Relazione progetto AA 2023-2024 di

# Metodologie di Programmazione

Diciotti Matteo

29 agosto 2024

**Dati autore**

| Cognome | Nome | E-mail | Matricola |
|---|---|---|---|
| Diciotti | Matteo | matteo.diciotti@edu.unifi.it | 7072181 |


## Indice

# 1 Presentazione progetto

Viene presentato nel seguito un progetto fittizio che occorre a mostrare il contesto dal quale prende spunto il codice sviluppato e in cui, ipoteticamente, si vorrebbe inserire.

## 1.1 Contesto: JDS

**J.D.S.**, ovvero **JDS is a Dispaly System**, è un software che nasce dall'esigenza di creare un ambiente di simulazione avanzato per display utilizzati in computer, console e altri dispositivi elettronici. Con l'aumento della complessità e della varietà dei display, è diventato cruciale disporre di strumenti che permettano di testare e ottimizzare le interfacce grafiche in modo efficiente e accurato. JDS si propone di fornire una piattaforma versatile e potente per la simulazione di display, consentendo agli sviluppatori di prevedere e risolvere problemi prima della fase di produzione.
Le **caratteristiche principali** del software sono:

- **Java:** il codice è interamente scritto in linguaggio Java, fornendo ottima portabilità al sistema.
- **Display Simulation:** Riproduzione accurata delle condizioni di visualizzazione su diversi tipi di display.
- **System Compatibility:** Supporto per una vasta gamma di dispositivi, dai computer ai dispositivi mobili.
- **In-depth Analysis:** Funzionalità di monitoraggio e reportistica per identificare e risolvere problemi di performance e qualità.
- **Scalability:** Capacità di adattarsi a progetti di qualsiasi dimensione e complessità.

## 1.2 Inserimento del progetto in JDS

Il progetto presentato al punto precedente si compone di varie parti che cooperano tra loro. In questo contesto possiamo identificare il posizionamento dell'elaborato come la parte centrale del sistema, la parte che implementa i concetti di display e parte dei servizi propri dei sistemi operativi interni agli stessi display. Non vengono quindi prese in considerazione, nel progetto sviluppato, le caratteristiche di reportistica e monitoraggio o strumenti di simulazione avanzati.

# 2 Specifiche Java

L'elaborato è stato implementato e testato attraverso l'utilizzo dell'IDE Eclipse, versione Linux 2024-06 (4.32.0) (fino alla versione Java `java.verision=21.0.4`), sul sistema Kubuntu, kernel `6.8.0-38-generic` e utilizzando per il progetto la versione 11 di Java (`java-11-openjdk-amd64`). Per i test sono stati utilizzati i framework JUnit4 e AssertJ[1].

# 3 Funzionalità del progetto

Il progetto implementa, come detto, un sistema per la simulazione della gestione di display.
I display possono avere nessuno, uno o più sensori, come per esempio un sensore di luminosità esterna, al fine di impostare automaticamente la luminosità dello schermo, o un orologio, al fine di modificare la gamma di colori durante le ore notturne.
Ogni display può incorrere in malfunzionamenti, è quindi stato implementato un meccanismo di controllo dei malfunzionamenti, in particolare nel caso di un cambio inatteso della risoluzione dello schermo o di una disconnessione improvvisa di un'interfaccia video.
Infine, ogni display dovrà proiettare, attraverso un cavo/canale sullo schermo un flusso di frame ricevuti

[1] Il framework AssertJ non è presente tra le librerie standard di Java, quindi è stato inserito come libreria aggiuntiva all'indirizzo "`$ROOT_PROJECT_DIRECTORY/test-libs/assertj-core-3.26.0.jar`" e inserita nel buildpath del progetto Eclipse.

da un'interfaccia video o dei comandi predefiniti. Questi frame possono essere semplici o composti, ovvero possono rappresentare un tipico schermo di un dispositivo, oppure possono essere composti in vari modi, generando quindi un mosaico di frame.

# 4 Diagramma UML

Si mostra a fig.1 il diagramma UML delle classi[2]:

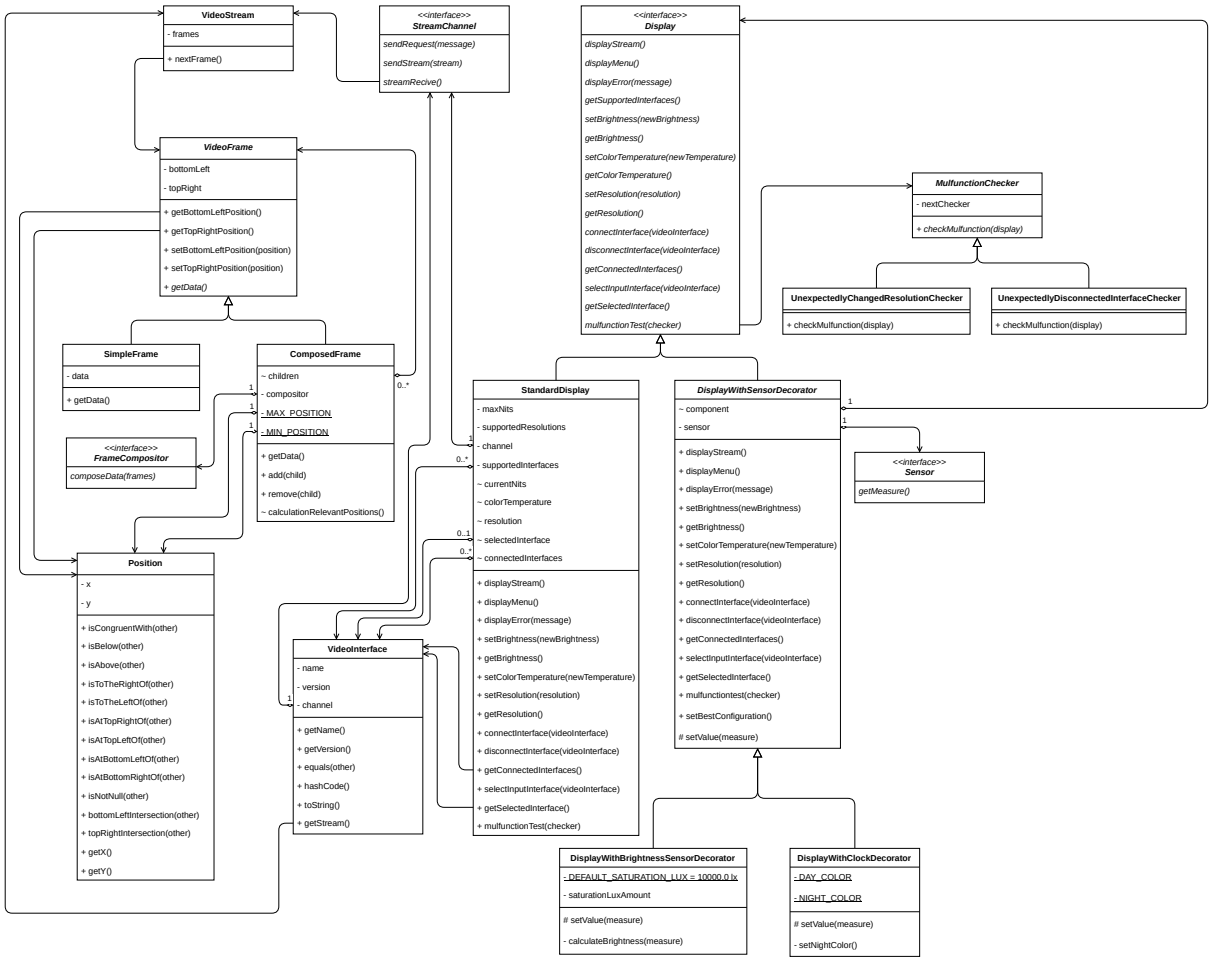


**Figura 1:** Diagramma delle classi descritto con *Unified Modelling Language*.

# 5 Lista pattern implementati

- Decorator, versione con un'unica responsabilità;
- Template Method;
- Chain of Responsability;
- Composite, variante type safe.

[2]Nel diagramma non sono stati inseriti i tipi per una questione di leggibilità data la dimensione dello stesso. Si veda l'allegato AllDiagrams.pdf per prendere visione del diagramma comprendente i tipi.

# 6 Descrizione scelte di design

## 6.1 Gerarchia `Display`, `Sensor` e `StreamChannel`

Per l'implementazione di `Display` è stato utilizzato il pattern **Decorator**[3] per l'aggiunta di un'unica responsabilità, ovvero la gestione di sensori. Il motivo primario che ha portato alla decisione di utilizzare il pattern è la grande varietà di sensori che possono essere associati ad un display, permettendo la creazione dinamica di innumerevoli display in base alle combinazioni di sensori.
L'oggetto da decorare, il *ConcreteComponent*, risulta essere `StandardDisplay` mentre l'unico *Decorator* risulta essere `DisplayWithSensorDecorator`.
I metodi decorati sono *displayStream()*, *displayMenu()* e *displayError(message)* ai quali viene aggiunta la richiesta di impostare preventivamente la miglior configurazione degli attributi del display in base ai valori ottenuti dal sensore presente (vedi metodo *setBestConfiguration()*).
Il motivo per cui `DisplayWithSensorDecorator` è un'astrazione e le sottoclassi `DisplayWithBrightnessSensorDecorator` e `DisplayWithClockDecorator` non sono state implementate come *ConcreteDecorator* è la volontà di ridurre la duplicazione di codice, implementando direttamente le operazioni ereditate da `Display` nella classe base e astraendo solo le operazioni strettamente dipendenti dai sensori, ovvero astraendo *setValue()*. Così facendo viene riutilizzato il codice del decoratore per ogni tipologia di sensore. Attraverso l'uso del pattern **Template Method** è stato possibile definire un algoritmo comune per il metodo *setBestConfiguration()*, il template method (definito `final` nella classe astratta), che richiamasse la funzione *setValue()*, implementata dalle sottoclassi (unico metodo da implementare per creare una classe di tipo `DisplayWithSensorDecorator` e quindi per aggiungere un nuovo tipo di sensore al dispositivo).
Inoltre le classi presenti nel sistema che implementano `DisplayWithSensorDecorator` utilizzano nel metodo *setValue()* dei valori predefiniti per il calcolo. Questa caratteristica è modificabile e deriva dal contesto, si può prevedere una comoda modifica che permetta la specificazione delle variabili in modo che il calcolo sia caratteristico per ogni sensore e per ogni display.
Infine, a proposito di `DisplayWithBrightnessSensorDecorator` e del metodo *setValue()*, è stato implementato un semplice algoritmo di calcolo lineare, ma attraverso l'utilizzo del pattern **Strategy** (non implementato) potevano essere create varie tipologie di algoritmo (lineare, logaritmico, ecc..) in modo che fosse possibile differenziare ulteriormente i display e i loro meccanismi di funzionamento. Tale aggiunta può quindi essere presa in considerazione per raffinazioni future dell'algoritmo di impostazione automatica della luminosità.
Non avendo implementato un meccanismo di logging, il metodo *DisplayWithSensorDecorator.setBestConfiguration()* (così come i metodi *MalfunctionChecker.checkMalfunction(·)* che vedremo in seguito) al momento stampa le eccezioni pervenute dai sensori sulla console. Si prevede una modifica del codice nel senso della creazione di un report complessivo del sistema, ma che risultava non utile al fine dell'elaborato.
Infine, sempre dettata dal contesto in cui si inserisce l'elaborato, è stata utilizzata una semplificazione per la proiezione dei frame sul display, ovvero vengono passati flussi o comandi all'interfaccia `StreamChannel`, la quale poi avrà il compito di gestire le richieste.

[3] Tutta la nomenclatura per la descrizione dei componenti dei pattern proviene dal manuale *Design patterns: elements of reusable object-oriented software*, G.O.F..

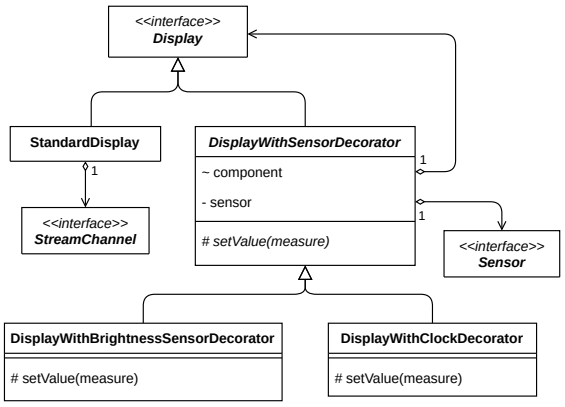


**Figura 2:** Gerarchia `jds.display.Display`.

## 6.2 Gerarchia `MalfunctionChecker`

Per il controllo dei malfunzionamenti dei display è stato implementato il pattern **Chain of Responsibility**, al fine di poter aggiungere controlli dinamicamente e di poter personalizzare la catena in base allo specifico display. Il pattern ha come *Handler* la classe `MalfunctionChecker` e le sottoclassi rappresentano i *ConcreteHandler* che implementano gli specifici controlli da effettuare.
L'accertamento viene affidato da un oggetto `Display` a un oggetto di tipo `MalfunctionChecker` che non se ne occupa direttamente ma che trasmette la richiesta alle sottoclassi. Solo in caso di mancata gestione da parte di una sottoclasse viene chiamato in causa il codice della classe base la quale passa la richiesta al successivo membro della catena oppure restituisce una stringa per specificare l'assenza di malfunzionamenti noti.

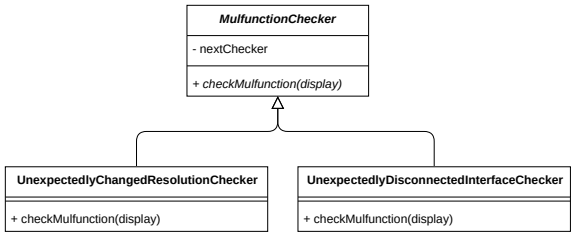


**Figura 3:** Gerarchia `jds.display.Malfunction.MalfunctionChecker`.

## 6.3 `VideoInterface`

Nel progetto non è stata considerata la componente audio e così facendo è stato ritenuto non essere necessario implementare le interfacce video come oggetti staticamente differenti nel sistema, data la scarsa distinzione a livello di operazioni tra le classi nel sistema, ma sono state distinte attraverso degli attributi nella classe `VideoInterface`, nella quale è stato sovrascritto il metodo *equals(·)* per controllare l'uguaglianza tra due oggetti. Si può comunque prevedere una ristrutturazione del codice nel senso della definizione e specifica delle interfacce video e audio con annesse le varie caratteristiche che possiedono.

```
1  public final class VideoInterface {
2
3      private final String name;
4      private final String version;
5      private final StreamChannel channel;
6
7      public VideoInterface(String name, String version, StreamChannel channel) {
8          this.name = Objects.requireNonNull(
9                  name,
```

```
10                "Null name argument");
11        this.version = Objects.requireNonNull(
12                version,
13                "Null version argument");
14        this.channel = Objects.requireNonNull(
15                channel,
16                "Null channel argument");
17    }
```

**Codice 1:** Attributi e costruttore della classe `jds.display.interfaces.VideoInterface`.

### 6.4 `VideoStream` e gerarchia `VideoFrame`

Infine, è stato utilizzato il pattern **Composite** per rappresentare la struttura dei frame di uno stream video. La classe *StreamVideo* possiede un riferimento ad un iteratore di `VideoFrame` i quali possono restituire le informazioni sui pixel in forma di vettore di byte. I `VideoFrame` sono staticamente suddivisi tra `SimpleFrame` e `ComposedFrame`, dove i primi contengono un vettore di byte il quale rappresenta il contenuto dei pixel del frame, mentre i frame composti sono la composizione su due dimensioni dei frame semplici (e di altri frame composti). La classe `SimpleFrame` rappresenta l'unica classe *Leaf* del pattern Composite, mentre la classe `ComposedFrame` rappresenta la classe *Composite*. Il supertipo comune, ovvero il *Component* del pattern, è l'interfaccia `VideoFrame`. Come prevede il pattern, prima che i `ComposedFrame` rimandino la richiesta per l'ottenimento delle informazioni sui pixel ad un'interfaccia `FrameComposer`, che ha il compito di creare attraverso qualche specifico algoritmo un array di byte che rappresenti le informazioni sui pixel della composizione, richiedono ai figli, attraverso la medesima operazione *getData()*, i dati necessari alla composizione. Si deve infine notare che la variante implementata è quella prona alla sicurezza dei metodi, ovvero quella che evita errori a run-time. Per questo motivo la presenza dei metodi *add(child)*, *remove(child)* eccetera è esclusiva della classe concreta `ComposedFrame`.

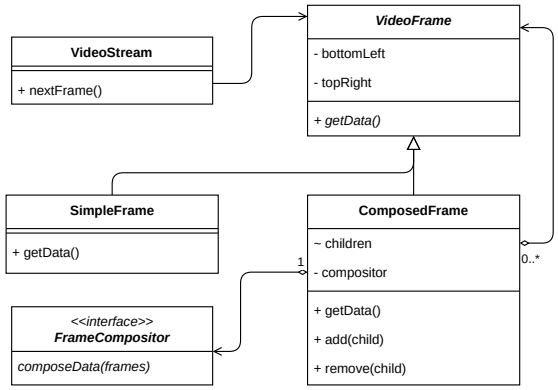


**Figura 4:** `jds.display.videostream.VideoStream` e gerarchia `jds.display.videostream.VideoFrame`.

## 7 Test

Si presentano nel seguito una serie di note preliminari che sono generali per tutto l'elaborato:

- I test sono stati scritti in forma non completamente estesa, ovvero inserendo più asserzioni all'interno di un unico JUnitTest per controllare la correttezza di un intero metodo del *Software Under Test* (o *SUT*) e non il singolo comportamento di questo. È stato ritenuto che la metodologia fosse accettabile dato il contesto, in quanto vengono comunque testati separatamente i metodi del SUT, permettendo l'identificazione dell'origine dei fallimenti, ma vengono tenuti uniti i test dei vari comportamenti del singolo metodo, limitando l'eccessiva produzione di micro-JUnitTest (che generalmente sarebbe da considerare la procedura corretta sebbene produca un codice più ampio).

- È stato deciso di non utilizzare `extracting(String)` e `hasFieldOrPropriety(String)` (e affini), metodi di `assertj`, per testare lo stato di oggetti poiché questi, sebbene siano metodi molto potenti, hanno il difetto di rompersi dopo che dei campi testati vengono rinominati, facendo venir meno uno dei principi dei test e del refactoring[4].
- È stato ritenuto che testare i metodi auto-generati dall'IDE, ovvero getters e setters ai quali non è stata aggiunta ulteriore logica, non fosse determinante e che aumentava inutilmente la dimensione dell'elaborato. Questa decisione è stata presa con la consapevolezza che, in un contesto non accademico, sarebbe stato preferibile implementarli comunque per verificare la correttezza dei metodi in futuro, nel caso in cui venisse aggiunta logica non testata.
- Il motivo per cui sono presenti dei campi definiti package-private (visibili a fig. 1 con simbolo ~) è da far risalire all'utilizzo di questi campi all'interno dei test: nessun campo definito package-private è stato utilizzato direttamente all'interno delle classi del medesimo pacchetto, è stata sempre utilizzata l'interfaccia pubblica (public e protected) delle classi.

Per testare il progetto sono state implementate all'interno della cartella sorgente *tests* le classi fittizie `jds.MockStreamChannel` e `jds.videostream.MockFrameCompositor`, servizi non implementati del sistema, e la classe `DisplaySetterToTestMalfunctions` che occorre nelle classi di test della gerarchia `MalfunctionChecker` per settare direttamente alcuni campi di display al di fuori dal pacchetto in cui risiede la classe `StandardDisplay`.

### 7.1 Gerarchia `Display`

Per testare la gerarchia Display è necessario testare i pattern **Decorator** e **Template Method**. Si è testato innanzitutto la classe `StandardDisplay`, ovvero il componente da decorare, attraverso test che per ogni metodo ne valutassero il corretto funzionamento, sia per il lancio di eccezioni (attraverso il metodo *assertThatThrownBy(·)* e l'utilizzo di lambda) che per il corretto svolgimento delle operazioni attese. Impostando alcuni argomenti della classe package-private è stato possibile settare direttamente i valori senza utilizzare altri metodi parte dell'interfaccia pubblica della classe, in modo da isolare il comportamento il più possibile.
In secondo luogo si è passati a testare la classe `DisplayWithSensorDecorator`, in particolare lo si è fatto all'interno delle classi di test relative alle sottoclassi che la implementano: `DisplayWithBrightnessSensorDecoratorTest` e `DisplayWithClockDecoratorTest`. Per creare delle implementazioni di `Sensor`, invece di produrre classi aggiuntive, dato che `Sensor<T>` è un'interfaccia funzionale, sono state assegnate delle lambda expressions con cast a sensori.
In queste classi e stato testato su una catena di 2 decoratori (differenti) e un componente che il comportamento dei metodi decorati ovvero *displayStream()*, *displayMenu()* e *displayError(message)*, modificasse lo stato dell'oggetto *component* alla base della "catena" di decoratori e che poi passasse la richiesta al successivo *decorator* o al *component*. Per tutti gli altri metodi è stato testato che venisse passata la richiesta al *component* senza svolgere ulteriori operazioni (e che quindi le eccezioni lanciate fossero uguali o che il risultato di un metodo fosse uguale al risultato del metodo chiamato sul *component*). Sono state inoltre testate le due implementazioni del metodo *setValue()* per accertare la correttezza del pattrern template method, oltre ovviamente al metodo *setBestConfiguration()*.

### 7.2 Gerarchia `MalfunctionChecker`

Per testare che il pattern **Chain of Responsability** abbia il corretto funzionamento si controlla nelle classi di test `UnexpectedlyChangedResolutionCheckerTest` e `UnexpectedlyDisconnectedInterfaceCheckerTest` che, se la classe che deve gestire la richiesta e attualmente non può/sa farlo allora passa la richiesta al successivo *ConcreteHandler*, altrimenti si controlla che restituisca il valore atteso.

[4] Il metodo `org.assertj.core.api.Assertions.assertThat(.)` è deprecato solo per alcuni tipi dei parametri (si può notare dall'IDE che lo segnala contestualmente al comando di `import`). I metodi utilizzati nel progetto possiedono un nome equivalente a quello presentato ma una firma differente: questi non risultano deprecati (si può infatti notare che l'IDE non li barra direttamente nel codice). Si veda a proposito la documentazione della libreria `assertJ`.

```
    @Test
    public void testCheckMulfunction() throws AbsentVideoInterfaceException {
        display.connectInterface(videoInterface);
        display.selectInputInterface(videoInterface);
        String noMulfunction = "No mulfunction detected";
        assertThat(checker1.checkMulfunction(display))
            .isEqualTo(noMulfunction);
        assertThat(checker2.checkMulfunction(display))
            .isEqualTo(noMulfunction);
        DisplaySetterToTestMalfunctions.setResolution(display, "4k");
        assertThat(checker1.checkMulfunction(display))
            .isEqualTo("Unexpectedly changed resolution");
        DisplaySetterToTestMalfunctions.setResolution(display, "4k");
        assertThat(checker2.checkMulfunction(display))
            .isEqualTo("Unexpectedly changed resolution");
    }
```

**Codice 2:** Test per la classe `jds.display.malfunction.UnexpectedlyChangedResolutionCheckerTest`.

### 7.3 `VideoInterface`

Nella classe `VideoInterface` è stato testato il corretto funzionamento dei metodi utilizzando le medesime tecniche utilizzate per le classi precedenti. In particolare sono stati testati i metodi *equals(·)*, *hashCode()* e *toString()* secondo quanto atteso in letteratura (*equals* risultante false in caso di `other` assegnato a null, *hashCode* uguale per elementi uguali, eccetera).

```
    @Test
    public void testEquals() {
        StreamChannel secondChannel = new MockStreamChannel();
        VideoInterface equalInter = new VideoInterface(name, version, channel);
        VideoInterface diffName  = new VideoInterface("HDMI", version, channel);
        VideoInterface diffVersion = new VideoInterface(name, "SXGA", channel);
        VideoInterface diffChannel = new VideoInterface(name, version, secondChannel);
        VideoInterface completelyDiff= new VideoInterface("HDMI", "1.3", secondChannel);
        assertThat(inter.equals(equalInter))
            .isEqualTo(equalInter.equals(inter))
            .isEqualTo(inter.equals(diffChannel))
            .isTrue();
        assertThat(inter.equals(completelyDiff))
            .isEqualTo(completelyDiff.equals(inter))
            .isEqualTo(inter.equals(diffName))
            .isEqualTo(inter.equals(diffVersion))
            .isFalse();
    }
```

**Codice 3:** Test del metodo *equals(other)* per la classe `jds.display.interfaces.VideoInterface`.

### 7.4 `VideoStream` e gerarchia `VideoFrame`

Una volta testata la classe `Position` con tecniche non dissimili da quelle precedentemente utilizzate, testando per ogni metodo che il risultato atteso sia coerente con quello ottenuto, si eseguono i test per `SimpleFrame`, ovvero si testa quel minimo di logica dei setters.
Si è testata successivamente la classe `ComposedFrame`, utilizzando la classe `MockFrameCompositor`. Dato che si tratta di un pattern **Composite** si testa che il metodo *getData()*, operazione astratta della classe base, sia passata dal *Composite* ai figli, e che quindi, in questo caso per come è stata implementata la classe `MockFrameCompositor`, che il risultato ottenuto contenga tutte le istanze dei dati dei figli.

```
    @Test
    public void testGetData() {
        assertThat(composed1.getData())
```

```
            .contains(data1);
        assertThat(composed2.getData())
            .contains(data1)
            .contains(data2)
            .contains(data3);
    }
```

**Codice 4:** Test del metodo *getData()* per la classe `jds.display.videostream.ComposedFrame`.

Infine è stata testata la classe `VideoStream` affinché fosse confermato che il metodo *nextFrame()* restituisse sempre il corretto frame.

```
    @Test
    public void testNextFrame() {
        assertThat(stream.nextFrame())
            .isSameAs(simple1);
        assertThat(stream.nextFrame())
            .isSameAs(simple2);
        assertThat(stream.nextFrame())
            .isSameAs(composed);
        assertThatThrownBy(() -> stream.nextFrame())
            .isInstanceOf(AbsentFrameException.class)
            .hasMessage("No frame found");
    }
```

**Codice 5:** Test del metodo *nextFrame()* per la classe `jds.display.videostream.VideoStream`.