Ingegneria del Software

Esercitazione 6

Trasformatore

- Date le classi Java Volt e PresaCorrente, che restituisce corrente a 220 volt:
- Si utilizzi il design pattern **?** per “realizzare” un trasformatore capace di erogare corrente a 3, 12, e 220 volt:

Si definisca la classe Java da aggiungere, specificando anche le relazioni con le classi esistenti

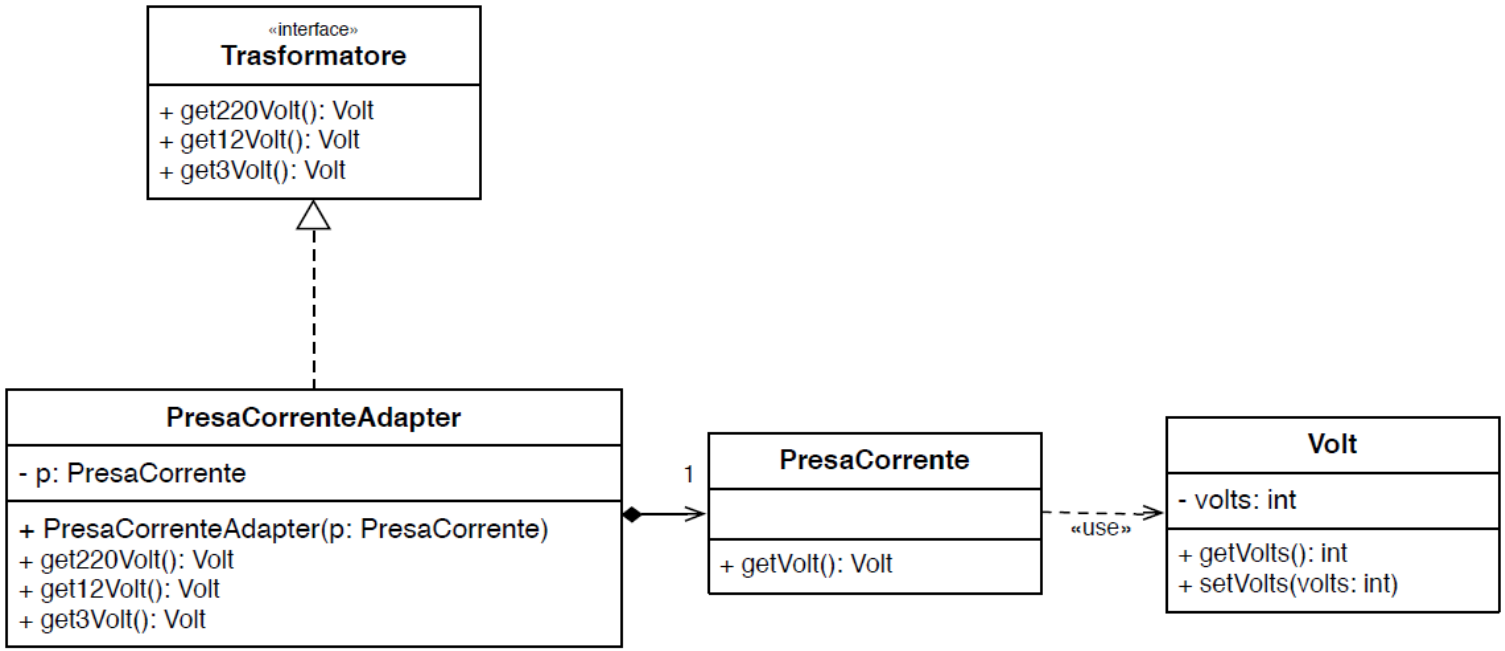
```
public interface Trasformatore {
        public Volt get220Volt();
        public Volt get12Volt();
        public Volt get3Volt();
}
```

Trasformatore

- Date le classi Java Volt e PresaCorrente, che restituisce corrente a 220 volt:
- Si utilizzi il design pattern adapter per “realizzare” un trasformatore capace di erogare corrente a 3, 12, e 220 volt:

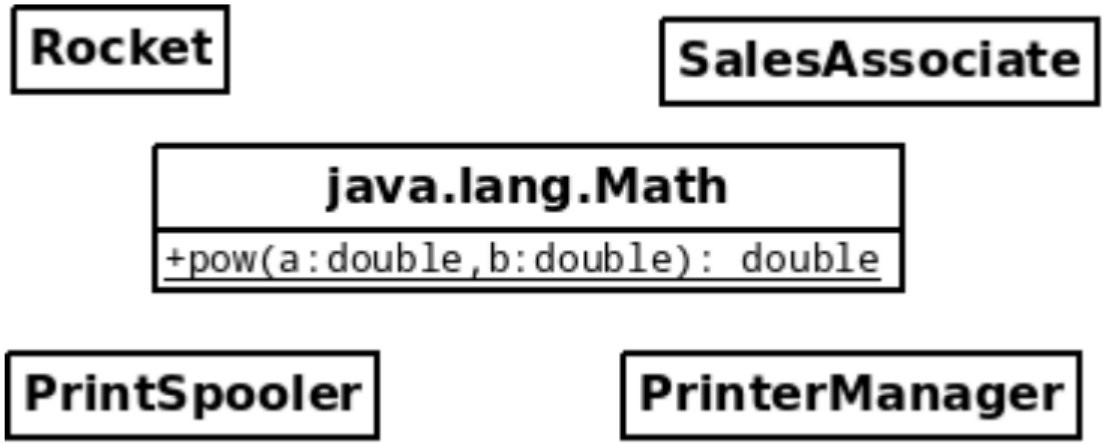
Si definisca la classe Java da aggiungere, specificando anche le relazioni con le classi esistenti

```
public interface Trasformatore {
        public Volt get220Volt();
        public Volt get12Volt();
        public Volt get3Volt();
}
```


«interface»
Trasformatore
+ get220Volt(): Volt
+ get12Volt(): Volt
+ get3Volt(): Volt
PresaCorrenteAdapter
- p: PresaCorrente
+ PresaCorrenteAdapter(p: PresaCorrente)
+ get220Volt(): Volt
+ get12Volt(): Volt
+ get3Volt(): Volt
1
PresaCorrente
+ getVolt(): Volt
«use»
Volt
- volts: int
+ getVolts(): int
+ setVolts(volts: int)

Singleton

Dato il seguente schema di classi, dire quali classi potrebbero applicare il pattern Singleton



Singleton

Rocket: non può essere un Singleton, è più probabile che sia una normale e comune classe.

SalesAssociate: come per Rocket.

java.lang.Math: ha metodi statici, dunque non può essere un Singleton.

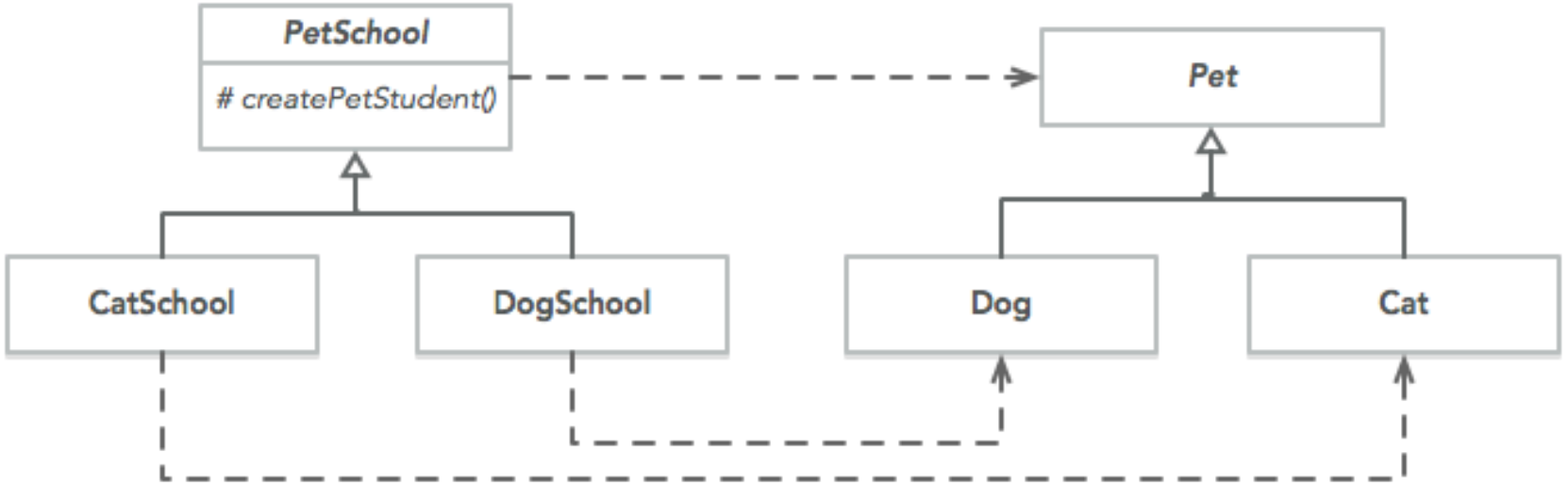
PrintSpooler: lo spooler di stampante è in ogni stampante, dunque non può essere un Singleton.

PrinterManager: un gestore di tutte le stampanti è decisamente un singleton. Una sola classe per gestire tutte le stampanti di un ufficio, per esempio.

PetSchool

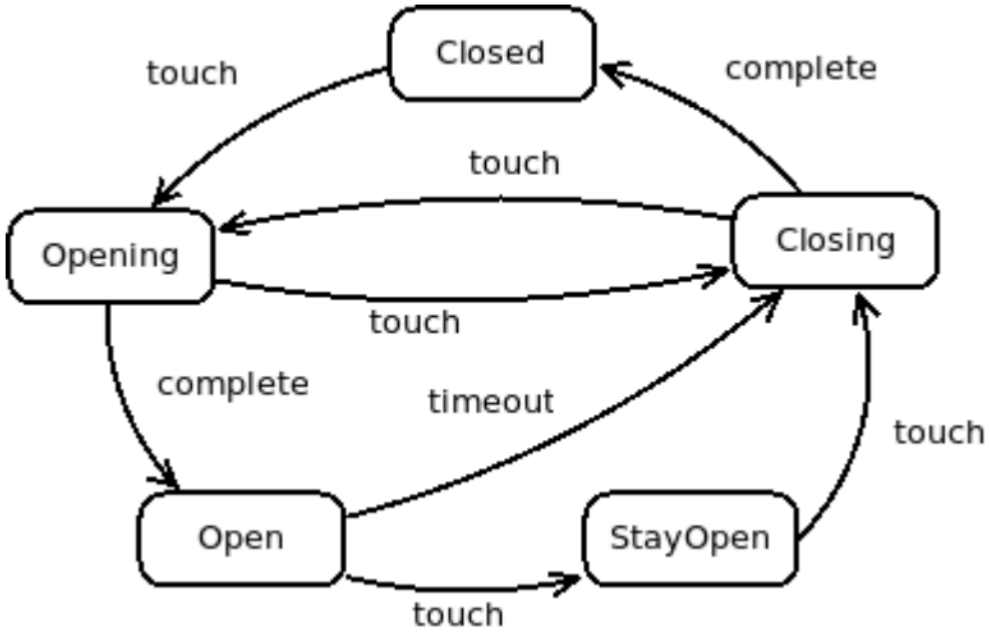
Rappresentare in UML ed implementare in Java un insieme di classi che permettano la creazione di una PetSchool. Una PetSchool è una scuola composta da animali invece che da essere umani. Ogni PetSchool è composta da animali dello stesso tipo (solo gatti, solo cani, etc.) in numero pari a 50. Il costruttore di PetSchool deve riempire la scuola. Usare un adeguato design pattern.

PetSchool

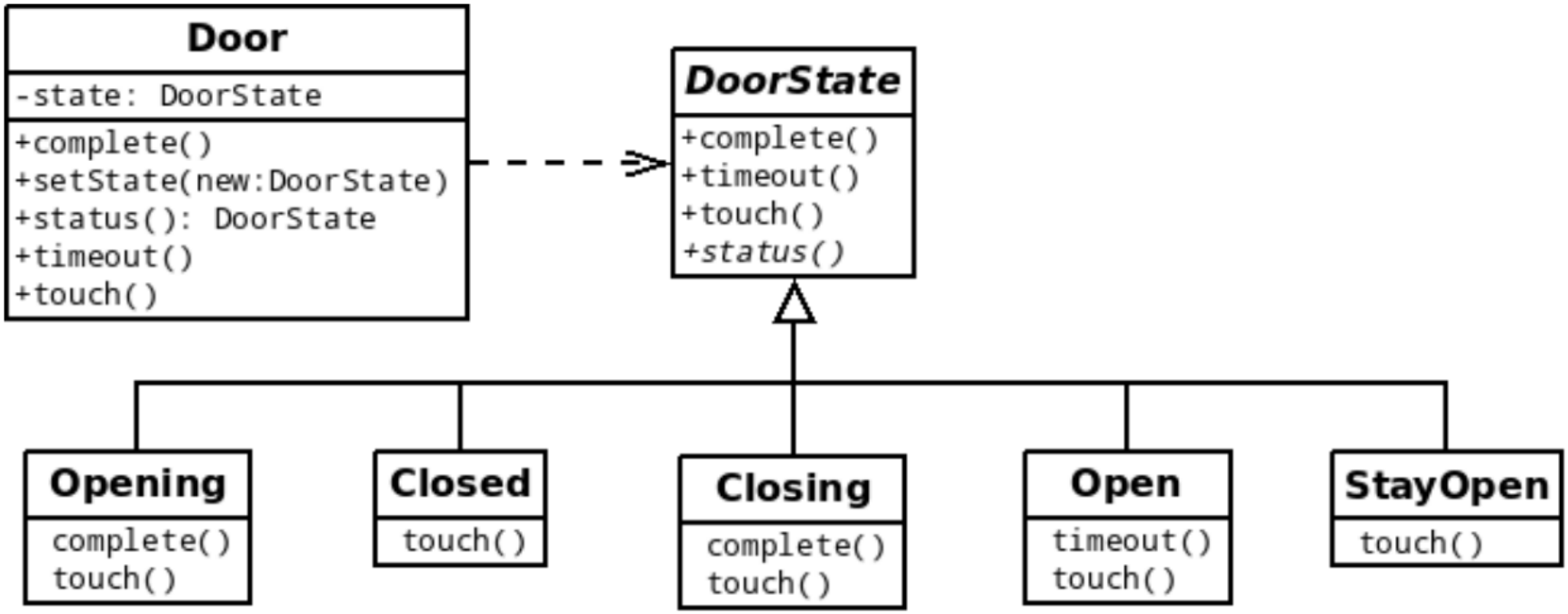

PetSchool
createPetStudent()
Pet
CatSchool
DogSchool
Dog
Cat

Porta

Dato il seguente grafo degli stati di una Porta, scrivere lo schema delle classi per gli stati



Porta

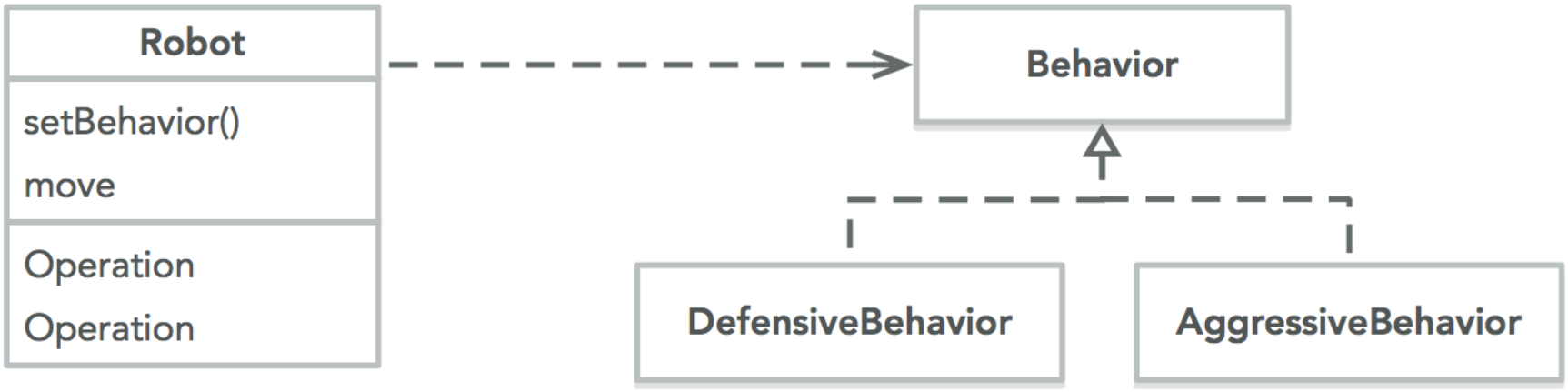

Door
-state: DoorState
+complete()
+setState(new:DoorState)
+status(): DoorState
+timeout()
+touch()
DoorState
+complete()
+timeout()
+touch()
+status()
Opening
complete()
touch()
Closed
touch()
Closing
complete()
touch()
Open
timeout()
touch()
StayOpen
touch()

Robot

Creare un diagramma di classi in UML che rappresenti un Robot. Un robot può muoversi in diversi modi, ad esempio in maniera aggressiva o difensiva. La modalità può cambiare a runtime. Usare un design pattern adeguato.

Robot

Strategy Pattern

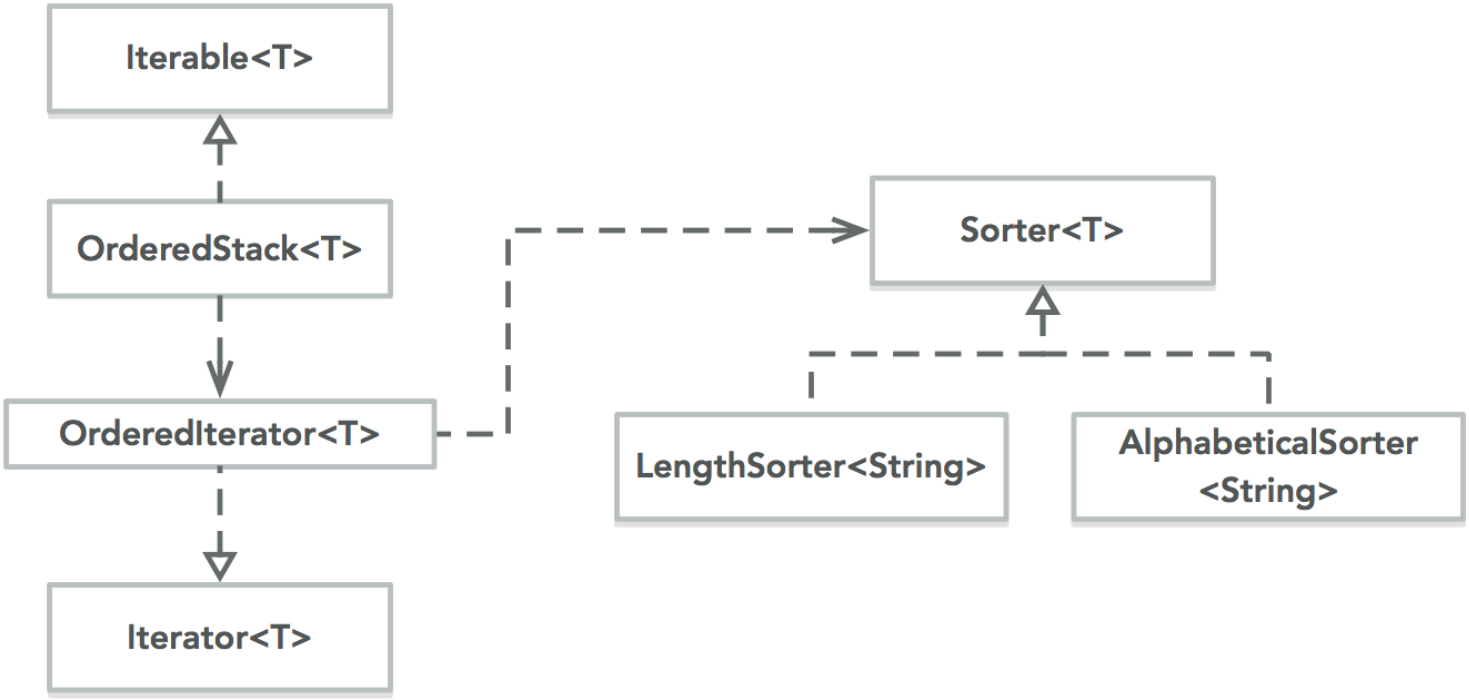


OrderedStack

Definire in UML uno Stack generico che ritorni un iteratore che legga lo stack rispetto ad un criterio personalizzabile dall'utilizzatore. Ad esempio uno Stack di stringhe può essere letto in ordine di lunghezza delle stringhe o in ordine alfabetico.
Usare un pattern adeguato.

OrderedStack

Strategy Pattern



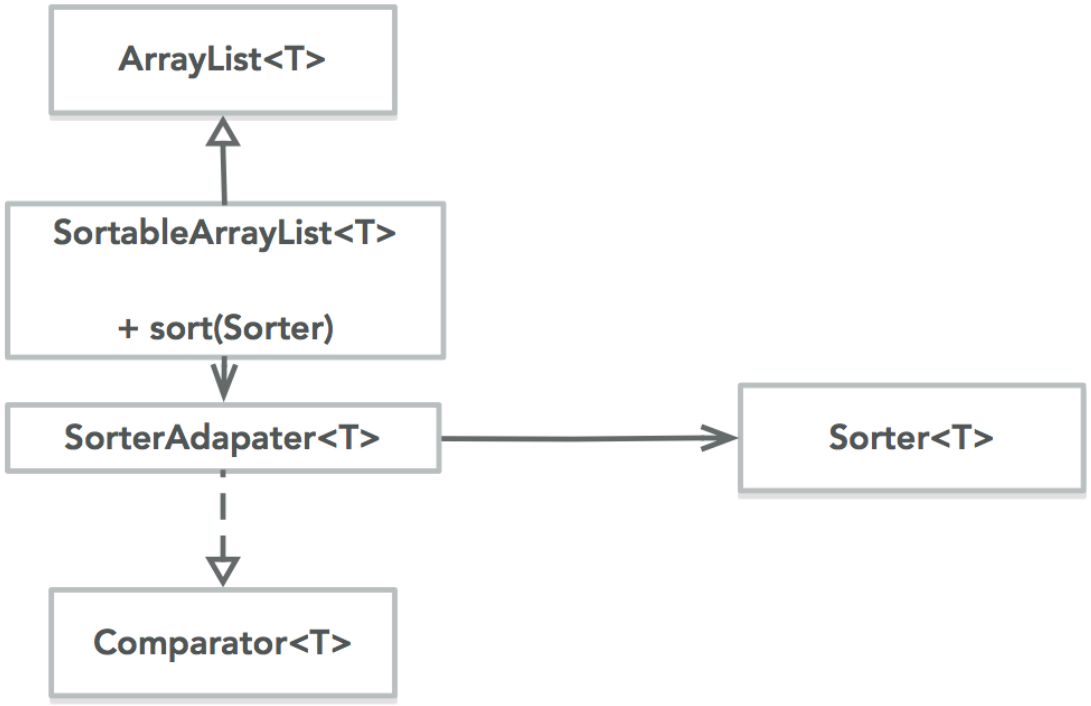
Sorter

E se volessimo usare un ArrayList con l'interfaccia Sorter per ordinarlo? ArrayList usa Comparator.

Che pattern possiamo utilizzare?

Sorter

Strategy Pattern



Weather Station

Implementare un sistema con il quale è possibile monitorare dati meteorologici. La classe WeatherData può essere aggiornata da un client con i dati aggiornati riguardo temperatura, pressione e umidità. La classe WeatherDisplay deve mostrare i dati non appena questi vengono aggiornati. Usare un pattern adeguato.

JML

Esercizio 1

```
public static boolean aggiungiStudente (Studente
s, Aula a);

class Aula {
    public boolean isIn (Studente s) {...}
}
```

Esercizio 1

```
/*@
@ requires s != null && a != null
@ ensures a.isIn(s) &&
@ (\forall Student t; !s.equals(t); a.isIn(t) <==> \old(a.isIn(t)))
@ \result <==> !\old(a.isIn(s))
@*/

public static boolean aggiungiStudente (Studente
s, Aula a);

class Aula {
    public boolean isIn (Studente s) {...}
}
```

Esercizio 2

```
class Indovina {
    static int count = 0;

    static int mistero(int x, String y,
int[] z) {
        System.out.println(count);
        System.out.println(y + z[x]);
        x++;
        z[x] = z[x-1] + 1;
        count++;
        return x;
    }
}
```

Esercizio 2

```
/*@
@ requires z != null && y != null &&
@             0 <= \old(x) <= \old(z.length-2)
@ ensures \result == \old(x)+1 &&
@ count==\old(count)+1 &&
@ z[\result] == z[\result - 1] + 1;
@ assignable z[\result]
@*/
int mistero(int x, String y, int[] z);
```

Esercizio 3

Sia data la classe Interval

```
public class Interval{
    private float low, high;
    public float getLowerBound(){…}
    public float getUpperBound(){…}
    public static Interval getInterval(float[] times, float
timePoint)
}
```

Esercizio 3

- Definire in JML il contratto che rispetti le seguenti specifiche:
  - *times* non nullo
  - *times* con valori in ordine strettamente crescente
  - Restituisce un oggetto di tipo *Interval* che corrisponde a un intervallo temporale avente come estremi due punti contigui di times.
  - *timePoint* deve essere maggiore o uguale all'estremo minore e strettamente minore dell'estremo maggiore

Esercizio 3

```
/*@
@ assignable \nothing
@ requires times != null && times.length >= 2 &&
@ timePoint >= times[0] && timePoint < times[times.length -1]
@ && (\forall int i; 0<=i &&
i<times.length-1;times[i]<times[i+1])
@ ensures (\exists int i; 0<=i<times.length-1;
@  times[i] == \result.getLowerBound() &&
@  times[i+1] == \result.getUpperBound())) &&
@  \result.getLowerBound() <= timePoint &&
@  \result.getUpperBound() > timePoint );
@*/
Interval getInterval(float[] times, float timePoint);
```