Ingegneria del Software

Esercitazione 7

Networking

SocSeq

Si realizzi in Java, utilizzando le librerie Thread e Socket un server di rete che gestisce un gioco elettronico distribuito, a cui possono giocare più utenti contemporaneamente.

Il gioco consiste nel leggere rapidamente una sequenza di numeri e nel riscriverla correttamente.

Una volta lanciato il client chiede all'utente di inserire username e password e le invia al server dopo aver inviato il messaggio "login". Se l'utente inserisce un username mai usato da nessuno nel sistema, il server lo registra associandolo con la password inserita. Se invece l'utente inserisce uno username già registrato il server verifica che la password ricevuta sia corretta. La procedura viene ripetuta fino a che il processo di login termina correttamente. Se il processo va a buon fine il server invia la stringa "true", altrimenti invia la stringa "false".

SocSeq (II)

Una volta loggato l'utente può decidere di iniziare una partita o di richiedere la lista dei record personali di tutti gli utenti.

Nel primo caso il client invia la stringa "partita". Il server invia al client la prima sequenza da ricordare. Il client la mostra per qualche istante all'utente, poi chiede all'utente di ridigitarla, e la invia al server. Il server controlla che essa sia uguale a quella inviata, se essa non è corretta invia la stringa "false" e termina la partita, altrimenti risponde con la stringa "true" e invia un'altra sequenza di numeri.

Se invece l'utente sceglie di vedere i record dei giocatori, il client invia al server la stringa "record" e questi risponde con una stringa rappresentante il numero dei giocatori seguita da una stringa per ognuno dei giocatori contenente i suoi record.

Quando il giocatore sceglie di uscire dal programma, il client invia al server la stringa "fine", e il server chiude la connessione col client.

Design Pattern

Impiegati (TDE)

Si consideri l'insieme di impiegati di un'azienda. Gli impiegati espongono tre metodi String getName() e String getOffice() che ritornano nome e ufficio degli impiegati e String getDescrizione() che ritorna le mansioni dell'impiegato.
Ci sono vari tipi di impiegato, per esempio gli ingegneri. Le responsabilità degli impiegati (si considerino per semplicità solo gli ingegneri) possono cambiare dinamicamente. In particolare, un ingegnere può avere la responsabilità di manager amministrativo o manager di progetto. Il comportamento del metodo String getDescrizione() viene modificato opportunamente.

Impiegati (TDE)

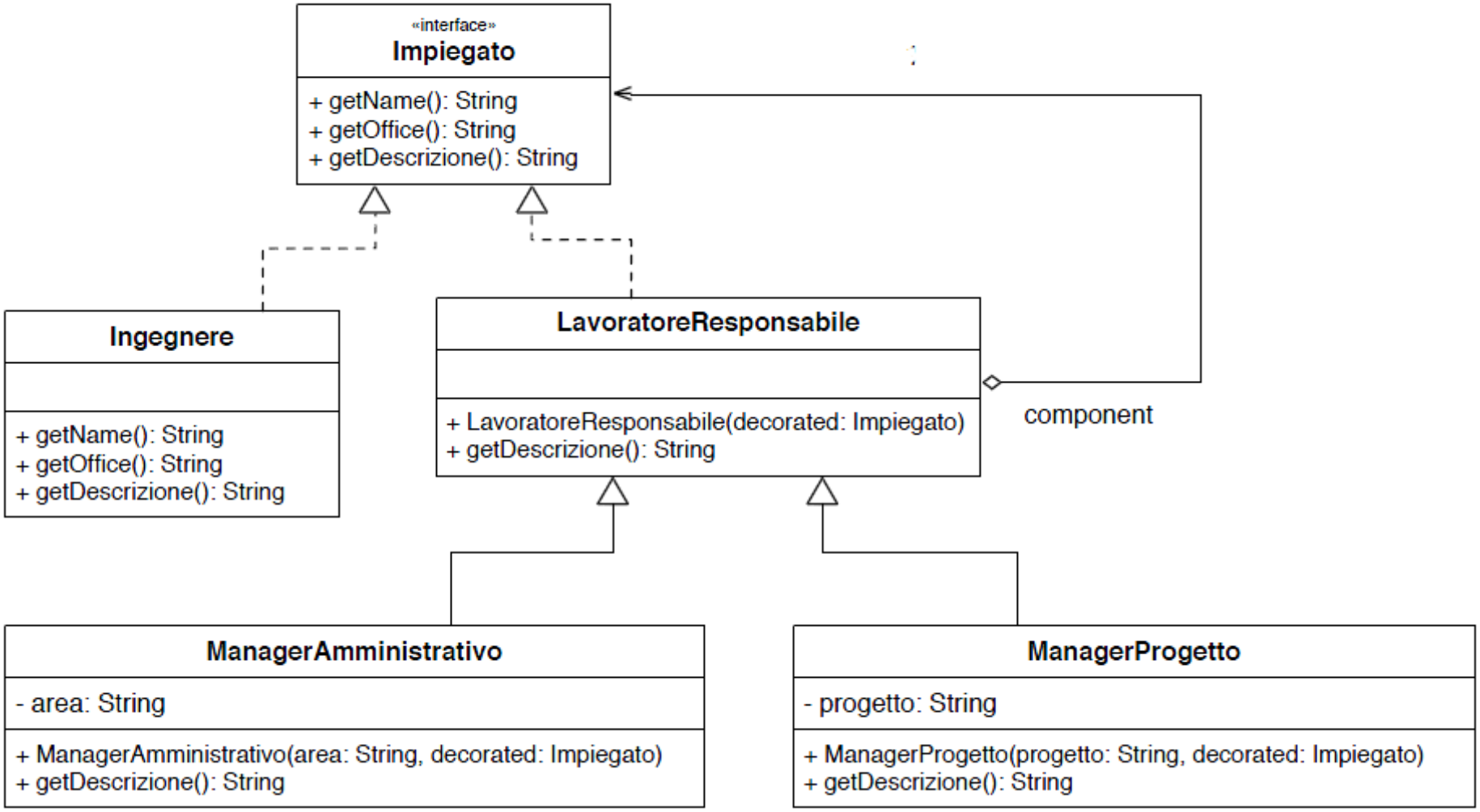
Per esempio, se un ingegnere *ing1* è manager amministrativo di un'area A la stringa "Manager area: A" viene concatenata alla descrizione ritornata dall'invocazione del metodo getDescrizione(). Se a *ing1* viene anche aggiunta la funzionalità di manager amministrativo dell'area B,, il metodo getDescrizione() invocato su *ing1* concatena la stringa "Manager area: B" alla descrizione di *ing1*.
Infine se a *ing1* viene aggiunta la funzionalità manager del progetto P1, la stringa ottenuta invocando il metodo getDescrizione() sull'oggetto *ing1* sarà concatenata la stringa "Manager di Progetto P1" a ciò che ritornava getDescrizione().
Come evidenziato dagli esempi, un ingegnere può essere manager amministrativo di più aree e/o manager di più progetti.

Impiegati (TDE)

1. Si modelli, attraverso un diagramma delle classi UML e un design pattern opportuno, una soluzione al problema sopra presentato.
2. Si scriva anche la struttura del codice Java risultante. Ovvero, si definiscano le classi identificate al passo precedente, le loro relazioni e le intestazioni dei metodi principali.
3. Si scriva il codice Java che crea un'istanza di un oggetto ingegnere con funzionalità di manager amministrativo per l'area A e per l'area B e project manager del progetto P1.

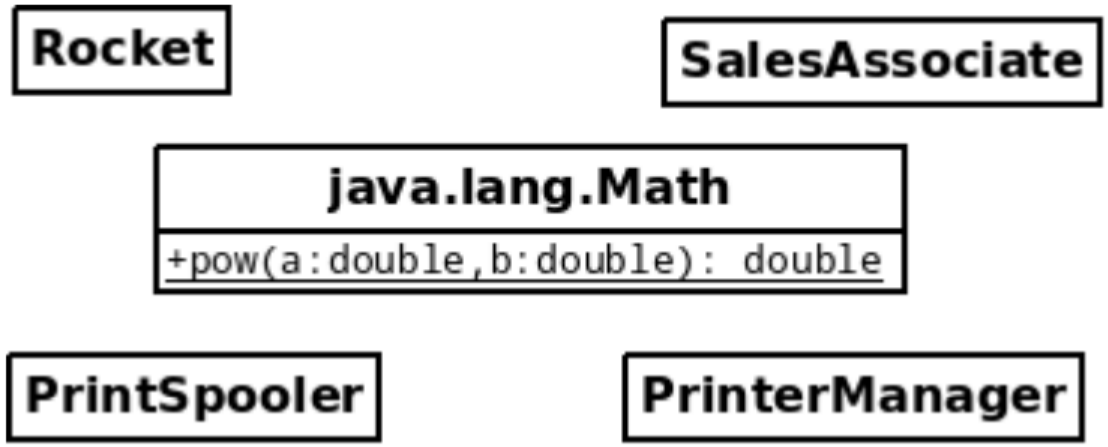
Impiegati (TDE)

È possibile usare il pattern Decorator



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.

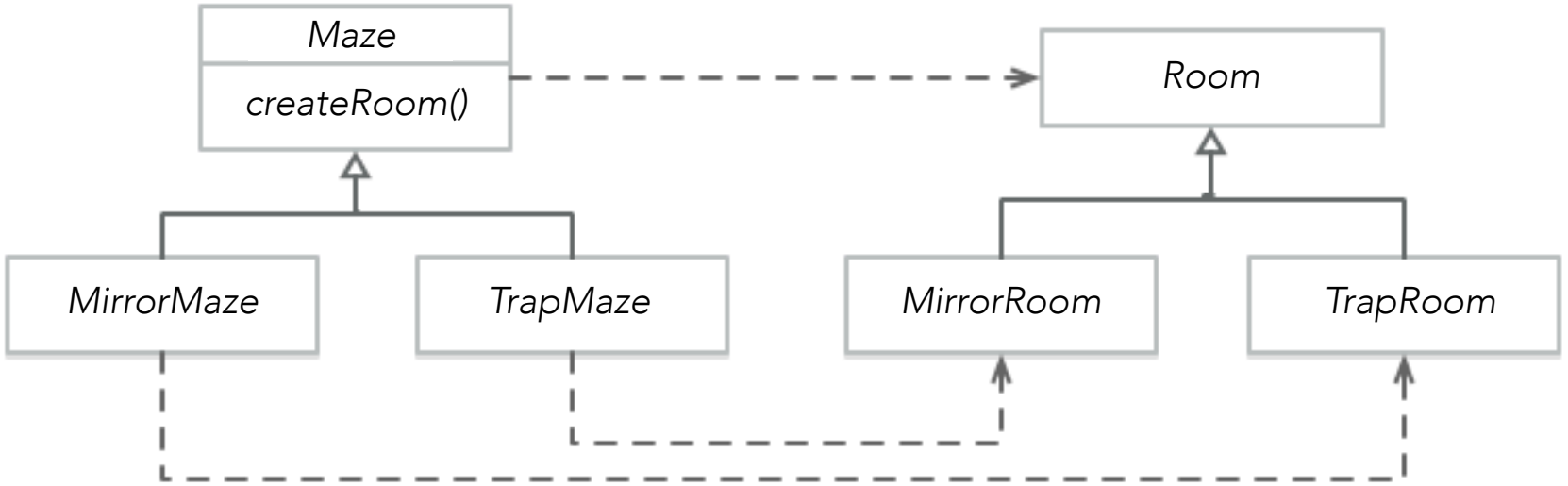
PrintSpooler: lo spooler di stampante è in ogni stampante, dunque non può essere un Singleton.

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

Labirinto

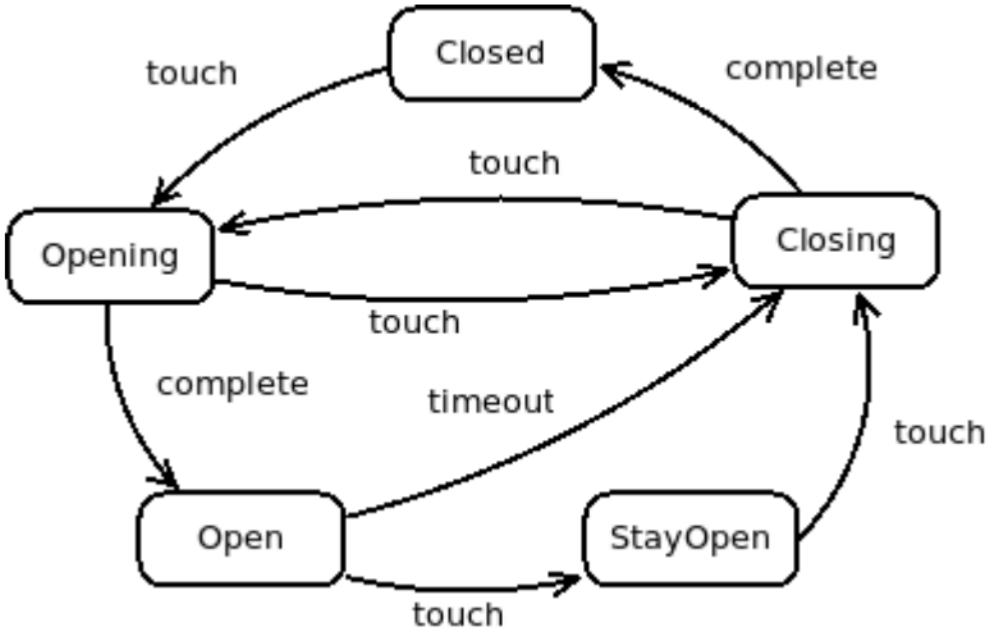
Rappresentare in UML ed implementare in Java un insieme di classi che permettano la creazione di un labirinto. Un labirinto è composto da stanze connesse tra di loro (si ometta la parte di connessione tra stanze nell'implementazione). Ogni labirinto è composto da stanze dello stesso tipo (stanze a specchi, stanze con trappole, etc.) in numero pari a 50. Il costruttore di labirinto deve creare le 50 stanze. Usare un adeguato design pattern.

Labirinto

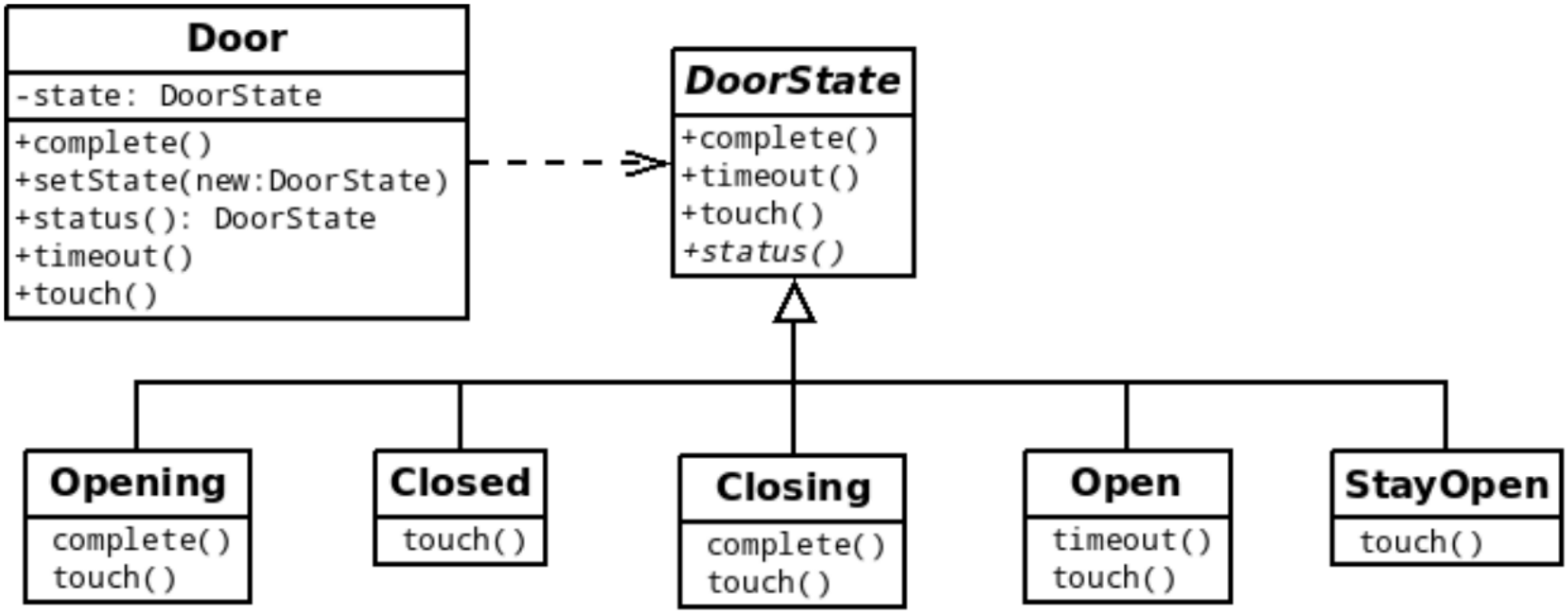

Maze
createRoom()
Room
MirrorMaze
TrapMaze
MirrorRoom
TrapRoom

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()

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

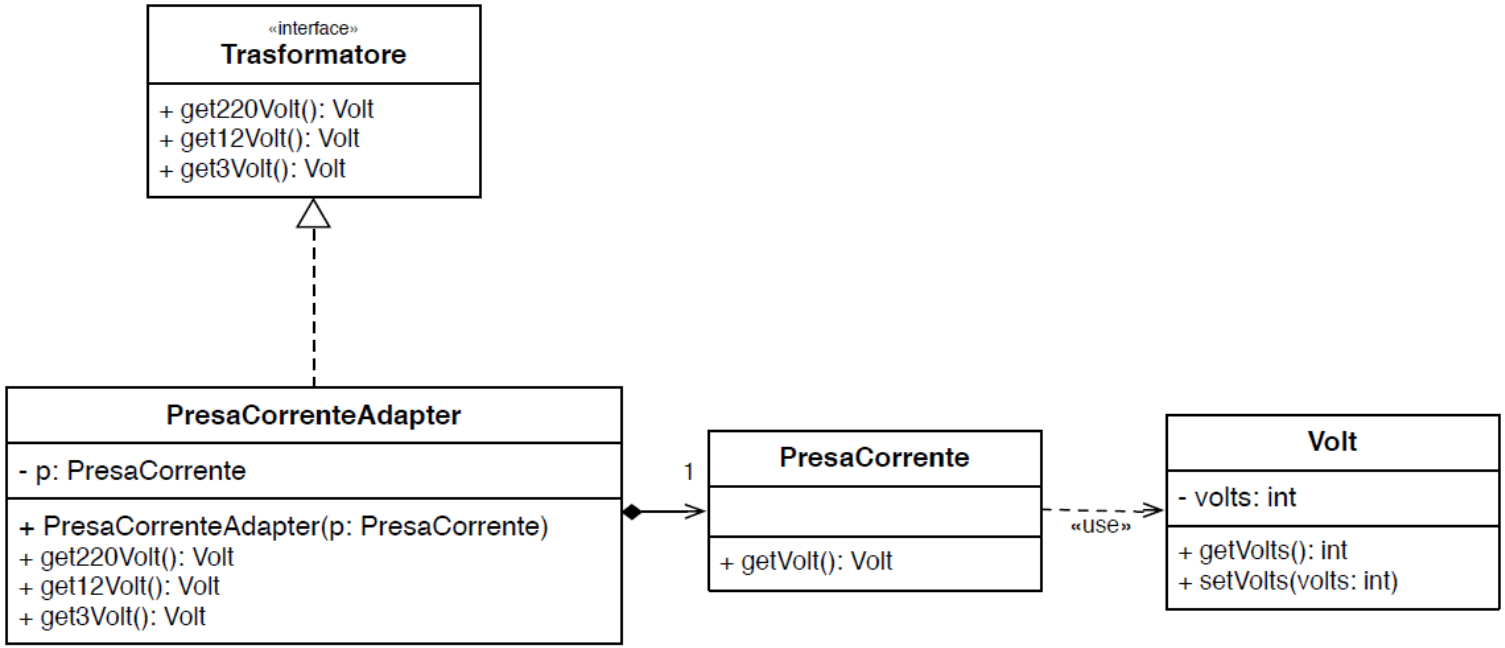
```
public interface Trasformatore {
        public Volt get220Volt();
        public Volt get12Volt();
        public Volt get3Volt();
}
```

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();
}
```

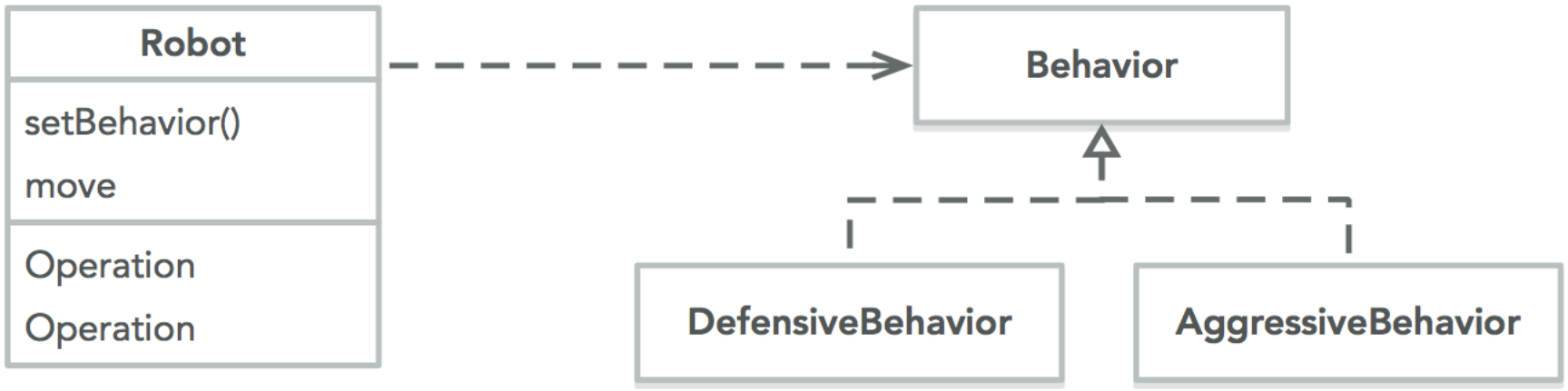

«interface»
Trasformatore
+ get220Volt(): Volt
+ get12Volt(): Volt
+ get3Volt(): Volt
PresaCorrenteAdapter
- p: PresaCorrente
+ PresaCorrenteAdapter(p: PresaCorrente)
+ get220Volt(): Volt
+ get12Volt(): Volt
+ get3Volt(): Volt
1
PresaCorrente
+ getVolt(): Volt
«use»
Volt
- volts: int
+ getVolts(): int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

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



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.