Sistema di domotica

Anno accademico 2023/2024
Corso di metodologie di programmazione

Nicola Papini
11/02/2024
7126857

Indice

1 Introduzione al progetto

Il progetto riguarda lo sviluppo di un prototipo di sistema software che simuli il comportamento di un mini sistema di domotica capace di occuparsi del controllo automatico dell'illuminazione. Questo avviene attraverso l'utilizzo di due diversi sensori:

- un sensore di movimento progettato per notificare il sistema sia quando rileva un movimento, sia quando non rileva alcun movimento per un periodo di tempo predefinito
- un sensore di luminosità progettato per rilevare variazioni di luce nell'ambiente, permettendo così un adeguamento automatico dell'intensità delle luci interne in base alla luminosità rilevata nella stanza.

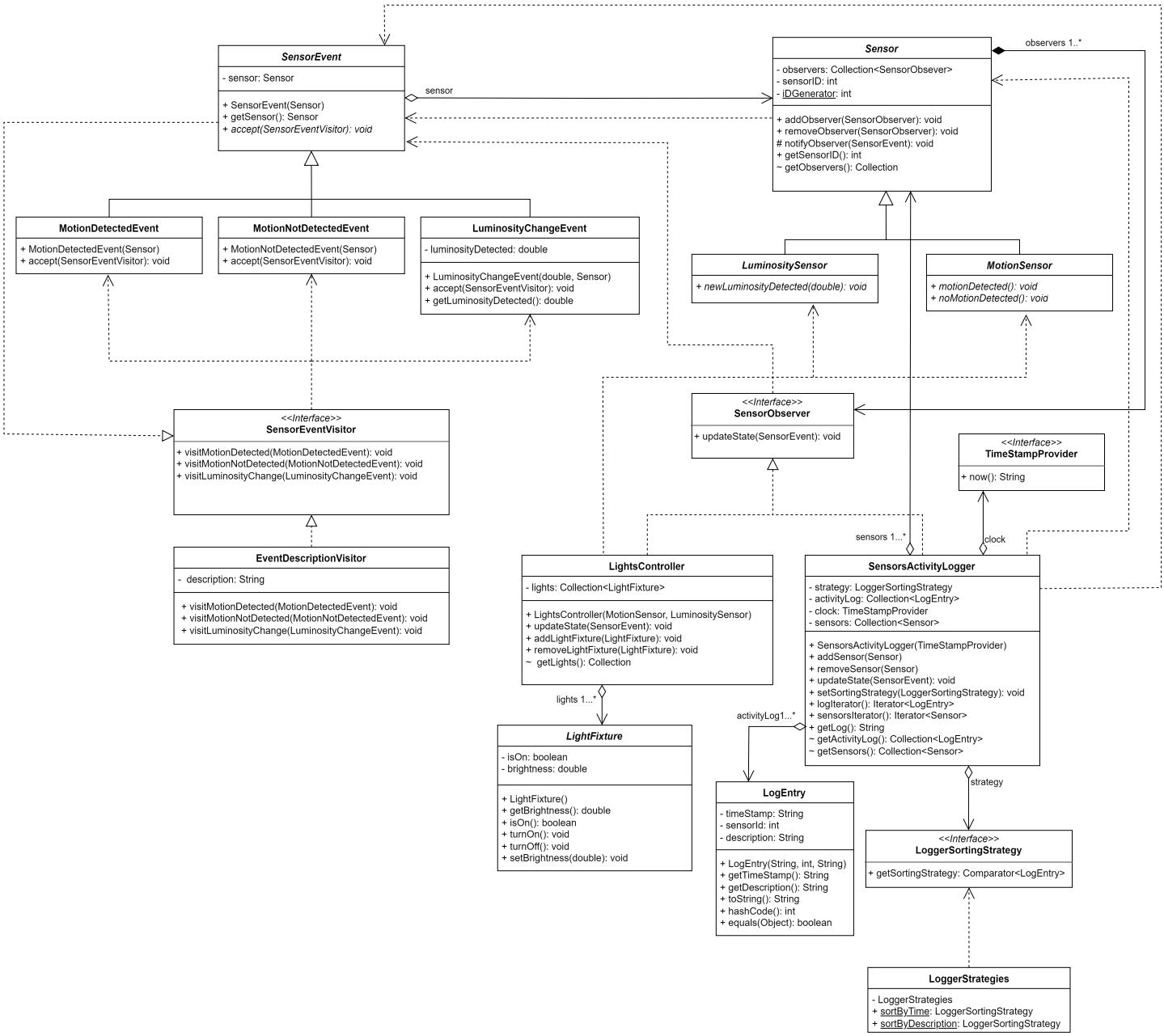
Inoltre, il sistema è dotato di un servizio di logging. Questo consente all'utente di accedere e recuperare i dati raccolti dai vari sensori nel corso del tempo. Le informazioni possono essere organizzate secondo diversi ordinamenti: è possibile visualizzarle in base alla data di registrazione dell'evento, in ordine cronologico crescente, o per tipo di evento rilevato. Il sistema, mantenuto per semplicità a poche funzionalità, è stato pensato in modo da essere facilmente estendibile a nuove funzionalità. Ad esempio, potrebbe essere aggiunto un sistema di controllo della temperatura o un sistema di sicurezza, riutilizzando la struttura di notifica già esistente.

2 Patterns

I patterns applicati nel progetto sono:

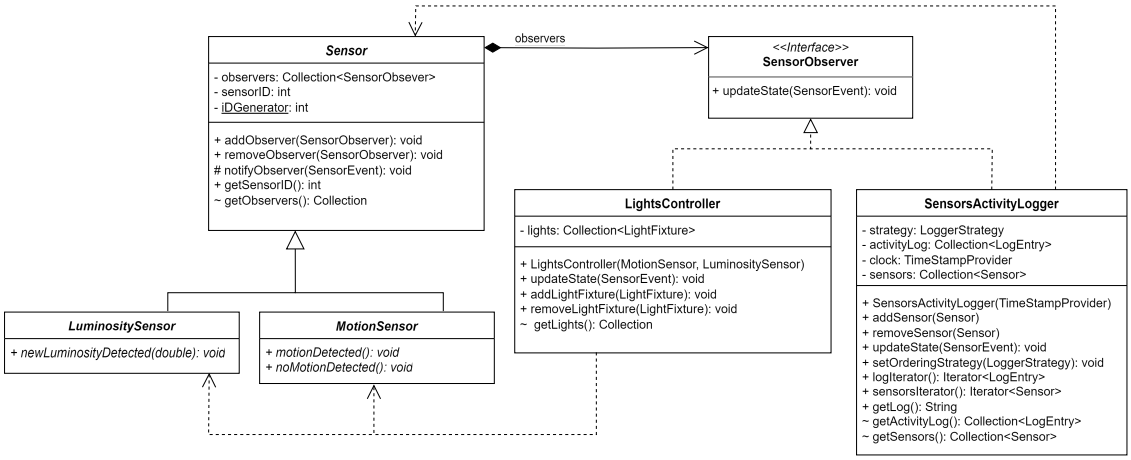
- Observer
- Visitor
- Strategy

La scelta di questi tre patterns è dovuta alla loro ottima applicabilità nel descrivere le funzionalità del progetto. In particolare, il pattern Observer è estremamente adatto per modellare le interazioni tra i sensori e i loro utilizzatori. Di seguito una vista di classi e interfacce in diagramma UML:


SensorEvent
- sensor: Sensor
+ SensorEvent(Sensor)
+ getSensor(): Sensor
+ accept(SensorEventVisitor): void
sensor
Sensor
- observers: Collection<SensorObsever>
- sensorID: int
- IDGenerator: int
+ addObserver(SensorObserver): void
+ removeObserver(SensorObserver): void
# notifyObserver(SensorEvent): void
+ getSensorID(): int
~ getObservers(): Collection
observers 1..*
MotionDetectedEvent
+ MotionDetectedEvent(Sensor)
+ accept(SensorEventVisitor): void
MotionNotDetectedEvent
+ MotionNotDetectedEvent(Sensor)
+ accept(SensorEventVisitor): void
LuminosityChangeEvent
- luminosityDetected: double
+ LuminosityChangeEvent(double, Sensor)
+ accept(SensorEventVisitor): void
+ getLuminosityDetected(): double
LuminositySensor
+ newLuminosityDetected(double): void
MotionSensor
+ motionDetected(): void
+ noMotionDetected(): void
<<Interface>>
SensorEventVisitor
+ visitMotionDetected(MotionDetectedEvent): void
+ visitMotionNotDetected(MotionNotDetectedEvent): void
+ visitLuminosityChange(LuminosityChangeEvent): void
<<Interface>>
SensorObserver
+ updateState(SensorEvent): void
<<Interface>>
TimeStampProvider
+ now(): String
EventDescriptionVisitor
- description: String
+ visitMotionDetected(MotionDetectedEvent): void
+ visitMotionNotDetected(MotionNotDetectedEvent): void
+ visitLuminosityChange(LuminosityChangeEvent): void
LightsController
- lights: Collection<LightFixture>
+ LightsController(MotionSensor, LuminositySensor)
+ updateState(SensorEvent): void
+ addLightFixture(LightFixture): void
+ removeLightFixture(LightFixture): void
~ getLights(): Collection
sensors 1...*
clock
SensorsActivityLogger
- strategy: LoggerSortingStrategy
- activityLog: Collection<LogEntry>
- clock: TimeStampProvider
- sensors: Collection<Sensor>
+ SensorsActivityLogger(TimeStampProvider)
+ addSensor(Sensor)
+ removeSensor(Sensor)
+ updateState(SensorEvent): void
+ setSortingStrategy(LoggerSortingStrategy): void
+ logIterator(): Iterator<LogEntry>
+ sensorsIterator(): Iterator<Sensor>
+ getLog(): String
~ getActivityLog(): Collection<LogEntry>
~ getSensors(): Collection<Sensor>
lights 1...*
activityLog1...*
strategy
LightFixture
- isOn: boolean
- brightness: double
+ LightFixture()
+ getBrightness(): double
+ isOn(): boolean
+ turnOn(): void
+ turnOff(): void
+ setBrightness(double): void
LogEntry
- timeStamp: String
- sensorId: int
- description: String
+ LogEntry(String, int, String)
+ getTimeStamp(): String
+ getDescription(): String
+ toString(): String
+ hashCode(): int
+ equals(Object): boolean
<<Interface>>
LoggerSortingStrategy
+ getSortingStrategy: Comparator<LogEntry>
LoggerStrategies
- LoggerStrategies
+ sortByTime: LoggerSortingStrategy
+ sortByDescription: LoggerSortingStrategy

2.1 Observer

Il pattern Observer è utilizzato nel progetto per gestire le notifiche dei sensori quando rilevano un nuovo evento. La scelta del pattern è giustificata dal fatto che la struttura di oggetti sensori deve essere in grado di notificare i loro rilevamenti ai sistemi interessati in modo che questi possano aggiornare il loro stato in base alla notifica. Inoltre, grazie al pattern, il sensore non deve sapere a priori chi notificare, questo rende la struttura riusabile nel caso ci siano nuovi sistemi interessati all'attività del sensore. In questo caso i partecipanti sono la classe astratta Sensor, come subject, insieme alle sue due implementazioni `LuminositySensor` e `MotionSensor` e l'interfaccia `SensorObserver`, come observer, insieme alle sue due implementazioni concrete `LightsController` e `SensorsActivityLogger`. Le implementazioni dei due sensori sono astratte poiché non rappresentabili in modo concreto senza avere effettivamente dei sensori che mandino segnali, entrambe sono comunque testate attraverso due mock nei test. Inoltre, quando un sensore genera una notifica associa ad essa un `SensorEvent`, ossia un oggetto che descrive l'evento generato e il sensore da cui è partita la notifica.



Un esempio di interazione è il seguente:

1. Si istanziano due sensori e si aggiunge ad esso un observer. Ho considerato i due sensori come implementazioni concrete per migliore leggibilità e comprensione.

```
public void test(){
    MotionSensor motionSensor = new MotionSensor();
    LuminositySensor luminositySensor = new LuminositySensor();
    LightsController lightsController = new LightsController(
                                            motionSensor,
                                            luminositySensor);
}
```

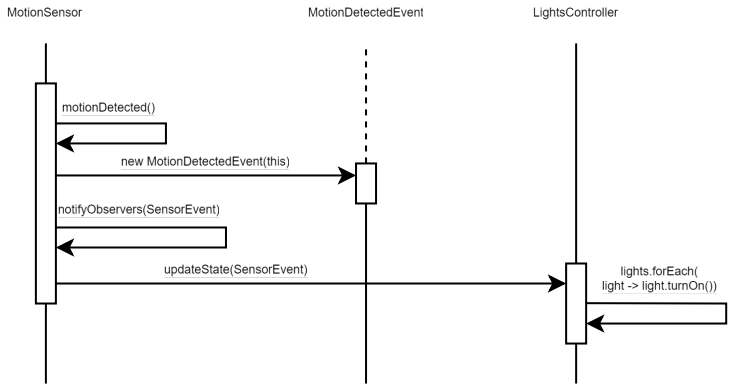
`lightsController` è aggiunto come observer dei due sensori all'interno del suo costruttore.

2. Per esempio, il sensore di movimento rileva un nuovo movimento e lo segnala attraverso il metodo

```
motionSensor.motionDetected();
```

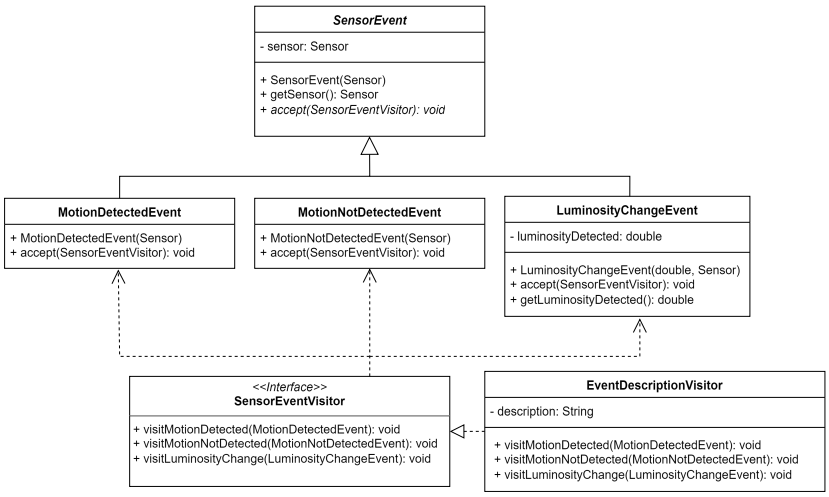
3. Il sensore genera un `MotionDetectedEvent` e, succesivamente, notifica tutti i suoi observer, in questo caso `lightsController`, passando come argomento l'evento generato.

4. Gli observer ricevono la notifica ed eseguono di conseguenza un'azione. Questa viene scelta dinamicamente attraverso l'utilizzo del pattern Visitor.



2.2 Visitor

Il pattern Visitor è stato utilizzato per gestire i diversi tipi di eventi dei sensori. Il pattern è stato implementato in due classi, `LightsController` e `SensorsActivityLogger`, come classe anonima per definire le operazioni specifiche per ogni tipo di evento del sensore. Inoltre, è stata definita una classe concreta `EventDescriptionVisitor` per fornire una descrizione, come `String`, degli eventi. Le classi `MotionDetectedEvent`, `MotionNotDetectedEvent` e `LuminosityChangeEvent` implementano la classe astratta `SensorEvent` e ognuna di queste ha un metodo `accept(SensorEventVisitor)` che, tramite il double dispatch, permette di invocare il metodo corretto in base al tipo di evento del sensore. In particolare, il vantaggio principale del pattern è che evita l'uso di istruzioni instance of o switch, che sarebbero meno flessibili e più difficili da mantenere. Inoltre, rende più facile l'aggiunta di nuove operazioni sulla struttura degli eventi.



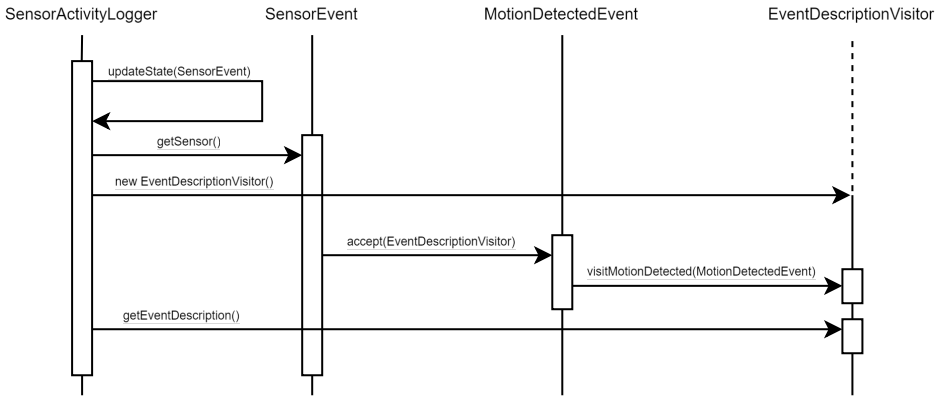
Un esempio di interazione è il seguente:

1. Si istanza un nuovo logger e si mette come observer di un sensore:

```
public void test(){
    SensorsActivityLogger logger = new SensorsActivityLogger();
    MotionSensor sensor = new MotionSensor()
    logger.addSensor(sensor);
}
```

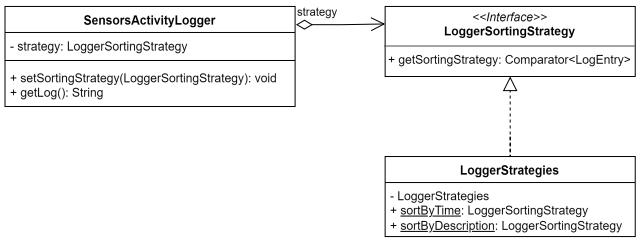
2. Il sensore rileva un nuovo movimemento e lo segnala al `logger`, quindi viene invocato il metodo `updateState(SensorEvent)` dove, in questo caso, l'evento è un `MotionDetectedEvent`

3. Di seguito l'interaction diagram:



2.3 Strategy

La scelta del pattern deriva dalla necessità di fornire diverse versioni del log, ognuna ordinata secondo un diverso criterio. Infatti grazie al riferimento all'oggetto di tipo `LoggerSortingStrategy` è possibile definire il criterio in modo flessibile. Inoltre, se necessario, è anche possibile definire facilmente nuovi criteri di ordinamento. Il criterio di default è quello dell'ordinamento in ordine cronologico crescente però è possibile modificarlo a runtime tramite `setSortingStrategy(LoggerSortingStrategy)`. L'interfaccia `LoggerSortingStrategy` definisce il metodo `getSortingStrategy()` che restituisce un `Comparator<LogEntry>` che verrà utilizzato ordinare il log prima di essere restituito dal metodo `getLog()`. Per ridurre la complessità del codice la logica degli ordinamenti è stata incapsulata nella classe `LoggerStrategies`. Questa serve solo per istanziare nuovi strategy, attraverso static factory method, per questo il suo costruttore è stato posto a private.



3 Dettaglio classi e Testing

3.1 Sensor

Il sensore viene identificato attraverso la variabile di istanza `sensorID` il cui valore viene automaticamente assegnato nel costruttore utilizzando la variabile static `iDGenerator`. Quest'ultima viene incrementata di uno ogni volta che un nuovo sensore viene creato. Inoltre, la classe mette a disposizione le funzionalità di base per gestire gli observer. Di questa classe è stato testato:

- il funzionamento di `addObserver(SensorObserver)` e `removeObserver(SensorObserver)` controllando che gli observer vengano aggiunti e rimossi correttamente e che, in caso l'input fosse non valido, venga generata la corretta eccezione
- la generazione dell'ID del sensore venga correttamente generata, in particolare che ogni successivo sensore abbia l'ID maggiore del precedente.

3.2 LuminositySensor/MotionSensor

Per quanto riguarda queste due classi, esse mettono a disposizione i metodi di notifica di base. In particolare, per quanto riguarda `MotionSensor` ho scelto di non implementare nessuno stato, essendo questo implicito nel tipo di notifica inviata. Nel caso del sensore di luminosità, lo stato del sensore, ovvero il nuovo valore rilevato, è incluso direttamente nell'evento generato dalla notifica a cui l'observer ha accesso diretto. Le loro implementazioni sono state "mockate" nei test. Ho scelto di testare le implementazioni mock poiché contengono una certa logica e non si limitano esclusivamente a setter o getter. Quindi, poiché queste implementazioni sono utilizzate in molti test, ho ritenuto opportuno garantirne il corretto funzionamento, assicurandomi che il tipo di evento generato fosse appropriato.

3.3 LightFixture

`LightFixture` è la classe astratta che definisce il comportamento di una generica luce. L'unica implementazione concreta è data da `MockSmartBulb` all'interno dei test. Ho scelto di utilizzare un'astrazione con una sola implementazione per rendere il sistema facilmente espandibile a nuovi tipi di luci, ognuno potenzialmente con diversi protocolli per l'impostazione della luminosità. Infatti tutti i metodi sono stati dichiarati final a parte `setBrightness(double)`.

3.4 LightsController

`LightsController` è responsabile della gestione di un insieme di luci, il cui comportamento è regolato dai due sensori che un oggetto di questa classe utilizza. Il controller definisce al suo interno la variabile d'istanza `lights`, collezione delle luci di cui il controller è responsabile. Inoltre sono a disposizione le funzionalità di base per aggiungere e rimuovere una luce. L'`updateState(SensorEvent)` esegue lo spegnimento di tutte le luci in caso riceva un `MotionNotDetectedEvent`, l'accensione delle luci per l'evento opposto, oppure la regolazione dell'intensità delle luci quando riceve un `LuminosityChangeEvent`. Di questa classe è stato testato il corretto funzionamento dei metodi di gestione delle luci, in modo analogo a quanto fatto per il `Sensor`, e il corretto comportamento delle luci in risposta a una notifica di un sensore, testando in isolamento ognuno dei possibili scenari.

3.5 LogEntry

Questa è la classe che descrive una generica entry del log. Un'entry è definita da un TimeStamp, l'ID del sensore che l'ha generata e una descrizione dell'evento. La scelta di implementarla come classe a parte e non come classe interna del logger è stata fatta per non appensantire troppo la classe `SensorsActivityLogger`, cosa che avrebbe peggiorato la leggibilità/manutenibilità. Oltre ai metodi di base sono stati aggiunti anche `toString()`, `equals(Object)` e `hashCode()`. Tutti e tre i metodi sono testati nella classe `LogEntryTest`.

3.6 SensorsActivityLogger

Questa classe è responsabile di tener traccia delle attività di tutti i sensori di cui è observer e, su richiesta dell'utente, di generare un log. La scelta di avere un criterio di ordinamento di default istanziato direttamente nel costruttore, invece di utilizzare dependency-injection, ha senso in quanto il metodo `sortByTime()`, che implementa lo strategy, difficilmente cambierà nel tempo. Le varie attività sono mantenute all'interno di una `Collection` di `LogEntry`. Quindi, ogniqualvolta viene generata una segnalazione da un sensore osservato, il log genera una nuova LogEntry con il TimeStamp corrente, l'ID del sensore e la descrizione dell'evento. Il TimeStamp è generato dall'astrazione `TimeStampProvider` che mette a disposizione il metodo `now()`, questo restituisce il TimeStamp come stringa in formato YY-MM-DD hh:mm:ss. La descrizione, invece, è generata da un visitor che crea una stringa descrittiva dell'evento. Sono a disposizione metodi di utilità quali due iterator per iterare sulla lista dei log o dei sensori e il metodo `resetLog()` che svuota il registro di tutte le attività registrate.

Per quanto riguarda i test è stato necessario "mockare" il TimeStampProvider. Il mock permette di impostare il TimeStamp a un valore arbitrario attraverso il metodo `setTime(String)`. Nei test si controlla:

- L'aggiunta e rimozione dei sensori, controllando che il sensore venga aggiunto correttamente alla lista dei sensori e che il logger venga aggiunto alla lista degli observer del sensore aggiunto. Analogamente è stato fatto per la rimozione
- La generazione e registrazione della `LogEntry` in corrispondenza dei vari eventi, assicurandosi che l'entry creata venga aggiunta nel log correttamente
- Il funzionamento del `getLog()` testandolo sia nel caso in cui si richieda un ordinamento cronologico crescente che per descrizione dell'evento
- il metodo `resetLog()` verificando che, una volta svuotata la lista dei sensori, il logger non sia più observer di nessuno di essi.

3.7 SensorEventVisitor

Questa è la classe che realizza il pattern Visitor. I test sono stati realizzati sia per la sua implementazione concreta `EventDescriptionVisitor`, controllando che la stringa di descrizione dell'evento fosse generata correttamente, che per le due implementazioni anonime, testate nei test delle rispettive classi di appartenenza.