern `Template` creando una classe astratta `DecoratorMusicalInstrument`

