Sistema Domotico

A CURA DI MARCO SARTORELLI, DAVIDE FOGAGNOLO, MATTEO MARTANI

Sommario:

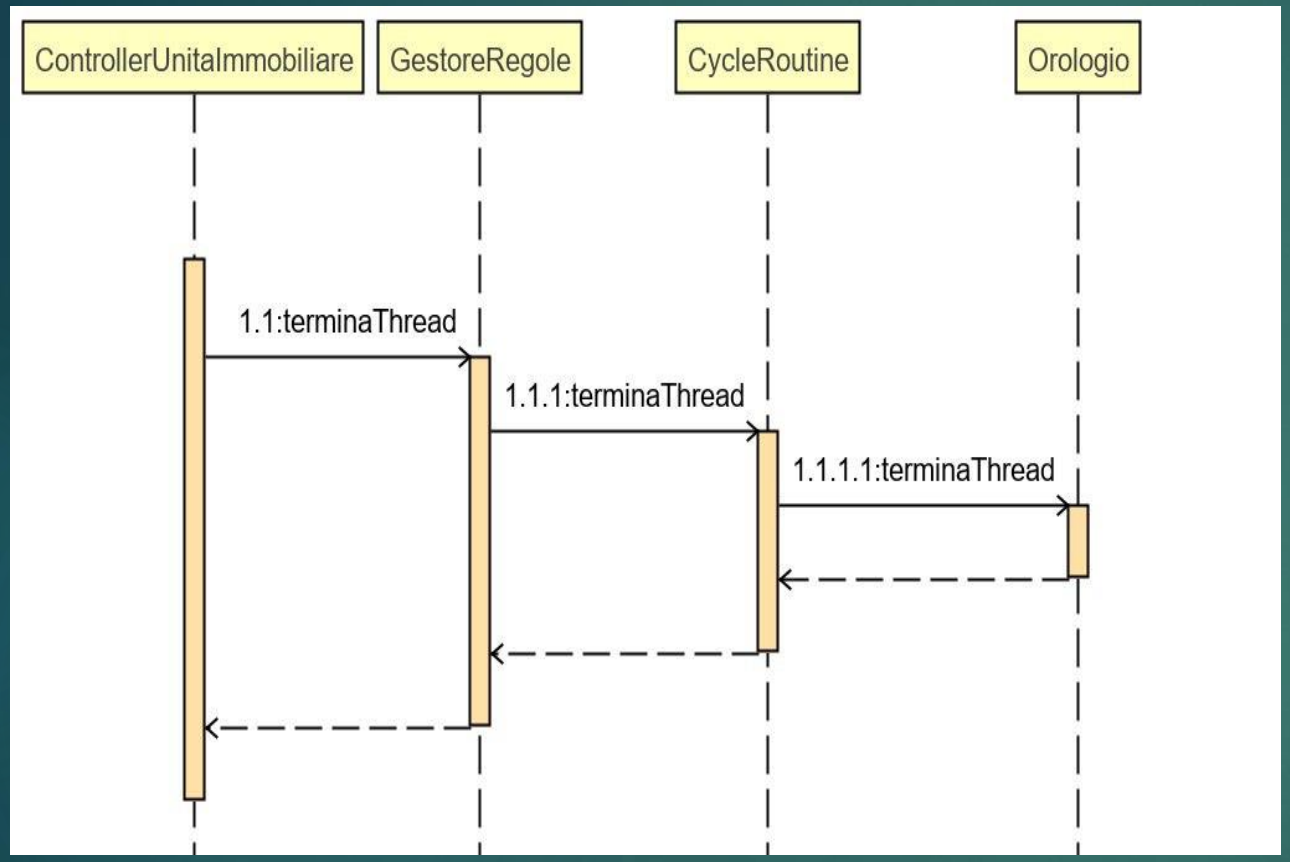
Grasp: Controller

- 1) Come individuo un oggetto a cui affidare una determinata operazione?
- 2) Il pattern controller svolge l'importante compito di assegnare la responsabilità di eventi che coinvolgono una chiamata di sistema a una classe che in una situazione di un caso d'uso andrà a rappresentare l'intero sistema.
- 3) da ricordare: il controller non svolge di per sé molto lavoro, il suo compito è quello di delegare

Grasp: Controller

- Sono presenti due controller distinti.
- 5) Il controller ''Unità Immobiliare'' è contenuto all'interno del package ''Unità immobiliare'' ed è l'unica classe di quel package che ha metodi ad accesso di pacchetto quindi tutte le richieste esterne passeranno da lì e avrà il compito di reindirizzare le tali richieste alla classe a cui la responsabilità è affidata.
- 6) Il controller ''Sistema Domotico'' funziona esattamente allo stesso modo del controller precedente solo ad un livello più esterno.
  E' colui che fa da tramite tra il package ''Applicazione'' e il main.

Grasp: Controller



In questa situazione specifica possiamo vedere come il controller di Unità Immobiliare delega la responsabilità specifica del metodo ''termina thread'' in una serie di passaggi.

Grasp: Creator

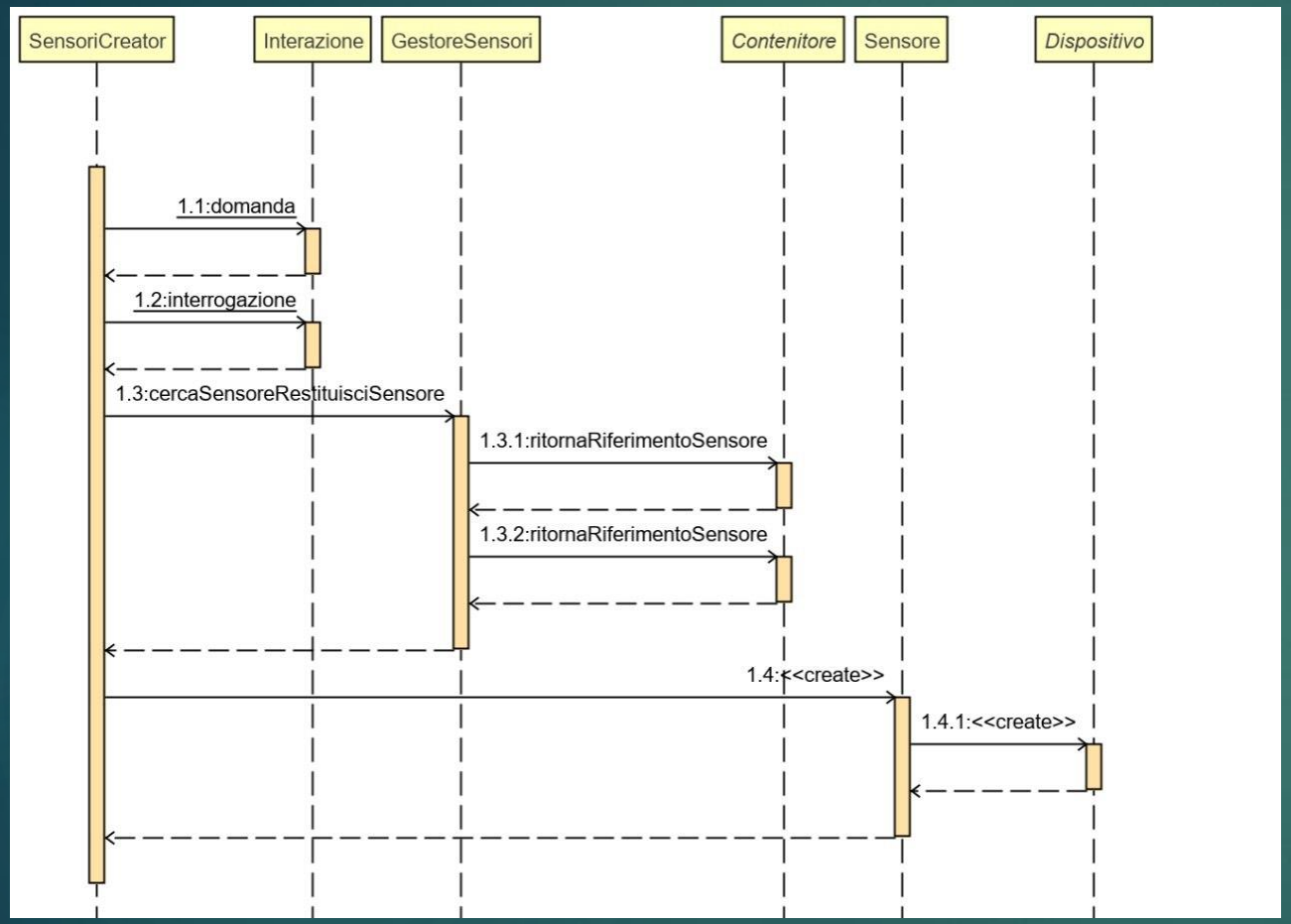
- 1) Come decido a chi affidare la responsabilità di creare oggetti di una classe A?
- 2) Potrei affidare tale compito alla classe B qualora essa contenga, registri, usi strettamente o contenga informazioni iniziali per A.
- 3)Questo pattern in particolare è molto importante in quanto l'azione specifica di creazione di un oggetto è una delle pratiche più diffuse e comuni in un sistema object oriented.
- 4)Un buon utilizzo di Creator può facilmente portare a un basso accoppiamento, alta coesione, incapsulamento, propensione al riuso

Grasp: Creator

- 5) alla classe ‘’CategorieCreator’’ viene affidata la responsabilità di creare oggetti di tipo ‘’CategoriaSensore’’ e ‘’CategoriaAttuatore’’.

- 6) mentre invece alla classe ‘’SensoriCreator’’ viene affidata la responsabilità di creare oggetti di tipo ’’Sensore’’ e ‘’Attuatore’’

Grasp: Creator



In questo specifico caso possiamo vedere come il metodo ''create'' viene invocato da ''SensoriCreator'' per creare un oggetto di tipo ''Sensore''.
Alla classe in questione poi sono affidate anche altre responsabilità

Solid: Single Responsability

- 1)Quante responsabilità dovrebbe avere una classe?
- 2)A una classe viene assegnata una responsabilità quando ha le informazioni per poter soddisfarla. Secondo Single Responsability, ogni classe dovrebbe essere specializzata in una singola responsabilità.
- 3)Nel caso in cui una classe ha diverse responsabilità e ne modifico una, potrebbero esserci dei cambiamenti non voluti anche sulle altre. Dunque avrei un alto accoppiamento e progetti fragili.
- 4)Single Responsability quindi ha come scopo quello di mantenere un'alta coesione tra classi.

Solid: Single Responsability

- 5)Le responsabilità della classe ''SistemaDomotico'' sono state assegnate ad altre classi, e nello specifico quelle a singola responsabilità sono:

1. La classe ''GestoreFlusso'' ha il compito di guidare l'utente nelle diverse operazioni che può fare all'interno della sezione ''SistemaDomotico''.
2. La classe ''LetturaFile'' legge un file di testo e trasforma in oggetti le frasi lette nel file, come richiesto da uno dei requisiti.
3. La classe ''Salvataggio'' che si occupa di ripristinare i dati in avvio e salvarli in chiusura.
4. La classe ''TerminazioneSistema'' che interrompe i thread in esecuzione per una corretta chiusura dell'applicativo

Solid: Single Responsability

- 6) Le responsabilità della classe ''Unità Immobiliare'' sono state assegnate ad altre classi, e nello specifico quelle a singola responsabilità sono:

1. La classe ''GestoreFlusso'' ha il compito di guidare l'utente nelle diverse operazioni che può fare all'interno della sezione ''UnitàImmobiliare'' (diverso da quello di 'SistemaDomotico'').
2. La classe ''GestoreSensori'' gestisce i diversi sensori, se attivarli, disattivarli
3. La classe ''GestoreRegole'' gestisce le diverse regole, se attivarle, disattivarle
4. La classe ''LetturaDaFile'' che trasforma linee di testo correttamente scritte in oggetti unità immobiliare

Solid: Single Responsability

Classe ''SistemaDomotico'' contiene unicamente attributi e metodi legati alla gestione delle unità immobiliari, e categorie sensori/attuatori associate ad esse

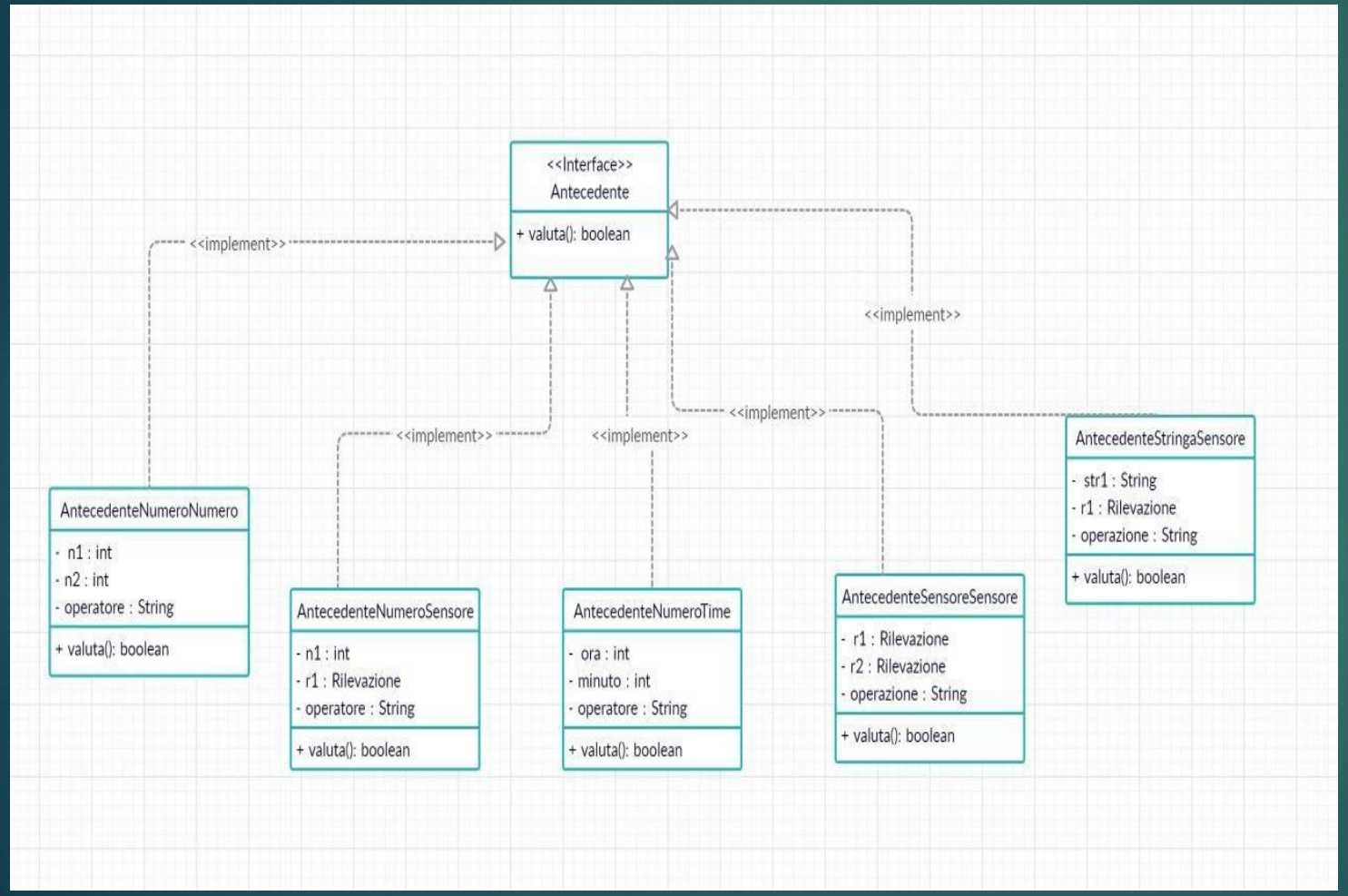
Solid: Open Closed

- 1) Quanto un progetto deve essere in grado di estendere il proprio comportamento senza stravolgimenti del codice?
- 2) Si vogliono creare dal principio moduli che poi possono essere estesi come comportamento a fronte di nuovi requisiti, ma evitando di modificare il codice di base che crea il modulo.
- 3)Si cerca di modificare il codice in maniera limitata.

Solid: Open Closed

- 4) E' stato applicato nel package ''DipendenteDalTempo'' con classi dedicate alla gestione delle regole. Una regola è composta da una parte antecedente, una conseguente, se la prima parte e' corretta(restituisce true) va eseguita la seconda.
- 5)Dato che la parte antecedente puo' essere definita con diversi tipi di dati e non si vuole dipendere da un controllo condizionale (if o switch) credo un interfaccia Antecedente che verra' poi estesa da diverse classi, ognuna delle quali rappresenta una combinazione di dati ammissibili da antecedente.
  Se nel tempo i dati aumentano devo solo aggiungere classi concrete senza toccare il resto del codice
- 6)Ogni regola ha un riferimento all'interfaccia con il quale chiama il metodo valuta, ignara di come questo verra' implementato.

Solid: Open Closed



Le classi che implementano l'interfaccia devono implementare il metodo associato al tipo di comportamento

Gof: Singleton

- 1)Il singleton è un design pattern di natura creazionale che ha lo scopo di garantire che di una classe specifica venga creata una e una sola istanza.
- 2)Questo principio viene realizzato creando una classe Singleton con costruttore privato in modo che l'istanziazione non sia diretta.
- 3)Viene fornito poi un metodo di query detto ''GetInstance'' statico, in modo che sia accessibile in ogni punto del mio programma, che restituisce l'istanza della classe solo qualora questa non sia ancora stata creata.

Gof: Singleton

- 4)Il pattern è applicato sulla classe ''ControllerSistemaDomotico''. L'approccio scelto è quello goloso: nel momento in cui le classi vengono caricate prova a ripristinare un salvataggio precedente e, qualora esista, una sua istanza assume quel valore.
- 5)Il main appena avviato prova a richiedere un'istanza, quindi se l'implementazione golosa ha successo ottengo l'istanza e la restituisco al main altrimenti nel caso in cui non c'era alcun salvataggio creo l'istanza (sempre che questa sia uguale a NULL).

Gof: Singleton


```
public static ControllerSistemaDomotico getInstance(){
    if(instance == null) {
        instance = new ControllerSistemaDomotico();
    }
    return instance;
}
```

Da questa immagine possiamo vedere chiaramente il funzionamento del metodo getInstance() presente nella classe ''ControllerSistemaDomotico''.
Il metodo è statico e ritorna un'istanza solo nel caso in cui questa sia nulla.

Gof: Observer

- 1)Come tenere traccia dei cambiamenti di stato o aggiornamenti da parte di un singolo oggetto?
- 2)Definisco Subject e Observer: i primi avranno il compito di avvisare i secondi nel caso in cui ci sia un evento, gli Observer poi reagiranno opportunamente a seconda del cambiamento.
- 3)Per farlo dovrò utilizzare un'interfaccia Observer implementata dai miei ConcreteObserver cioè gli osservatori e un'interfaccia Subject implementata dai miei ConcreteSubject, quest'ultimi avranno il compito di tenere traccia dei vari ConcreteObserver che li monitorano.

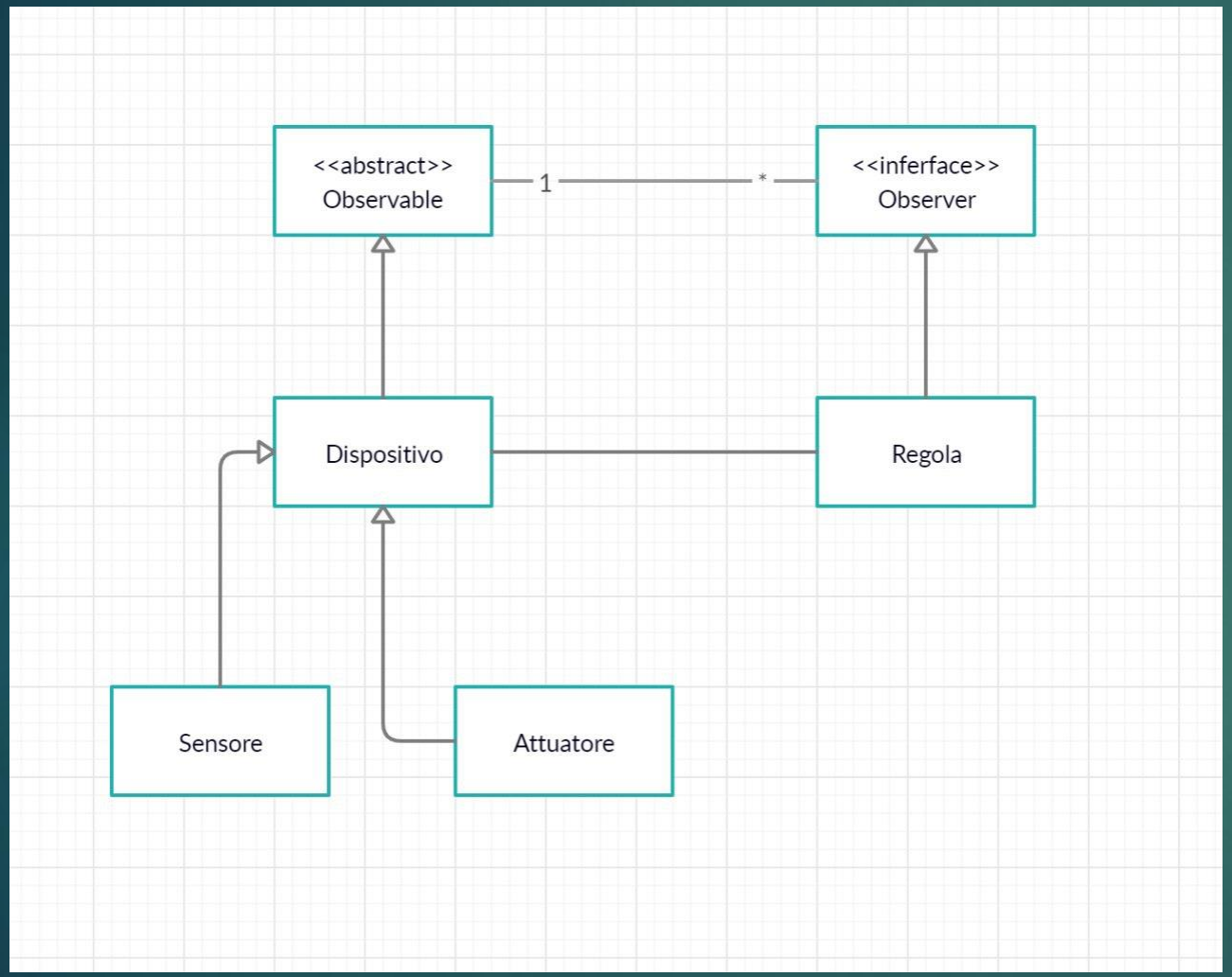
Gof: Observer


```
private boolean antecedente() {
    if((sensoreCondizione[0]!=null&&!sensoreCondizione[0].isAttivo())||
            (sensoreCondizione[1]!=null&&!sensoreCondizione[1].isAttivo()))
        return false;
    else
        return condizione.valuta();
}
```

```
private boolean antecedente() {
    if(activeComponents==0)
        return condizione.valuta();
    else return false;
}
```

- 1)Nel caso in cui un dispositivo, che può essere un sensore o un attuatore, che viene utilzzato in una regola o come antecedente o come conseguente venisse disattivato non c'è bisogno di controllare ulteriormente tale regola: prima dovevamo controllare periodicamente che il componente fosse attivo tramite polling. Usando il pattern Observer però posso semplificare il tutto
- 2)Dispositivo, che è una classe astratta da cui poi sensori e attuatori ereditano, eredita dalla classe Observable. Estendendola acquista più metodi quali addObserver e removeObserver che servono per aggiungere gli oggetti che voglio notificare, setChange ovvero il metodo chiamato prima di notificare se si vuole che i client vengano avvisati e infine notify che avvisa tutti gli Observer che qualcosa è cambiato.

Gof: Observer



Da quest'immagine possiamo vedere il diagramma UML delle classi che il pattern coinvolge.

3)Per quanto riguarda gli oggetti ConcreteObserver sono le regole ad implementare l'interfaccia Observer che al suo interno ha sia antecedente (contiene al più due sensori) che conseguente.
Quando creo questa regola chiede a tutti i sensori il loro stato.
Nel metodo chiede all'attuatore di avvisarlo nel caso in cui si verificasse un cambiamento, è quella che ha il metodo update che viene chiamato dalla classe dispositivo
Nel metodo update dato che il numero di sensori o attuatori non è fisso c'è una variabile intera che rappresenta i componenti attivi.

Refactoring: Extract Method

- 1) Nel refactoring, quando ho un frammento di codice separato lungo o non ben specificato cosa faccio?
- 2)Definisco quel pezzo di codice all'interno di un metodo, dandogli un nome significato.
- 3) Il nome deve fare riferimento a quello che il metodo fa all'interno del programma.

Refactoring: Extract Method

- 4) Applico il Pattern per semplificare un metodo lungo all'interno della classe ''Regola''.
- 5) Per rappresentare tutte le possibilità che ''Regola'' poteva assumere, si divide in sotto procedure, definite sempre nella stessa classe. Posso così chiamare questi metodi, che hanno le stesse funzioni di prima, ma riscritte con un numero minore di righe di codice.
- 6) Ottengo maggior comprensibilità del codice.

Refactoring: Extract Method


```
private void primoCampoTime(String operatore,String[] anticedenti){
    if (operatore.contains(">"))
        operatore = operatore.replace( target: ">",  replacement: "<");
    else if (operatore.contains("<"))
        operatore = operatore.replace( target: "<",  replacement: ">");
    secondoCampoTime(operatore, anticedenti);
}

private void secondoCampoTime(String operatore,String[] anticedenti){

    int ore = Integer.parseInt(anticedenti[1].split( regex: "[.]")[0]);
    int minuti = Integer.parseInt(anticedenti[1].split( regex: "[.]")[1]);
    condizione = new AntecedenteNumeroTime(ore,minuti,operatore);

}
```

Il controllo sulla regola che verifica se inizia con un ''IF'' viene messo in un metodo a parte, in cui potremo nel futuro aggiungere altri controlli, indipendenti dal tipo di regola.
Il controllo dell'operatore viene messo in un suo metodo
Il controllo del time nell'antecedente viene messo in un suo metodo.
Con il primo metodo sposto solo il time a secondo campo e gestisco i casi come uguali con un metodo comune

Refactoring: Extract Class

- 1) Quando ho una classe che però lavora con tanti dati, è possibile dividerla in più parti, per una gestione migliore?
- 2) Posso creare una nuova classe, così che si semplifica il codice di una, distribuendo metodi e attributi tra le due classi.
- 3) Posso così avere diverse classi che gestiscono poche responsabilità

Refactoring: Extract Class

- 4) Viene applicato nella classe Regola, dato che si occupa del parsing sia di antecedente che di conseguente che sono due parti in realtà logicamente diverse.
- 5) Si crea la classe ParserConseguente, definendo un metodo parsing nel quale inserire il metodo precedentemente appartenente a Regola,
- 6) Così si può avere un'istanza di ParserConseguente in Regola e si potrà usare per chiamare il metodo e far restituire l'oggetto Conseguente precedentemente creato.

Refactoring: Extract Class

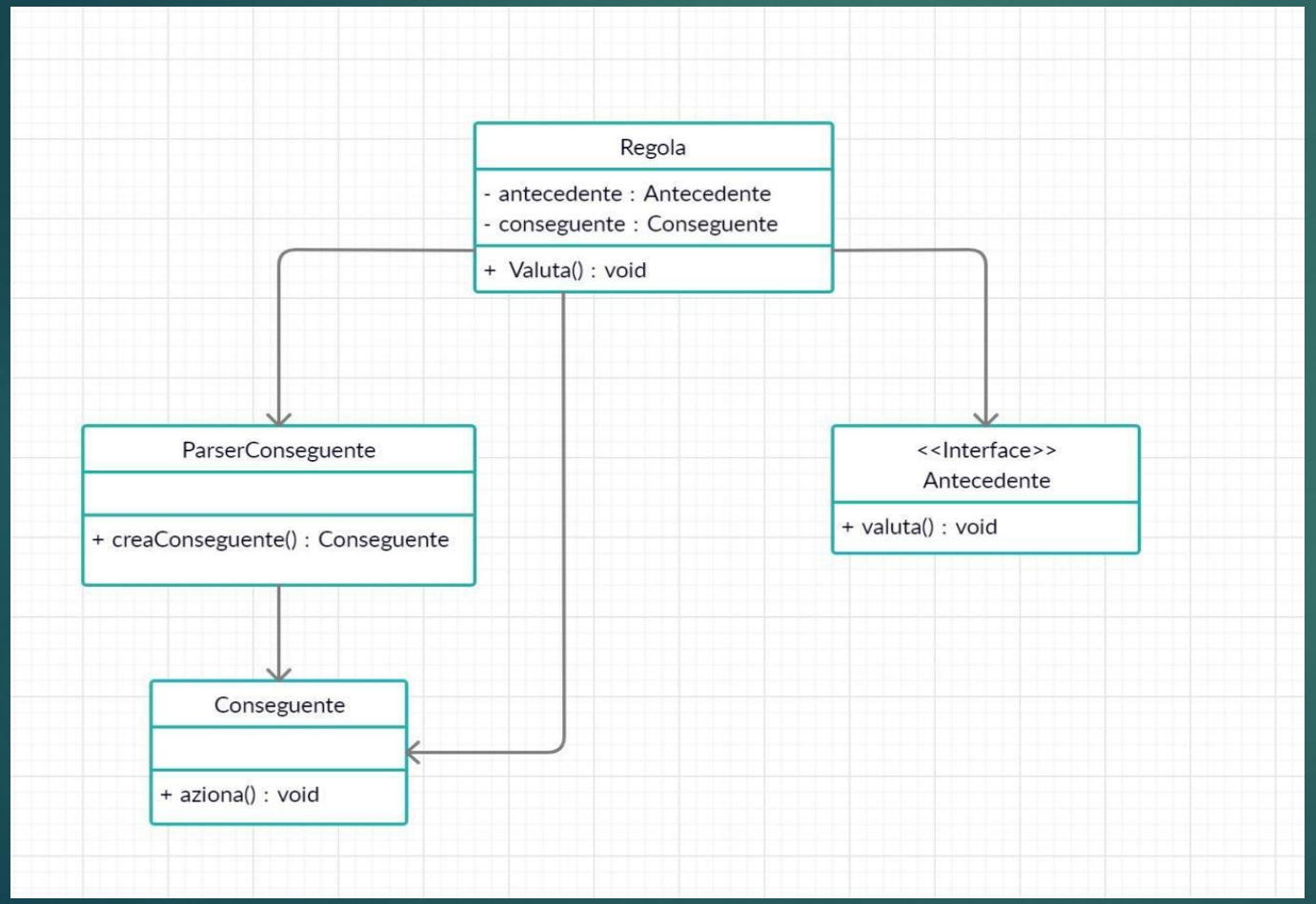


Diagramma della classi, che rappresenta la divisione di ''Regola'' nelle sue due parti

Testing JUnit

- 1) La verifica risponde alla domanda «Stiamo costruendo il software in modo corretto?», si pone come obbiettivo la scoperta di bug generati da input o sequenze di input.
- 2)Sono molto utili durante il refactoring per assicurarci che la modifica di un pezzo di codice non renda errato il comportamento di un'unita' di codice diversa per effetti collaterali.
- 3)I test dovrebbero essere compiuti in maniera automatica notificando gli sviluppatori in caso di in successo.
- JUnit e' un framework per test di unita' del linguaggio Java.

Testing JUnit

Come primo test abbiamo verificato i requisiti funzionali di UnitaImmobiliare, ovvero la gia' presenza di una stanza deve impedire di poterne creare una uguale e in maniera simile la gia' presenza di un artefatto in una stanza deve impedirne la creazione di uno identico.
Inoltre seguendo le linee guida del boundary value analysis/stress testing sono stati scelti input di lunghezza ampiamente maggiore di quella standard media e caratteri appartenenti a plane Unicode non normalmente usati dagli utenti del sistema

Classi esterne coinvolte sono : ControllerUnitaImmobiliare, SistemaDomotico

```
@Test
void aggiungiStanzaCode() {
    UnitaImmobiliare im = new UnitaImmobiliare( nome: "casa",  tipo: "appartamento",
            new ControllerUnitaImmobiliare(new SistemaDomotico()));
    //Controllo sia che funzioni inserendo un nome molto lungo sia
    //che la stanza prima non esista e dopo venga riconosciuta
    assertTrue(im.aggiungiStanzaCode("ProvaProvaProvaProvaProvaProva" +
            "ProvaProvaProvaProvaProvaProvaProva"));
    //Ora ci dovrebbe gia' essere
    assertFalse(im.aggiungiStanzaCode("ProvaProvaProvaProvaProva" +
            "ProvaProvaProvaProvaProvaProvaProvaProva"));
    //Testo ora gli artefatti
    assertTrue(im.aggiungiArtefattoCode( nomeStanza: "ProvaProvaProvaProva" +
                    "ProvaProvaProvaProvaProvaProvaProvaProvaProva",
            nomeArtefatto: "𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀"));
    //Ora ci dovrebbe gia' essere
    assertFalse(im.aggiungiArtefattoCode( nomeStanza: "ProvaProvaProvaProva" +
                    "ProvaProvaProvaProvaProvaProvaProvaProvaProva",
            nomeArtefatto: "𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀𐐀"));
}
```

Testing JUnit

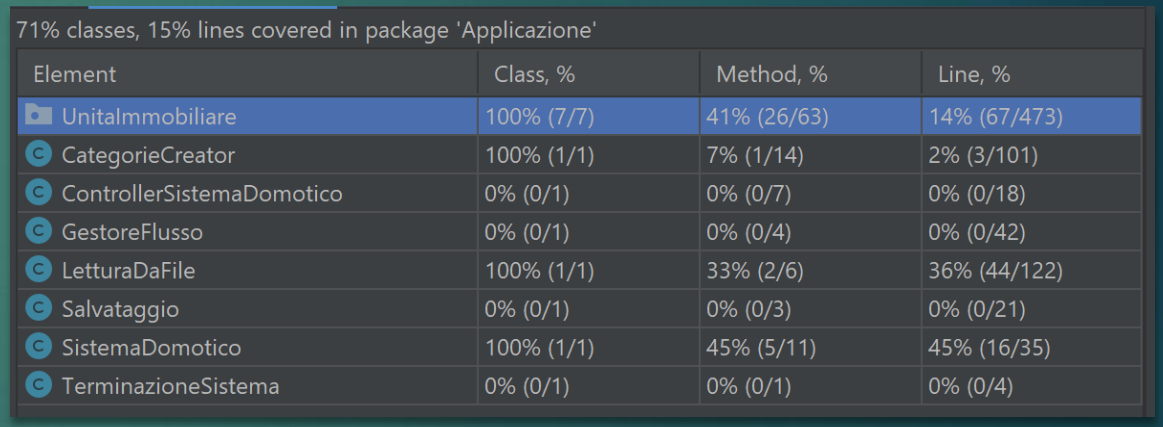
Abbiamo con questo test provato a verificare completamente il flusso di esecuzione di un metodo del nostro programma (testing white box).
Dopo aver chiamato l'esecuzione di importaUnitaImmobilireRiga e avergli passato la stringa da trasformare in oggetti del nostro sistema abbiamo controllato che ogni componente fosse stato correttamente aggiunto e che anche le classi esterne coinvolte nell'operazione avessero risposto correttamente alle invocazioni effettuate dal metodo testato

Classi esterne coinvolte sono : SistemaDomotica, UnitaImmobiliare,ControllerUnitaImmobiliare

```
@Test
void importaUnitaImmobiliariRiga() {
    SistemaDomotico sm = new SistemaDomotico();
    sm.aggiungiCategoriaSensore(new CategoriaSensore( nome: "luce"));
    sm.aggiungiCategoriaSensore(new CategoriaSensore( nome: "clima"));
    sm.aggiungiCategoriaAttuatore(new CategoriaAttuatore( nome: "tapparella"));
    UnitaImmobiliare im = new LetturaDaFile(sm,new CategorieCreator(sm)).
            importaUnitaImmobiliariRiga("" +
            "smeraldo-condominio-cucina-luce-0-tapparella-0-piedistallo-clima-0- -0");
    ControllerUnitaImmobiliare controller = im.getController();

    //Controllo che abbia creato la categoria luce,tapparella,clima nella stanza cucina
    //e nell'artefatto piedistallo in cucina
    assertFalse(controller.aggiungiCategoriaSensoreAStanza
            ( stanza: "cucina", catSensore: "luce"));
    assertFalse(controller.aggiungiCategoriaAttuatoreAStanza
            ( stanza: "cucina", catAttuatore: "tapparella"));
    assertTrue(controller.aggiungiCategoriaSensoreAArtefatto
            ( stanza: "cucina", artefatto: "piedistallo", catSensore: "clima"));


    //Controllo stanze e artefatti
    assertFalse(controller.aggiungiStanzaCode( s: "cucina"));
    assertFalse(controller.aggiungiArtefattoCode( nomeStanza: "cucina", nomeArtefatto: "piedistallo"));

    //Controllo nome e tipo immobile
    assertTrue(controller.getNomeUnitaImmobiliare().equals("smeraldo"));
    assertTrue(controller.getTipoUnitaImmobiliare().equals("condominio"));
}
```

Testing JUnit

Riportiamo per completezza le metriche di coverage generate dall'IDE riguardo il testing LetturaDaFile.
Fa inoltre vedere quali classi sono coinvolte durante il testing.



71% classes, 15% lines covered in package 'Applicazione'

Element	Class, %	Method, %	Line, %
UnitaImmobiliare	100% (7/7)	41% (26/63)	14% (67/473)
CategorieCreator	100% (1/1)	7% (1/14)	2% (3/101)
ControllerSistemaDomotico	0% (0/1)	0% (0/7)	0% (0/18)
GestoreFlusso	0% (0/1)	0% (0/4)	0% (0/42)
LetturaDaFile	100% (1/1)	33% (2/6)	36% (44/122)
Salvataggio	0% (0/1)	0% (0/3)	0% (0/21)
SistemaDomotico	100% (1/1)	45% (5/11)	45% (16/35)
TerminazioneSistema	0% (0/1)	0% (0/1)	0% (0/4)