UNIVERSITÀ
DEGLI STUDI
FIRENZE

Università degli Studi di Firenze

Dipartimento di Ingegneria dell'informazione

Sistema di gestione di sensori eterogenei

Relazione per il corso di Ingegneria del software

Studente:
Andrei Florea

Matricola:
7024180

Corso:
Ingegneria del software

Docente corso:
Prof. Enrico Vicario

Anno accademico 2024/2025

aprile 2025

Indice

Elenco delle figure

Elenco dei listati

1 Analisi

Il presente elaborato descrive la progettazione e l'implementazione di un sistema software in Java per il monitoraggio di sensori ambientali eterogenei, utilizzando i *design pattern Adapter*, *Composite* e *Observer*. L'obiettivo principale è integrare tre tipi di sensori con interfacce diverse in un sistema unificato che permetta di gestire le misurazioni in modo coerente e notificare una centralina di controllo al variare dei valori rilevati.

1.1 Contesto

Si hanno delle classi (A, B, C) che realizzano la gestione di sensori con interfacce diverse (`getMeasure()`, `misura()`, `measure()`). Si vogliono portare in uno schema Composite in cui si possano ottenere tutte le misure. Non è possibile modificare A, B, C, e quindi devono essere adattate con un Adapter. Si vuole poi che ogni sensore possa notificare a una centralina le sue variazioni e che lo faccia usando lo schema Observer. Infine, si vogliono scrivere dei test JUnit che verifichino i meccanismi caratterizzati.

1.2 Architettura

Il programma è sviluppato in Java. Per mantenere una separazione delle responsabilità, la struttura del progetto è stata divisa in quattro *package*: `domain`, `data`, `observer` e `sensors`. Essi si occupano, rispettivamente, di gestire la struttura uniforme e l'adattamento dei componenti, la rappresentazione dei dati misurati, la logica di osservazione e reazione, e la simulazione dei sensori. La struttura d'insieme è visibile nella figura 2.

Per utilizzare il programma è stata creata un'interfaccia a riga di comando, che permette d'interagire col sistema in modo semplice e intuitivo.

2 Progettazione

2.1 Requisiti funzionali

Sono stati definiti i seguenti requisiti per guidare lo sviluppo del sistema.

- **RF1 (Acquisizione dati).** Il sistema deve poter ottenere il valore misurato da ciascun tipo di sensore disponibile (tipo A, B, C), indipendentemente dalla sua interfaccia specifica.
- **RF2 (Adattamento obbligatorio).** Le classi originali dei sensori (A, B, C) sono considerate componenti esterni non modificabili. Il sistema deve adattarle a un'interfaccia interna comune senza alterare le classi originali.
- **RF3 (Aggregazione sensori).** Deve essere possibile creare gruppi di sensori. Il sistema deve permettere di richiedere tutte le misurazioni dei sensori appartenenti a un gruppo (o all'intera struttura) tramite un'unica operazione sull'elemento radice del gruppo.
- **RF4 (Notifica centralina).** Il sistema deve includere una centralina che viene notificata automaticamente quando si forza l'aggiornamento dei sensori (a seguito di una richiesta di notifica). La notifica deve contenere informazioni identificative del sensore e il valore misurato.
- **RF5 (Collaudo unitario).** Devono essere implementati test unitarî (JUnit) per verificare il corretto funzionamento.

2.2 Casi d'uso

I casi d'uso descrivono le interazioni principali tra l'utente e il sistema. L'applicativo ha un solo attore, l'utente, che s'interfaccia con la centralina. L'utente ha la possibilità di visualizzare le misure arrivate alla centralina, di richiedere l'aggiornamento di un sensore specifico o di richiedere l'aggiornamento di tutti i sensori. Gli schemi qui riportati rappresentano i casi d'uso.

2.3 Diagramma dei casi d'uso

- **Utente:** Persona che interagisce con il sistema tramite l'interfaccia a riga di comando per monitorare i sensori.

Nota: La centralina, in questo modello, agisce più come un componente interno (un osservatore) che come un attore esterno che inizia azioni.

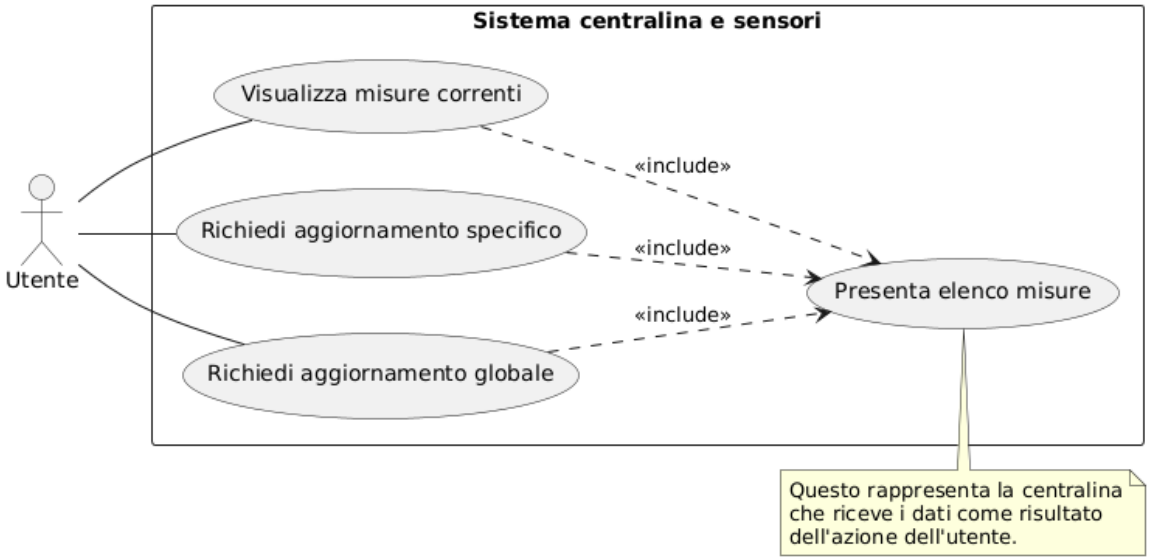

Figura 1: Diagramma dei casi d'uso.

2.4 Modelli dei casi d'uso

Caso d'uso 1	**Visualizza misure correnti**
Descrizione	L'utente richiede di vedere l'ultimo stato noto delle misure dei sensori gestiti dal sistema.
Livello	Obiettivo utente
Attore	Utente
Flusso base	1. L'utente seleziona l'opzione per visualizzare le misure correnti. 2. Il sistema presenta all'utente l'elenco delle ultime misure disponibili per ciascun sensore conosciuto.
Flusso alternativo	Se il sistema non dispone ancora di alcuna misura, informa l'utente che non ci sono dati disponibili da visualizzare.

Caso d'uso 2	**Richiedi aggiornamento specifico**
Descrizione	L'utente richiede al sistema di ottenere una nuova misura da un sensore specifico e di visualizzare lo stato aggiornato.
Livello	Obiettivo utente
Attore	Utente
Flusso base	1. L'utente seleziona l'opzione per aggiornare un sensore specifico. 2. Il sistema chiede all'utente di specificare quale sensore aggiornare. 3. L'utente fornisce l'identificativo del sensore. 4. Il sistema richiede e ottiene una nuova lettura dal sensore specificato. 5. Il sistema aggiorna l'ultima misura nota per quel sensore. 6. Il sistema presenta all'utente l'elenco aggiornato delle ultime misure disponibili per tutti i sensori.
Flusso alternativo	Se l'identificativo fornito dall'utente non corrisponde a un sensore gestito dal sistema, il sistema informa l'utente dell'errore.

Caso d'uso 3	**Richiedi aggiornamento globale**
Descrizione	L'utente richiede al sistema di ottenere una nuova misura da tutti i sensori gestiti e di visualizzare lo stato complessivo aggiornato.
Livello	Obiettivo utente
Attore	Utente
Flusso base	1. L'utente seleziona l'opzione per aggiornare tutti i sensori. 2. Il sistema richiede e ottiene una nuova lettura da ciascun sensore gestito. 3. Il sistema aggiorna l'ultima misura nota per ogni sensore. 4. Il sistema presenta all'utente l'elenco aggiornato delle ultime misure disponibili per tutti i sensori.

2.5 Struttura delle classi

2.6 Diagramma delle classi

Il diagramma dei *package* e delle classi implementate è stato realizzato con StarUML. Di seguito una descrizione concisa delle classi, delle dipendenze e delle loro responsabilità.

- `domain`. Definisce e implementa la struttura gerarchica dei componenti sensore tramite il pattern Composite (`Component`, `Composite`) e adatta i sensori specifici tramite il pattern Adapter (`AdapterA/B/C`), rendendoli anche osservabili (pattern Observer - Subject). **Dipendenze:** Dipende da `sensors` (per adattare), `data` (per creare `Misurazione`), e `java.util` (per estendere `Observable` e usare `List`). Gli Adapter fungono da foglie del Composite e da soggetti osservabili.

- **`data`**. Contiene le classi che rappresentano la struttura dei dati scambiati, principalmente il *data transfer object* `Misurazione`. Fornisce un formato dati standardizzato, disaccoppiando la rappresentazione dei dati dai componenti che li producono (`domain`) e li consumano (`observer`).
- **`observer`**. Contiene le implementazioni concrete degli osservatori che reagiscono alle notifiche. Dipende da `data` (per usare `Misurazione`) e `java.util` (per implementare `Observer` e usare `List`, `Observable`).
  È stato Mantenuto separato da `domain` per rispettare la separazione delle responsabilità tra *Subject* e *Observer* e per favorire l'estensibilità futura con altri tipi di *Observer*.
- **`sensors`**. Contiene le classi (`A`, `B`, `C`) che simulano i sensori fisici originali da adattare, ciascuna con il proprio metodo specifico. Isola la logica di simulazione dei dati grezzi. La separazione garantisce che eventuali modifiche future.

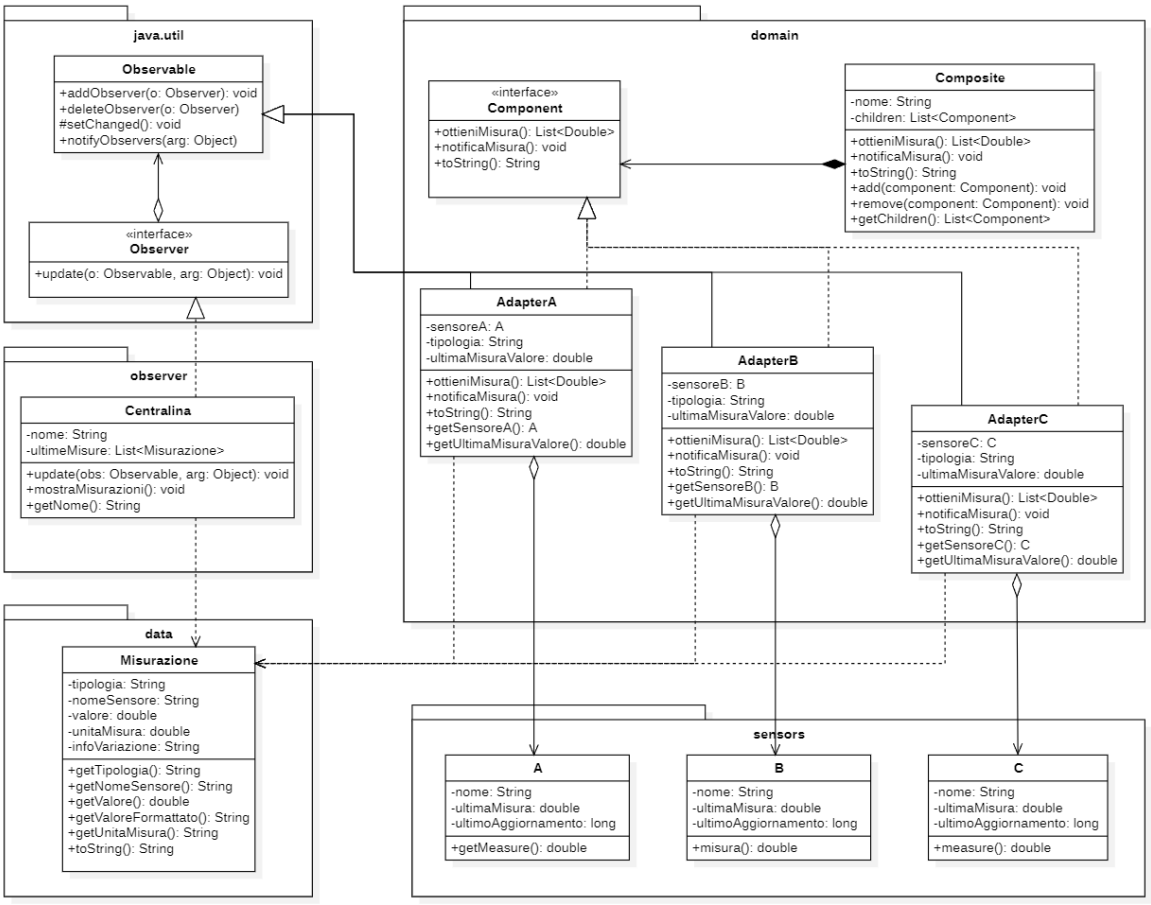

Figura 2: Diagramma delle classi.

2.7 Diagramma di sequenza: ottenere tutte le misure

Illustra come la richiesta di misure viene gestita dalla struttura Composite.

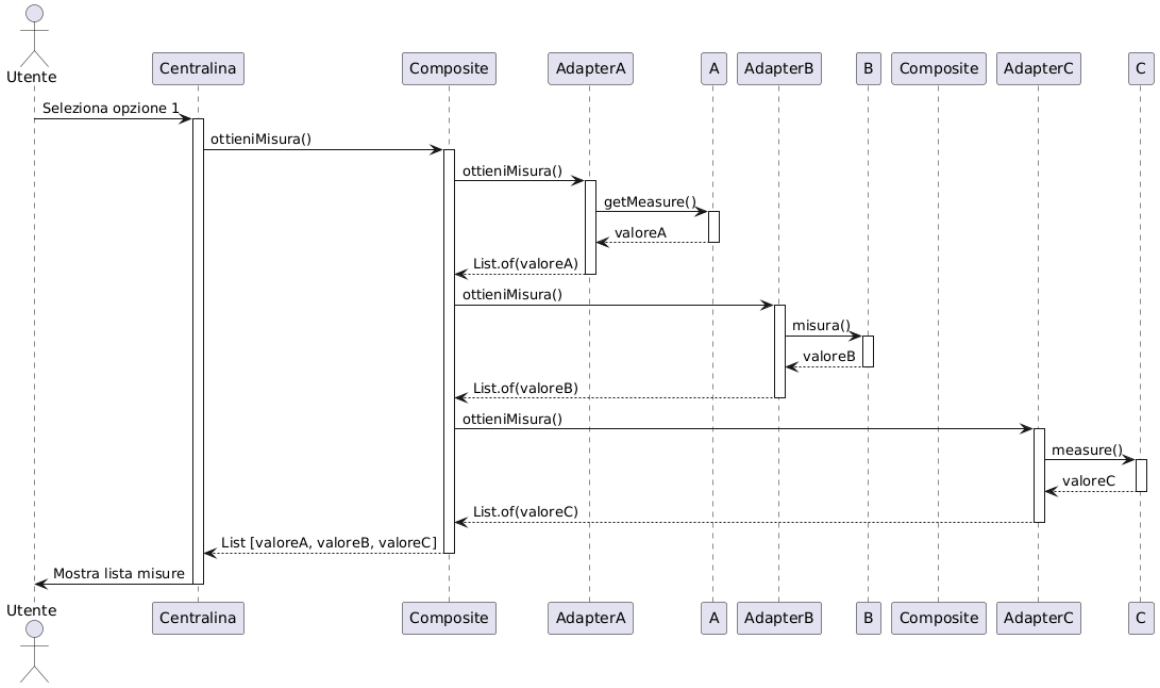

Figura 3: Diagramma di sequenza per ottenere tutte le misure.

2.8 Diagramma di attività: flusso interattivo

Descrive il flusso logico dell'interfaccia utente a riga di comando.

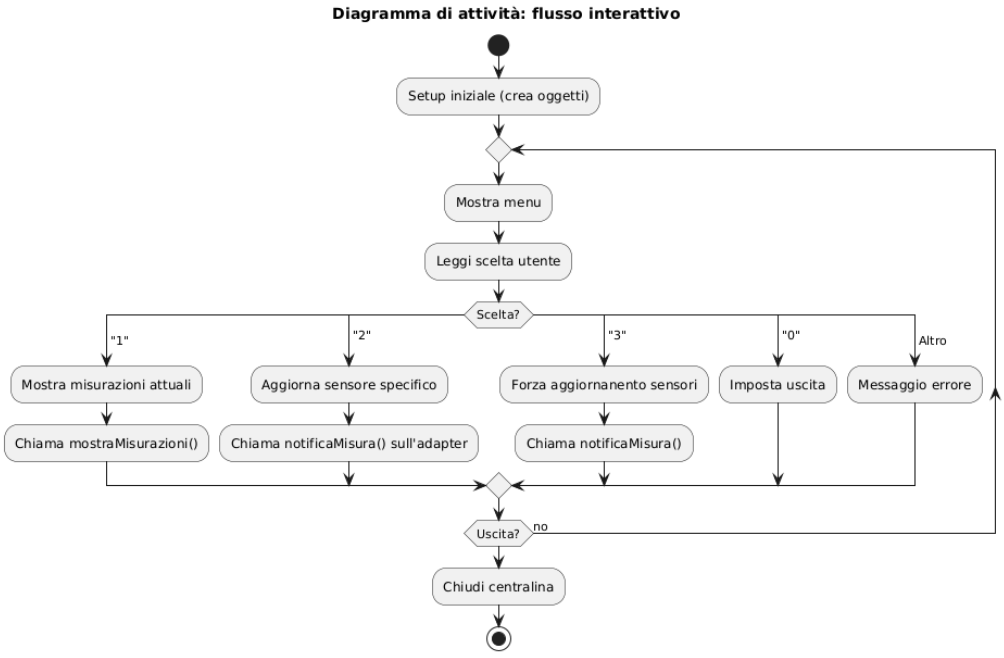

Figura 4: Diagramma di attività del flusso interattivo principale.

3 Implementazione

L'applicativo è stato realizzato in Java, utilizzando come ambiente di programmazione IntelliJ IDEA. Il repo è visibile qui. L'architettura del sistema si basa sull'applicazione combinata di tre *pattern* di progettazione.

3.1 Simulazione sensori

Il `sensors` ha la responsabilità esclusiva di contenere le classi che simulano i sensori fisici originali (`A`, `B`, `C`).

Questo *package* è mantenuto isolato e non ha dipendenze verso gli altri moduli dell'applicazione (`domain`, `data`, `observer`), garantendo che la logica di simulazione sia indipendente dal resto del sistema.

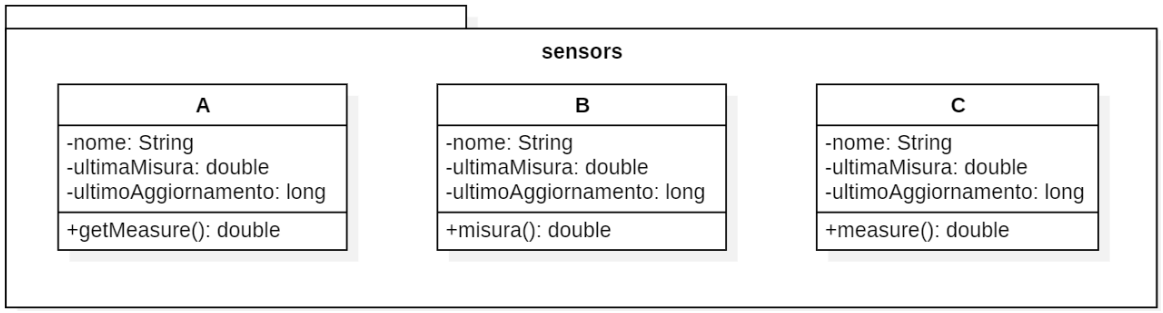

Figura 5: Diagramma delle classi del *package* `sensors`.

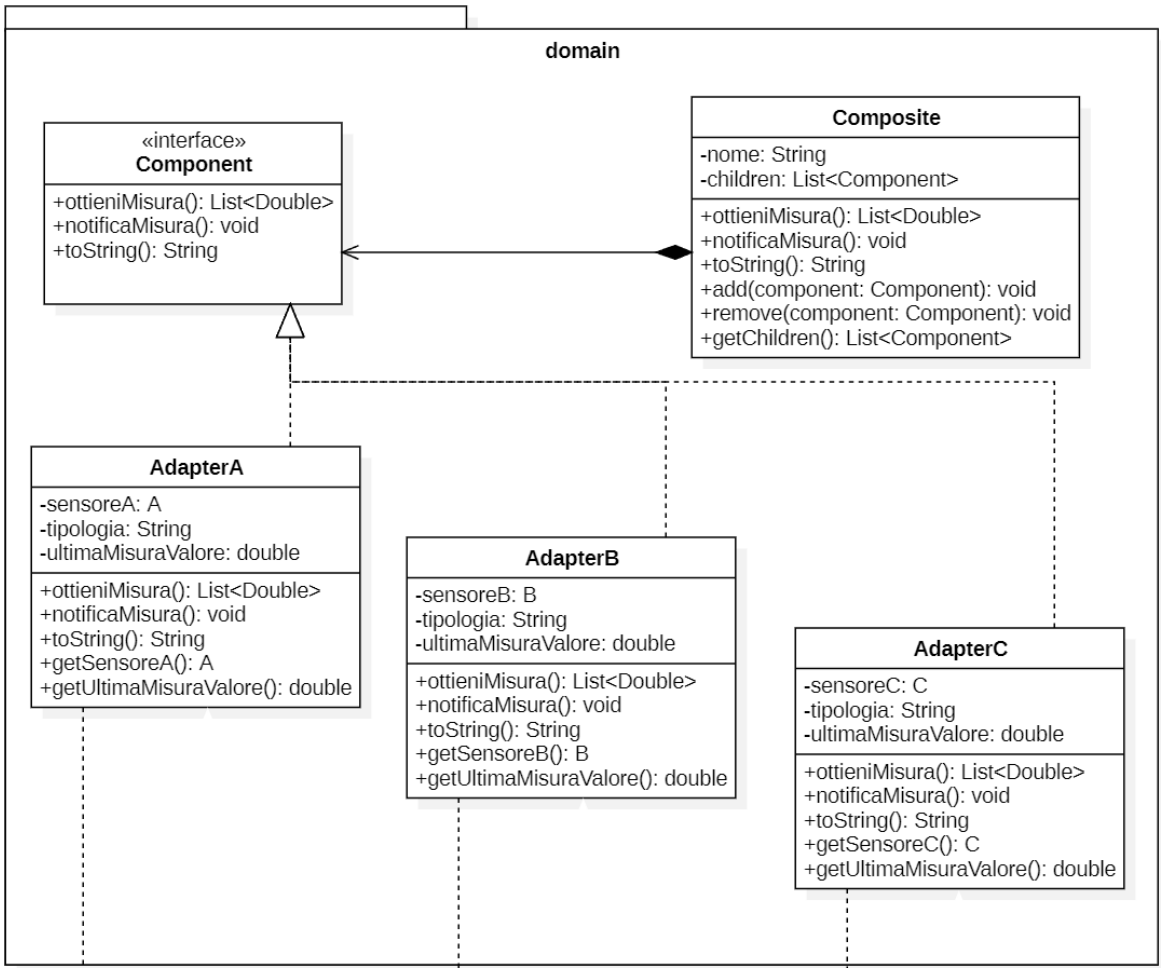

Figura 6: Particolare del *package* `domain` con le sue classi.

3.2 Adapter

3.2.1 Motivazione

La necessità di interagire con classi di sensori preesistenti (`A`, `B`, `C` nel *package* `sensors`), ciascuna dotata di un'interfaccia proprietaria e non modificabile per la lettura dei dati (`getMeasure()`, `misura()`, `measure()`), impone l'uso del *pattern* Adapter. Questo permette di creare un ponte tra queste interfacce eterogenee e un'interfaccia comune (`Component`), consentendo una gestione uniforme dei diversi tipi di sensore.

3.2.2 Implementazione

Per ciascun tipo di sensore originale (A, B, C), è stata creata una classe Adapter specifica (`AdapterA`, `AdapterB`, `AdapterC`).

- Ogni classe Adapter contiene un riferimento all'istanza del sensore originale che deve adattare (composizione).
- Ogni classe Adapter implementa l'interfaccia comune `Component` (definita per il *pattern* Composite, vedi sezione successiva), che richiede il metodo `List<Double> ottieniMisura()`.
- Ogni classe Adapter estende la classe `java.util.Observable` per poter fungere da soggetto nel *pattern* Observer (si veda la sezione successiva).

Il metodo `ottieniMisura()` dell'adapter delega la chiamata al metodo specifico del sensore contenuto e restituisce il singolo risultato `double` in una `List<Double>` (usando `Collections.singletonList`) per conformarsi all'interfaccia `Component`. Il metodo `notificaMisura()` ottiene il valore corrente dal sensore, crea un oggetto `Misurazione` e usa i metodi ereditati da `Observable` (`setChanged()`, `notifyObservers()`) per notificare gli observer registrati.

In sintesi, gli Adapter nel *package* `domain` non solo risolvono il problema delle interfacce eterogenee dei sensori, ma fungono anche da elementi foglia nella struttura Composite e da sorgenti di notifica per il pattern Observer, collegando così diversi aspetti chiave dell'architettura.

```java
package domain;

import data.Misurazione;
import sensors.A;
import java.util.Collections;
import java.util.List;
import java.util.Observable;
// ... altri import

@SuppressWarnings("deprecation")
// Estende Observable e implementa Component
public class AdapterA extends Observable implements Component {
    private final A sensoreA;
    private final String tipologia = "temperatura";
    private Double ultimaMisuraValore = null;
    // ... costruttore ...

    @Override
    public List<Double> ottieniMisura() { // Metodo interfaccia Component
        double misura = sensoreA.getMeasure(); // Delega ad A
        ultimaMisuraValore = misura;
        return Collections.singletonList(misura); // Adatta ritorno a List<Double>
    }

    @Override
    public void notificaMisura() {
        double misuraCorrente = sensoreA.getMeasure(); // Nuova lettura
        // ... (calcolo variazione omesso)...
        // Crea DTO dal package data
        Misurazione misurazione = new Misurazione(/*...*/);
```

```
31          ultimaMisuraValore = misuraCorrente;
32
33          // Logica Observer ereditata
34          setChanged();
35          notifyObservers(misurazione); // Notifica con Misurazione
36      }
37  }
```

Listato 1: Estratto da AdapterA nel *package* `domain`.

3.3 Composite

3.3.1 Motivazione

Il requisito funzionale 3 richiede di poter organizzare i sensori (o meglio, i loro adattatori) in gruppi e di poter trattare un singolo sensore e un gruppo di sensori nello stesso modo, in particolare per ottenere tutte le misurazioni con un'unica chiamata. Il *pattern* Composite permette di costruire strutture ad albero (gerarchie parte-tutto) in cui sia i nodi foglia (oggetti singoli) sia i nodi interni (composizioni di oggetti) implementano la stessa interfaccia.

3.3.2 Implementazione

È stata definita l'interfaccia `Component` che rappresenta l'astrazione comune per tutti gli elementi della struttura.

```
1  package domain;
2  import java.util.List;
3
4  public interface Component {
5      List<Double> ottieniMisura();
6      void notificaMisura();
7  }
```

Listato 2: Interfaccia Component.

Questa interfaccia è implementata da:

- Le classi foglia sono le classi `AdapterA`, `AdapterB`, `AdapterC`. Implementano `ottieniMisura()` restituendo la loro singola misura (in una lista) e `notificaMisura()` eseguendo la logica di notifica Observer.
- La classe `Composite` rappresenta un nodo interno che può contenere altri `Component` (sia foglie che altri compositi). Mantiene una lista (`List<Component> children`) dei suoi figli.

La classe `Composite` implementa i metodi dell'interfaccia `Component` delegando le operazioni ai figli:

- `ottieniMisura()` itera sui figli, chiama `ottieniMisura()` su ciascuno e aggrega tutte le liste di risultati in un'unica lista.
- `notificaMisura()` itera sui figli e chiama `notificaMisura()` su ciascuno.

Inoltre, fornisce metodi `add(Component)` e `remove(Component)` per gestire la collezione di figli.

```
1  package domain;
2  import java.util.ArrayList;
3  import java.util.List;
4
5  public class Composite implements Component {
6      private final List<Component> children = new ArrayList<>();
7      // ... costruttore e altri metodi ...
8
9      @Override
10     public List<Double> ottieniMisura() { // Delega e aggrega
11         List<Double> misureAggregate = new ArrayList<>();
```

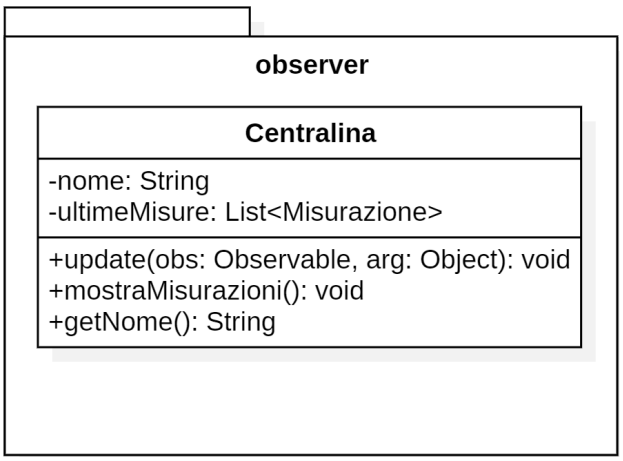

Figura 7: Particolare del *package* `observer` con la classe `Centralina`.

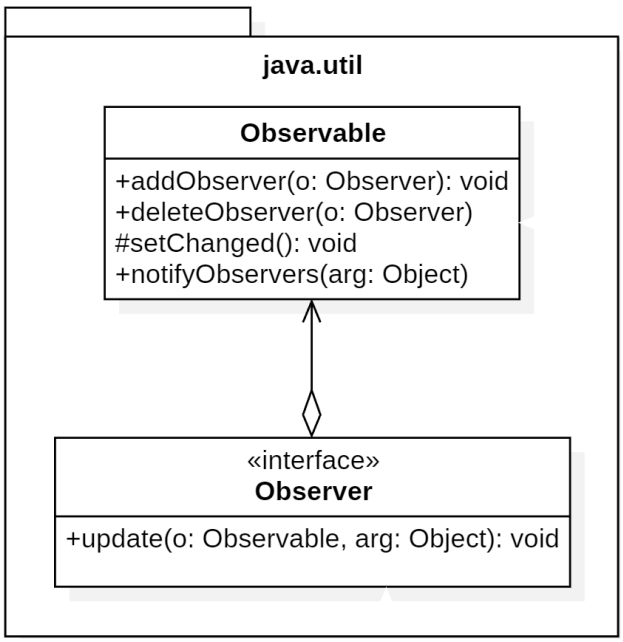

Figura 8: Particolare del *package* `java.util`.

```
12          for (Component child : children) {
13              misureAggregate.addAll(child.ottieniMisura());
14          }
15          return misureAggregate;
16      }
17
18      @Override
19      public void notificaMisura() { // Delega la richiesta di notifica
20          for (Component child : children) {
21              child.notificaMisura();
22          }
23      }
24  }
```

Listato 3: Estratto dalla classe Composite nel *package* `domain`.

La scelta di far restituire `List<Double>` da `ottieniMisura()` è fondamentale per l'uniformità: permette al cliente di chiamare lo stesso metodo su una foglia (ottenendo una lista con un elemento) o su un composito (ottenendo una lista con molti elementi) senza dover distinguere i due casi.

3.4 Observer

3.4.1 Motivazione

Il requisito funzionale 4 richiede che la centralina sia informata automaticamente dei nuovi valori misurati dai sensori. Il *package* Observer definisce una dipendenza uno-a-molti tra oggetti, in modo che quando un oggetto (il soggetto, o osservabile) cambia stato, tutti i suoi dipendenti (gli osservatori) vengano notificati e aggiornati automaticamente.

L'implementazione (figura 7) si basa sul meccanismo fornito da `java.util` (figura 8). I ruoli sono distribuiti tra i *package* come segue.

3.4.2 Implementazione

È stato utilizzato il meccanismo fornito da Java (classi `java.util.Observable` e interfaccia `java.util.Observer`). Le parti del *pattern* sono:

- **Osservabile (soggetto):** Le classi `AdapterA`, `AdapterB`, `AdapterC` estendono `Observable`. Questo fornisce loro i metodi per gestire una lista di observer (`addObserver()`, `deleteObserver()`)

e per notificarli (`setChanged()`, `notifyObservers()`). Come visto nel Listato 4, il metodo `notificaMisura()` dell'adapter si occupa di chiamare `setChanged()` e `notifyObservers(mis)`.

- **Osservatore**. La classe `Centralina` implementa l'interfaccia `Observer`, che richiede l'implementazione del metodo `update(Observable o, Object arg)`.

- La classe `Centralina` realizza l'interfaccia `java.util.Observer`, implementando quindi il metodo `update(Observable o, Object arg)`. Questo metodo rappresenta il punto di ingresso per le notifiche *push* inviate dagli `Adapter`.

- Esiste una dipendenza concettuale verso `java.util.Observable`, indicato come tipo del primo parametro di `update`, rappresentando la sorgente della notifica.

Quando un *adapter* chiama `notifyObservers(misurazione)`, il metodo `update()` di tutte le istanze di `Centralina` registrate su quell'*adapter* viene invocato automaticamente. Un oggetto della classe `Misurazione` viene passato come argomento.

```
package observer;
import data.Misurazione;
import java.util.Observable;
import java.util.Observer;

@SuppressWarnings("deprecation")
public class Centralina implements Observer {
    @Override
    public void update(Observable obs, Object arg) {
        if (arg instanceof Misurazione misurazione) {
            // Aggiorna o aggiungi la misurazione alla lista
            boolean aggiornata = false;
            for (int i = 0; i < ultimeMisure.size(); i++) {
                if (ultimeMisure.get(i).getNomeSensore().equals(misurazione.
    getNomeSensore())) {
                    ultimeMisure.set(i, misurazione);
                    aggiornata = true;
                    break;
                }
            }

            if (!aggiornata) {
                ultimeMisure.add(misurazione);
            }

        } else {
            System.out.println(nome + ": dato non riconosciuto da " + arg);
        }
    }
    // ... metodo per visualizzare le misure sul terminale
}
```

Listato 4: Estratto dalla classe Centralina nel *package* `observer`.

Questo disaccoppia gli *adapter* dalla centralina (che non conosce i dettagli interni degli *adapter* e riceve solo un oggetto `Misurazione`).

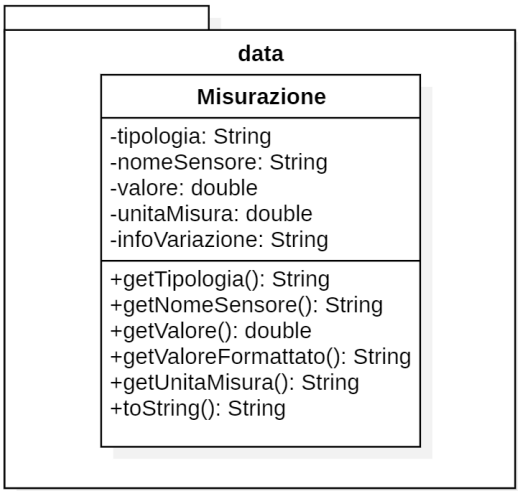

Figura 9: Particolare del *package* `data`.

3.5 *Data transfer object*

3.5.1 Motivazione

Per disaccoppiare i componenti che producono dati (gli Adapter nel *package* `domain`) da quelli che li consumano (la Centralina nel *package* `observer`), è stata introdotta una classe specifica per il trasporto delle informazioni relative a una singola misurazione. Questo approccio standardizza il formato dei dati scambiati e semplifica le interfacce.

3.5.2 Implementazione

Il *package* `data` contiene la classe `Misurazione`, progettata come un *data transfer object* immutabile (figura 9).

- **Attributi**. Incapsula tutte le informazioni rilevanti per una misurazione come campi privati e finali: `tipologia` (es. «temperatura»), `nomeSensore`, `valore` numerico, `unitaMisura` (es. «°C») e una stringa `variazione` (che può contenere informazioni aggiuntive come la variazione percentuale).
- **Formattazione:** Include un metodo `getValoreFormattato()` che utilizza un `DecimalFormat` statico per presentare il valore numerico con una precisione definita (una cifra decimale). Il metodo `toString()` sovrascritto fornisce una rappresentazione testuale completa e formattata dell'intera misurazione, utile per la visualizzazione.

Questa classe funge quindi da contenitore dati omogeneo, utilizzato principalmente come argomento nelle notifiche del *pattern* Observer.

```
1  package data;
2  import java.text.DecimalFormat;
3  public class Misurazione {
4      private final String tipologia;
5      private final String nomeSensore;
6      private final double valore;
7      private final String unitaMisura;
8      private final String variazione; // Contiene info aggiuntive (es. %)
9      private static final DecimalFormat formatter = new DecimalFormat("0.0");
10
11     // Costruttore per inizializzare l'oggetto immutabile
12     public Misurazione(String tipologia, String nomeSensore, double valore,
13                        String unitaMisura, String variazione) {
14         this.tipologia = tipologia;
15         this.nomeSensore = nomeSensore;
16         this.valore = valore;
```

```
        this.unitaMisura = unitaMisura;
        this.variazione = variazione;
    }

    // Metodi getter (esempio, altri omessi per brevit )
    public String getNomeSensore() { return nomeSensore; }
    public double getValore() { return valore; }

    // Metodo per formattazione specifica del valore
    public String getValoreFormattato() {
        return formatter.format(valore);
    }

    @Override
    public String toString() {
        return nomeSensore + " (" + tipologia + "): " + getValoreFormattato() +
                " " + unitaMisura + variazione;
    }
}
```

Listato 5: Estratto dalla classe Misurazione (DTO) nel *package* `data`.

4 Collaudo unitario

Sono stati implementati test unitarî utilizzando JUnit 5 per verificare la correttezza delle componenti chiave e l'implementazione dei *pattern*. È possibile trovare tutta la parte di codice relativa ai collaudi unitarî nella cartella `src/test`.

4.1 Test degli adapter

I test unitarî per queste classi (es. `AdapterATest`) sono strutturalmente simili e mirano a verificare:

- l'aderenza ai *pattern*: si controlla che l'*adapter* implementi l'interfaccia `Component` (per il *pattern* Composite) e che estenda la classe `Observable` (per il *pattern* Observer). Questo viene verificato tramite asserzioni `instanceof`;
- il recupero della misura (`ottieniMisura`), verificando che il metodo deleghi correttamente la chiamata al sensore (simulato tramite un *mock* per isolare il test), restituisca il valore ottenuto incapsulato nel formato richiesto dall'interfaccia `Componente` aggiorni lo stato interno dell'*adapter* (l'attributo `ultimaMisuraValore`);
- la notifica all'*observer* (`notificaMisura`), che si assicura che la chiamata a `notificaMisura` recuperi un nuovo valore dal sensore (*mock*), crei un oggetto `Misurazione` contenente i dati corretti e lo invii agli *observer* registrati. L'avvenuta notifica e la correttezza dei dati vengono verificate usando un `TestObserver` fittizio, come mostrato nell'estratto del listato 6. Viene controllato anche l'aggiornamento dello stato interno dell'*adapter*.

È stato utilizzato un oggetto simulato (*mock*) del sensore all'interno della classe di test per fornire valori prevedibili e rendere il collaudo deterministico. La soluzione è presentata nel listato seguente.

```
@Test
@DisplayName("notificaMisura: notifica observer con Misurazione...")
void testNotificaMisura() {
    ((MockSensoreA) sensore).setNextValue(21.5);
    TestObserver observer = new TestObserver();
    adapter.addObserver(observer);
    assertNull(adapter.getUltimaMisuraValore()); // precondizione

    adapter.notificaMisura();

    // Verifica che l'observer sia stato notificato correttamente
    assertEquals(1, observer.getUpdateCount(), "Notifica avvenuta?");
```

```
13      assertNotNull(observer.getLastArg(), "Argomento notifica presente?");
14      assertInstanceOf(Misurazione.class, observer.getLastArg(), "Tipo argomento
        corretto?");
15
16      // Verifica dati essenziali nella misurazione ricevuta
17      Misurazione notifica = (Misurazione) observer.getLastArg();
18      assertEquals("TestAdapterA", notifica.getNomeSensore());
19      assertEquals(21.5, notifica.getValore(), 0.001);
20      // ... (altre asserzioni sui dati della Misurazione omesse)
21      assertEquals(21.5, adapter.getUltimaMisuraValore(), 0.001);
22  }
```

Listato 6: Estratto da AdapterATest: verifica essenziale di notificaMisura

4.2 Test del Composite (`CompositeTest`)

I test per la classe `Composite` si concentrano sulla sua responsabilità principale nel *pattern* Composite: gestire una collezione di `Component` (figli) e trattare la collezione come un singolo `Component`. Si verifica quindi:

- la corretta gestione dei figli tramite i metodi `add` e `remove`;
- che il metodo `ottieniMisura` deleghi la chiamata a tutti i figli (foglie o altri Composite) e aggreghi correttamente i risultati in un'unica lista. Viene provato anche il comportamento con Composite vuoti e annidati;
- che il metodo `notificaMisura` propaghi efficacemente la chiamata a tutti i componenti figli, inclusi quelli contenuti in sotto-composite.

Questi test utilizzano oggetti *mock* (`TestComponent`) per simulare foglie con risposte predefinite.

4.3 Test dell'Observer concreto (`CentralinaTest`)

La classe `Centralina` agisce come osservatore principale nel sistema. I test verificano:

- la corretta ricezione delle notifiche tramite il metodo `update`, assicurandosi che processi solo oggetti di tipo `Misurazione` e ignori notifiche con dati non validi;
- la corretta memorizzazione e aggiornamento delle misurazioni ricevute (l'ultima misura per ogni sensore sovrascrive la precedente);
- la capacità di gestire misurazioni provenienti da sensori multipli contemporaneamente;
- che il metodo `mostraMisurazioni` rifletta accuratamente lo stato interno corrente della centralina (verificato tramite cattura dell'output su console).

Per simulare l'invio controllato delle notifiche, viene utilizzato un oggetto `TestObservable` simulato.

4.4 Test dei sensori (`SensoreATest`, ecc.)

I test per le classi base dei sensori (`A`, `B`, `C`) verificano:

- l'inizializzazione corretta (es. nome del sensore);
- che i metodi per ottenere la misura (es. `getMeasure`, `misura`) restituiscano valori numerici;
- principalmente, che la logica di *variazione* del valore tra letture successive rispetti le regole implementate (es. variazioni più ampie dopo lunghi periodi di inattività per il sensore A).

4.5 Test della classe dati (`MisurazioneTest`)

Infine, semplici test sulla classe `Misurazione` assicurano che questo *data transfer object* funzioni come atteso. Verificano:

- che i costruttori memorizzino correttamente i dati forniti;
- che i metodi per la formattazione dell'uscita (`getValoreFormattato`, `toString`) producano le stringhe nel formato desiderato.

Tutti i test eseguiti hanno avuto esito positivo, come mostrato in figura 10.

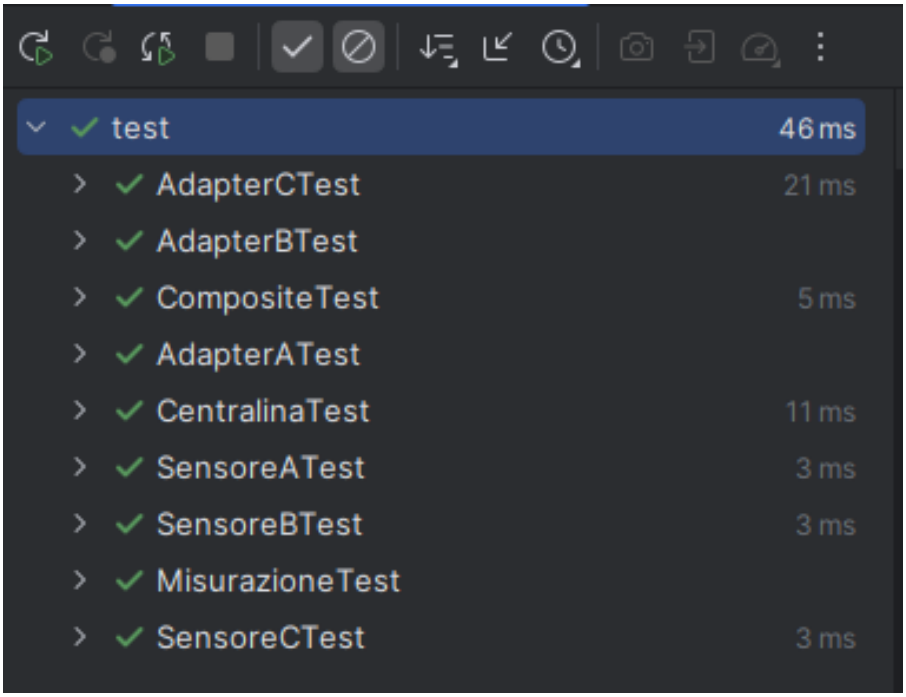

Figura 10: Esito positivo dell'esecuzione della serie di test (cattura schermo, dettaglio).