Università degli Studi di Firenze

Dipartimento di Ingegneria dell'informazione

Sistema di gestione di sensori eterogenei

Elaborato per il corso di Ingegneria del software

Anno accademico 2024/2025

aprile 2025

Indice

Elenco delle figure

Elenco dei listati

1 Introduzione

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 applicativo

Il sistema simula un contesto in cui diversi tipi di sensori (ad esempio per temperatura, umidità, pressione), potenzialmente provenienti da fornitori diversi e quindi con metodi di accesso ai dati non uniformi, devono essere gestiti da un unico programma centrale.

1.2 Obiettivi del progetto

Gli obiettivi specifici del progetto sono stati:

- Creare un meccanismo per leggere i dati da sensori con interfacce diverse (es. `getMeasure()`, `misura()`, `measure()`) attraverso un'unica interfaccia comune.
- Organizzare logicamente i sensori e poter richiedere le misurazioni da un intero gruppo con una singola operazione.
- Implementazione delle notifiche con una centralina che venga informata automaticamente quando un sensore rileva un nuovo valore.
- Assicurare la correttezza del sistema attraverso test JUnit che verifichino l'implementazione dei *pattern* e la logica applicativa.

2 Requisiti

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

2.1 Requisiti funzionali

- **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 Requisiti non funzionali

Anche se non implementati esplicitamente in dettaglio, si considerano importanti i seguenti aspetti:

- **Affidabilità.** In un sistema reale, la gestione degli errori (es. sensore non raggiungibile) sarebbe cruciale. La centralina dovrebbe poter continuare a funzionare con i sensori disponibili.
- **Manutenibilità.** L'uso dei *pattern* di progettazione mira a rendere il codice più facile da capire, modificare ed estendere (es. aggiungere nuovi tipi di sensori).
- **Interfaccia.** Deve essere presente un'interfaccia utente minimale (a riga di comando) che permetta di richiedere le misurazioni.

3 Casi d'uso

I casi d'uso descrivono le interazioni principali tra l'utente e il sistema.

3.1 Attori

- **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.

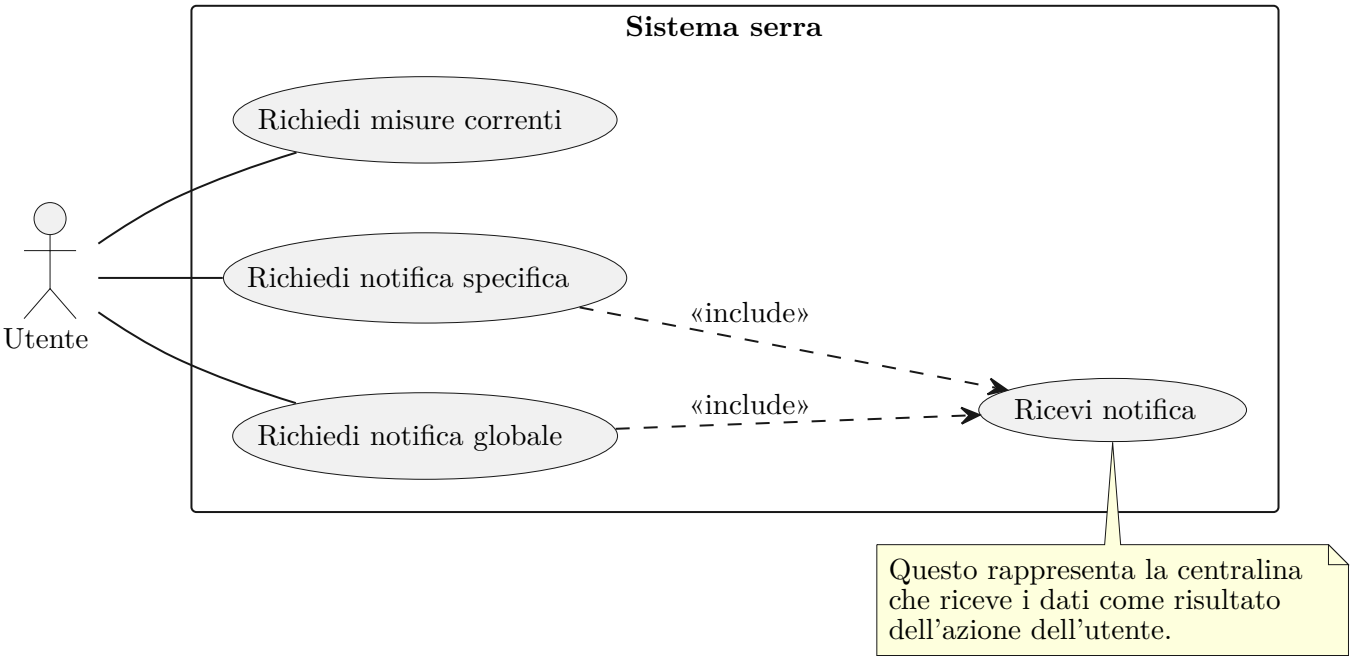

Figura 1: Diagramma dei casi d'uso.

3.2 Descrizione dei casi d'uso

Tabella 1: Caso d'uso: Richiedi misure correnti (stato attuale)

Nome	Richiedi misure correnti
Attore principale	Utente
Precondizioni	Il sistema è avviato e i sensori sono configurati nella struttura e c'è già una misura disponibile.
Flusso principale	1. L'utente seleziona l'opzione per ottenere tutte le misure correnti. 2. Il sistema invoca il metodo `mostraMisurazioni()` sull'elemento Centralina. 3. Il sistema stampa la lista di tutte e sole le misure correnti salvate nella centralina.
Postcondizioni	L'utente visualizza i valori correnti di tutti i sensori.

Tabella 2: Caso d'uso: Richiedi notifica specifica/globale

Nome	Richiedi notifica specifica/globale
Attore principale	Utente
Precondizioni	Il sistema è avviato, la centralina è registrata come *observer* sugli *adapter*.
Flusso principale	1. L'utente seleziona l'opzione per richiedere una notifica (da un sensore specifico o da tutti). 2. Se *specifica*: l'utente indica quale sensore (A, B, o C). Il sistema invoca `notificaMisura()` sull'*adapter* corrispondente. 3. Se *globale*: il sistema invoca `notificaMisura()` sull'elemento radice (`Composite`). 4. L'*adapter* (o gli *adapter*, nel caso globale) ottiene il valore corrente dal sensore associato. 5. L'*adapter* crea un oggetto `Misurazione` con i dettagli. 6. L'*adapter* notifica i suoi observer (la centralina) passando l'oggetto `Misurazione`. 7. La centralina riceve la notifica (tramite il suo metodo `update()`) e visualizza le informazioni ricevute.
Postcondizioni	L'utente visualizza sul terminale la notifica.

4 Diagrammi di progettazione

Questa sezione illustra la struttura e le interazioni del sistema attraverso diagrammi UML.

4.1 Diagramma delle classi

Il diagramma mostra le classi, le interfacce e le loro relazioni.

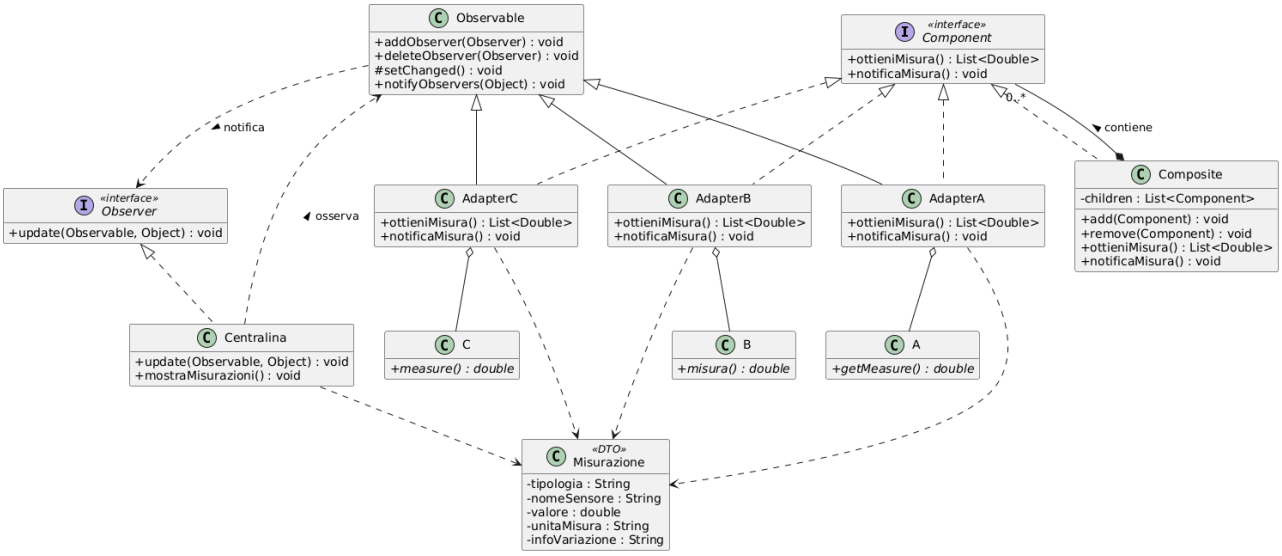

Figura 2: Diagramma delle classi.

4.2 Diagramma di sequenza: ottenere tutte le misure

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

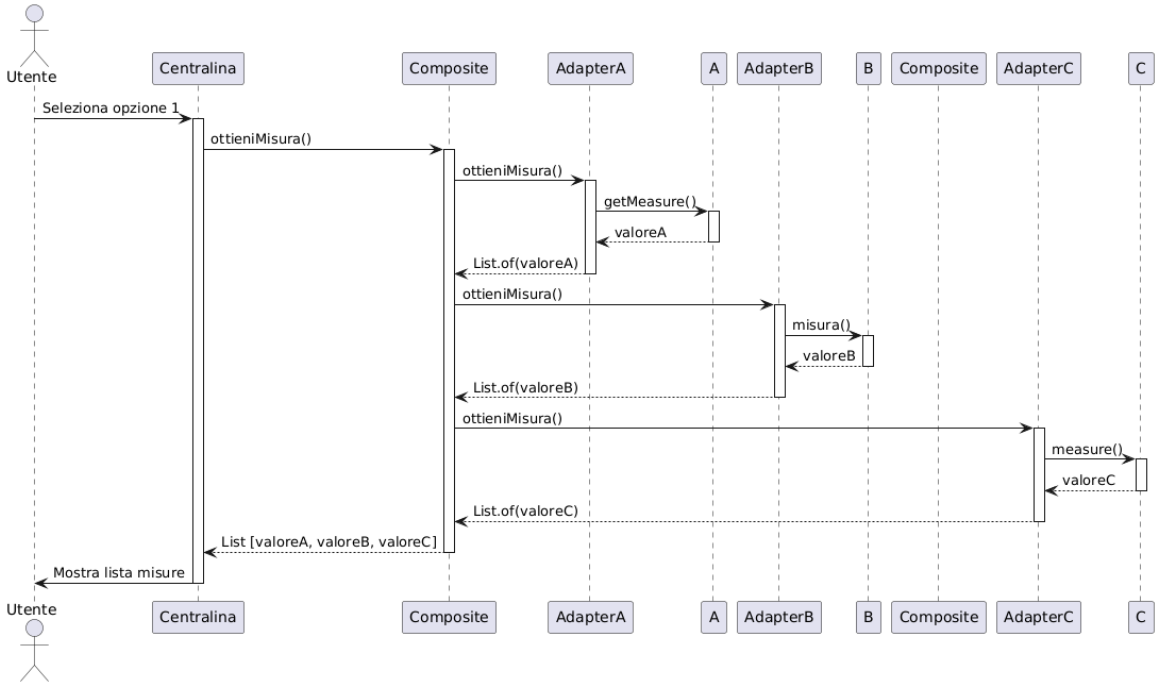

Figura 3: Diagramma di sequenza per ottenere tutte le misure.

4.3 Diagramma di attività: flusso interattivo

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

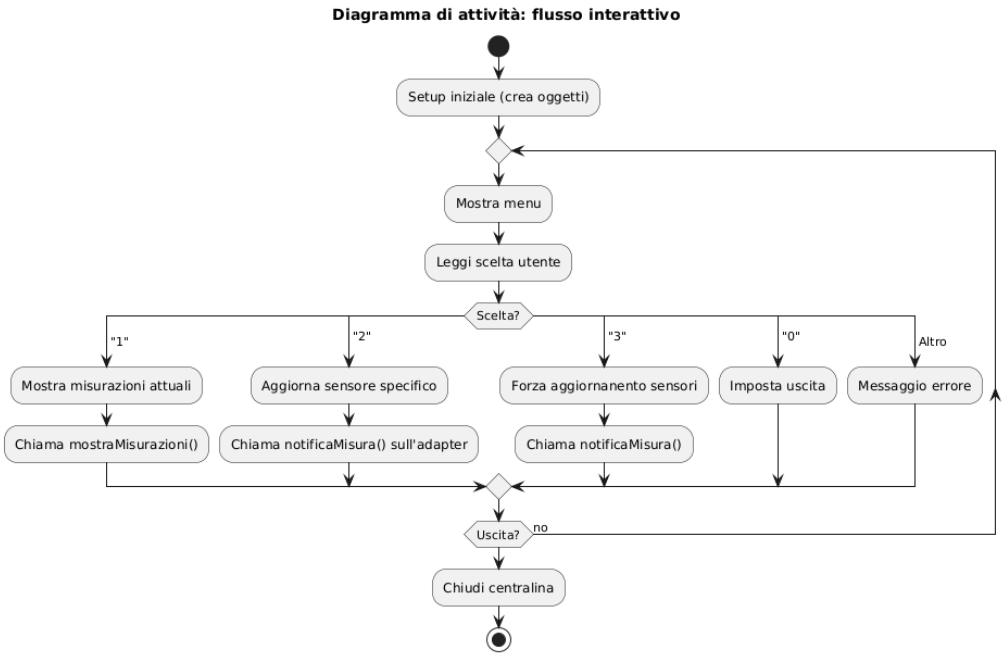

Figura 4: Diagramma di attività del flusso interattivo principale

5 Implementazione

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

5.1 Adapter

5.1.1 Motivazione

Il requisito funzionale 2 impone di lavorare con classi di sensori preesistenti (A, B, C) che non possono essere modificate e che possiedono interfacce diverse per ottenere la misurazione (`getMeasure()`, `misura()`, `measure()`). Per poter interagire con questi sensori in modo uniforme, è necessario adattare le loro interfacce specifiche a un'interfaccia comune definita all'interno del nostro sistema. Il *pattern* Adapter risponde esattamente a questa esigenza, permettendo a classi con interfacce incompatibili di collaborare.

5.1.2 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.

```java
package adapter;

import composite.Component;
import model.Misurazione;
import sensori.A;
import java.util.Collections;
import java.util.List;
import java.util.Observable;
// ... altri import

@SuppressWarnings("deprecation")
public class AdapterA extends Observable implements Component {
    private final A sensoreA;
    private final String tipologia = "temperatura";
    private Double ultimaMisuraValore = null;
    private static final DecimalFormat df = new DecimalFormat("0.0");
    // ... costruttore ...

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

    @Override
    public void notificaMisura() {
        double misuraCorrente = sensoreA.getMeasure();
        // ... (calcolo variazione omesso) ...
        Misurazione misurazione = new Misurazione(
                tipologia,
                sensoreA.getNome(),
                misuraCorrente,
                unitaMisura,
                variazioneInfo
        );
        ultimaMisuraValore = misuraCorrente;

        // Logica Observer
        setChanged();
        notifyObservers(misurazione); // Notifica con l'oggetto Misurazione
    }
}
```

Listato 1: Estratto da AdapterA

5.2 Composite

5.2.1 Motivazione

Il requisito funzionale 3 richiede di poter organizzare i sensori (o meglio, i loro adapter) 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.

5.2.2 Implementazione

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

```
package composite;
import java.util.List;

public interface Component {
    List<Double> ottieniMisura();
    void notificaMisura();
}
```

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.

```
package composite;
import java.util.ArrayList;
import java.util.List;

public class Composite implements Component {
    private final List<Component> children = new ArrayList<>();
    // ... costruttore e altri metodi ...

    @Override
    public List<Double> ottieniMisura() { // Delega e aggrega
        List<Double> misureAggregate = new ArrayList<>();
        for (Component child : children) {
            misureAggregate.addAll(child.ottieniMisura());
        }
        return misureAggregate;
    }

    @Override
    public void notificaMisura() { // Delega la richiesta di notifica
        for (Component child : children) {
            child.notificaMisura();
        }
    }
}
```

Listato 3: Estratto dalla classe Composite

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.

5.3 Observer

5.3.1 Motivazione

Il requisito funzionale 4 richiede che la centralina sia informata automaticamente dei nuovi valori misurati dai sensori. Il *pattern* 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.

5.3.2 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 1, 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)`.

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

```
1  package observer;
2  import model.Misurazione;
3  import java.util.Observable;
4  import java.util.Observer;
5
6  @SuppressWarnings("deprecation")
7  public class Centralina implements Observer {
8      @Override
9      public void update(Observable obs, Object arg) {
10         if (arg instanceof Misurazione misurazione) {
11             // Aggiorna o aggiungi la misurazione alla lista
12             boolean aggiornata = false;
13             for (int i = 0; i < ultimeMisure.size(); i++) {
14                 if (ultimeMisure.get(i).getNomeSensore().equals(misurazione.
       getNomeSensore())) {
15                     ultimeMisure.set(i, misurazione);
16                     aggiornata = true;
17                     break;
18                 }
19             }
20
21             if (!aggiornata) {
22                 ultimeMisure.add(misurazione);
23             }
24
25         } else {
26             System.out.println(nome + ": dato non riconosciuto da " + arg);
27         }
28     }
29     // ... metodo per visualizzare le misure sul terminale
30 }
```

Listato 4: Estratto dalla classe Centralina (Observer)

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

6 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`.

6.1 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 5. 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?");
    assertNotNull(observer.getLastArg(), "Argomento notifica presente?");
    assertInstanceOf(Misurazione.class, observer.getLastArg(), "Tipo argomento
    corretto?");

    // Verifica dati essenziali nella misurazione ricevuta
    Misurazione notifica = (Misurazione) observer.getLastArg();
    assertEquals("TestAdapterA", notifica.getNomeSensore());
    assertEquals(21.5, notifica.getValore(), 0.001);
    // ... (altre asserzioni sui dati della Misurazione omesse)
    assertEquals(21.5, adapter.getUltimaMisuraValore(), 0.001);
}
```

Listato 5: Estratto da AdapterATest: verifica essenziale di notificaMisura

6.2 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.

6.3 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.

6.4 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).

6.5 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 5.

7 Conclusioni

7.1 Risultati ottenuti

L'applicazione dei *pattern* sopra descritti ha portato a un'architettura modulare, più facile da comprendere, mantenere e estendere.

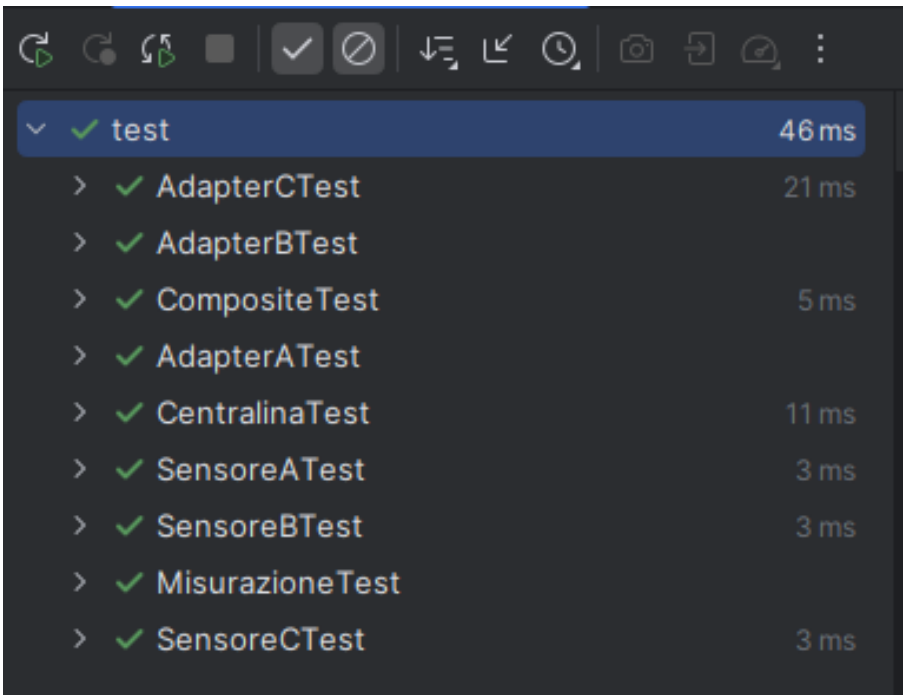

Figura 5: Esito positivo dell'esecuzione della serie di test (cattura schermo, dettaglio)

7.2 Possibili estensioni

Il sistema attuale è facilmente estendibile. Diamo di seguito alcune idee.

- **Aggiunta di nuovi sensori**. Grazie all'Adapter, integrare un nuovo tipo di sensore richiederebbe solo la creazione di un nuovo adattatore specifico.
- **Persistenza dati**. Salvare le misurazioni ricevute dalla centralina su un una base dati per avere uno storico delle misurazioni.
- **Interfaccia grafica**. Sostituire l'interfaccia a riga di comando con un'interfaccia grafica.