Relazione progetto AA 2023-2024 di

Metodologie di Programmazione

Diciotti Matteo

27 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, aumentando la portabilità dello stesso progetto.
- **Display Simulation:** Riproduzione accurata delle condizioni di visualizzazione su diversi tipi di display.
- **System Compatibility:** Supporto per una vasta gamma di dispositivi, dai computer alle console di gioco.
- **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 implementano 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 questo 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 generazione e la 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.
Data la varietà dei possibili schermi, è stato implementato un costruttore per questi che rendesse la creazione meno complessa, attraverso un'interfaccia più fluida.
Infine, ogni display dovrà proiettare, attraverso un cavo/canale sullo schermo un flusso di frame ricevuti de un'interfaccia video. Questi frame possono essere semplici o composti, ovvero possono rappresentare un tipico schermo di un dispositivo, oppure possono essere composti in vari modi, quindi generando un mosaico di frame.

[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.

4 Diagramma UML

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

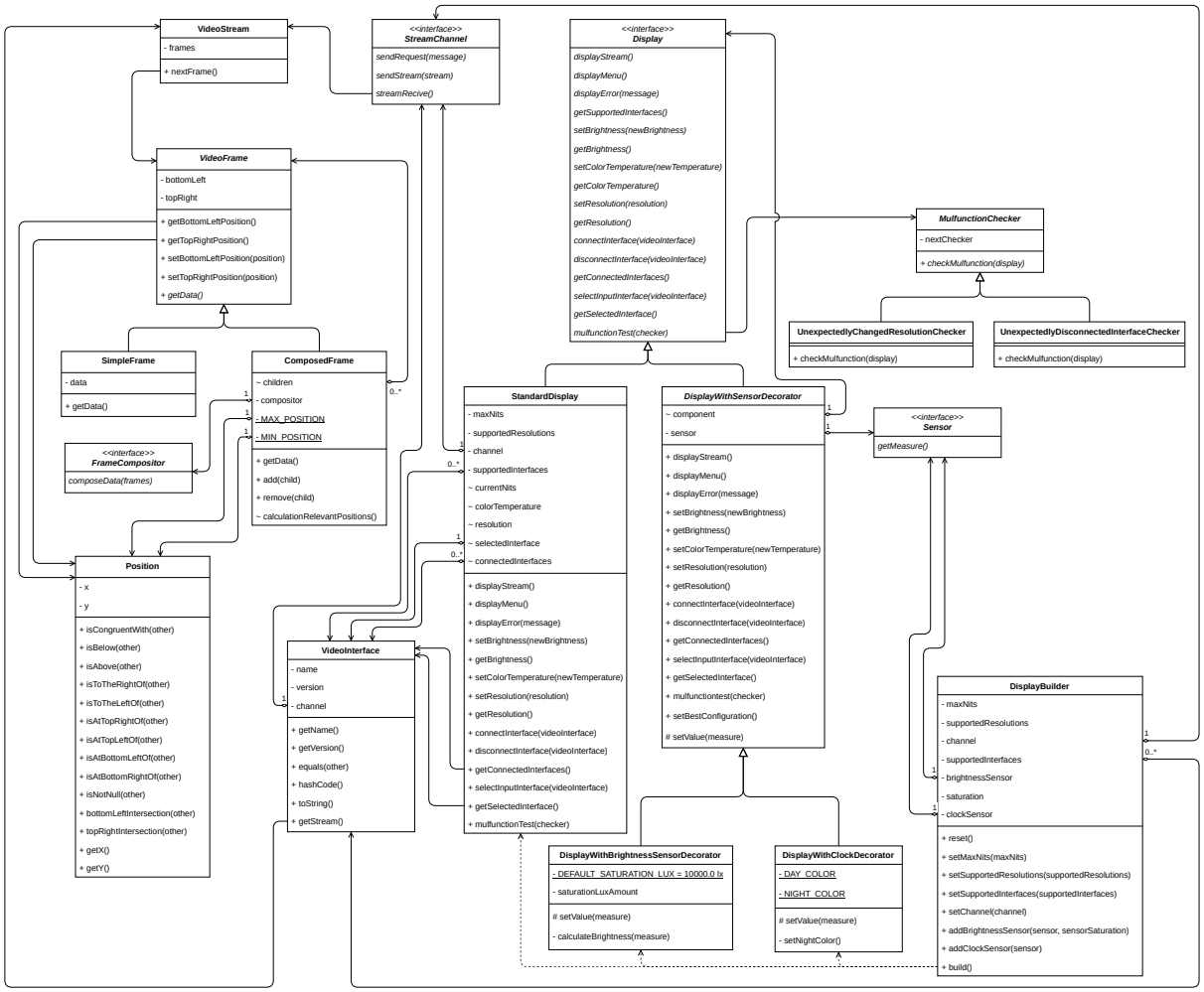

Figura 1: Diagramma delle classi descritto con *Unified Modelling Lenguage*.

5 Lista pattern implementati

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

6 Descrizione scelte di design

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

Per l'implementazione di `Display` è stato utilizzato il pattern **Decorator**[2] 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`.
Il motivo per cui `DisplayWithSensorDecorator` è un'astrazione e le sottoclassi `DisplayWithBrightnessSensorDecorator` e `DisplayWithClockDecorator` non sono stati implementati 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()*. Cosi facendo viene riutilizzato il codice del decoratore per ogni tipologia di sensore. Attraverso l'uso quindi del pattern **Template Method** è stato possibile definire un algoritmo comune per il metodo *setBestConfiguration()*, il template method, che richiamasse la funzione *setValue()*, implementata dalle sottoclassi (unico metodo da implementare per creare un nuovo decoratore di tipo `DisplayWithSensorDecorator`).
Inoltre le classi presenti nel sistema che implementano `DisplayWithSensorDecorator` utilizzano nel metodo *setValue()* dei valori di default predefiniti. 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 specifico 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 specifici.
Non avendo implementato un meccanismo di logging, il metodo *DisplayWithSensorDecorator.setBestConfiguration()* (così come i metodi *MulfunctionChecker.checkMulfunction(·)* 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 del progetto.
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.

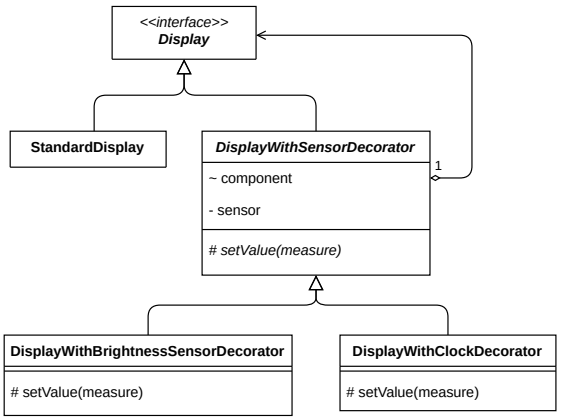

Figura 2: Gerarchia Display.

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

6.2 Gerarchia `MulfunctionChecker`

Per il controllo dei malfunzionamenti 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 `MulfunctionChecker` e le sottoclassi rappresentano i *ConcreteHandler* che implementano gli specifici controlli da effettuare.

6.3 `Builder`

Data la varietà delle tipologie di Display e data la grande quantità di argomenti che questi possiedono è stato ritenuto che l'implementazione del pattern **Builder**, sebbene nella versione senza astrazione, potesse rendere più agevole generazione di display, permettendone la creazione in modo fluido. La classe `DisplayBuilder` rappresenta l'implementazione del pattern.

6.4 `VideoInterface`

Nel progetto non è stata considerata la componente audio e così facendo è stato ritenuto non essere necessario implementare le interfacce audio come oggetti staticamente differenti, 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 portano.

6.5 `VideoStream` e gerarchia `VideoFrame`

Infine è stato utilizzato il pattern **Composite** per rappresentare la struttura dei frame. I `VideoFrame` sono staticamente suddivisi tra `SimpleFrame` e `ComposedFrame`, dove i primi rappresentano un rettangolo e 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). I secondi rimandano la richiesta dell'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.

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).
- È stato deciso di non utilizzare `extractiong(String)` e `hasFieldOrPropriety(String)` (e affini), metodi di `assertj`, per testare lo stato di oggetti poiché questi, sebbene molto potenti, hanno il difetto di rompersi rinominando i campi, quindi vengono meno a uno dei principi dei test.[3]

[3] 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`.

- 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 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.