



UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

Elaborato di Ingegneria del Software

A.A. 2021/2022

Elia Pitozzi
Ali Laaraj

Matr. 727344
Matr. 727655

Indice della presentazione

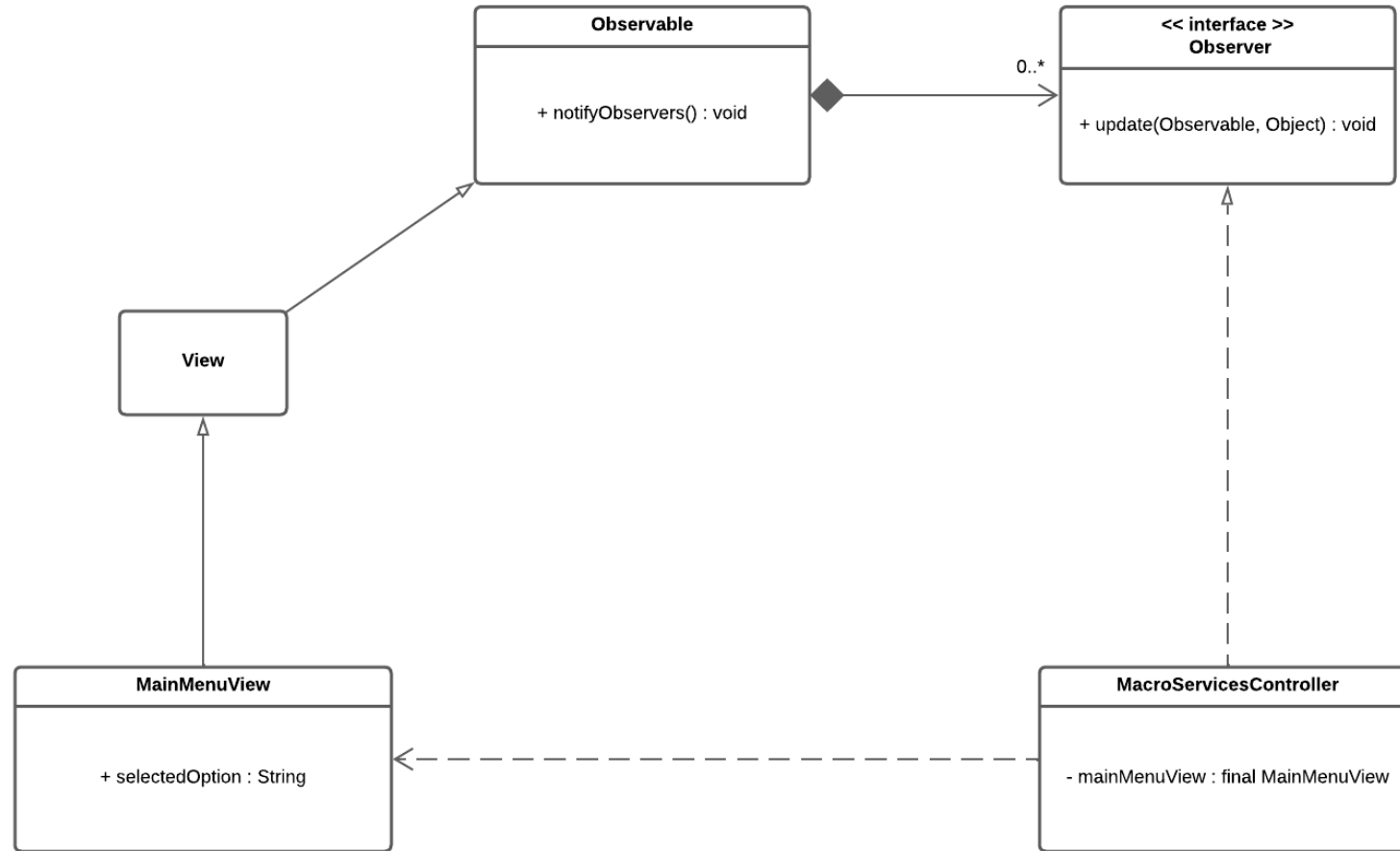
➤ Principio di separazione modello-vista	1-3
➤ GRASP High Cohesion	4
➤ GRASP Protected Variation	5-6
➤ SOLID Interface Segregation Principle	7-8
➤ SOLID Dependency Inversion Principle	9-10
➤ GoF State	11-12
➤ GoF Chain of Responsibility	13-14
➤ Testing	15-19
➤ Refactoring	20-21

Separazione modello-vista

The background of the slide is composed of several overlapping geometric shapes. On the left, there is a solid light green shape. To its right is a large white area containing the text. Further right, there are vertical bands of black and grey, some with a wavy, organic pattern. On the far right, there are more green shapes, some with a wavy pattern and others solid, creating a layered, abstract effect.

Separazione modello-vista

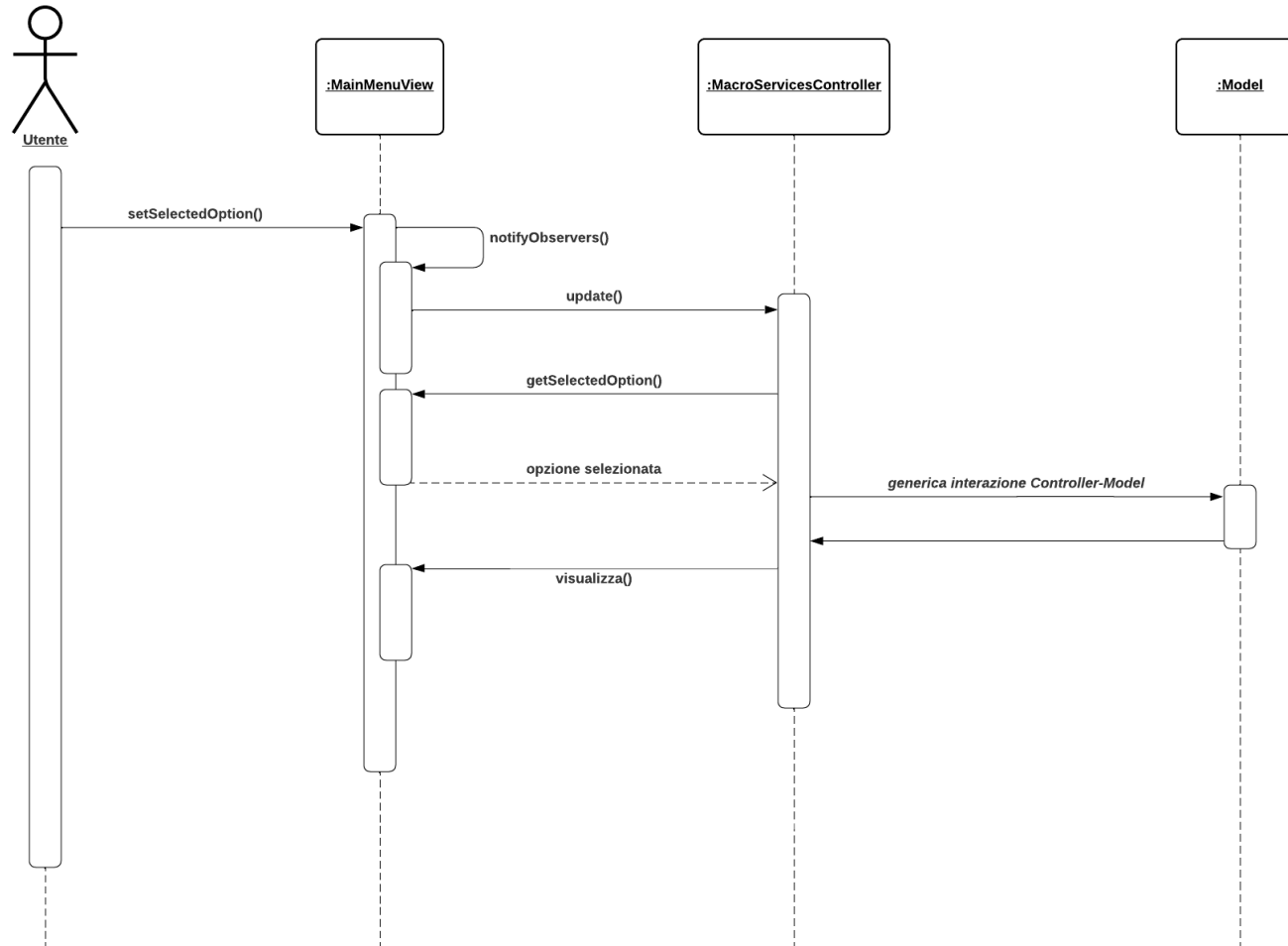
Diagramma UML



*Basato sul pattern
Observer, uno
schema stile
publisher-
subscriber.*

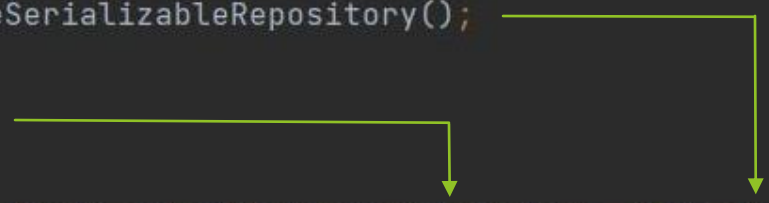
Separazione modello-vista

Diagramma di sequenza



Il pattern MVC nel codice

```
final var gestoreOfferte = new GestoreOfferteSerializableRepository();  
  
final var mainMenuView = new MainMenuView();  
  
final var macroServicesController = new MacroServicesController(mainMenuView, gestoreOfferte);
```



```
graph LR;
    gestoreOfferte --> macroServicesController;
    mainMenuView --> macroServicesController;
```

Model e View non comunicano tra di loro: infatti, a un oggetto Model (**gestoreOfferte**) non passiamo riferimenti della View.
Il Controller, invece, fa da ponte e riceve riferimenti sia del Model che della View.

```
private void aggiungiSottoCategorie(GerarchiaDiCategorie gerarchia) {  
    if (view.chiediConfermaInserimentoSottoCategoriaConNome(gerarchia.getNome())) {  
        var categoriaFiglio : CategoriaFiglio = chiediECreaCategoriaFiglioIn(gerarchia);  
        var gerarchiaFiglio : GerarchiaDiCategorie = gerarchia.inserisciSottoCategoria(categoriaFiglio);  
    }  
}
```

```
public void setSelectedOption(String selectedOption) {  
    this.selectedOption = selectedOption;  
    setChanged();  
    notifyObservers();  
}
```

Controller → View

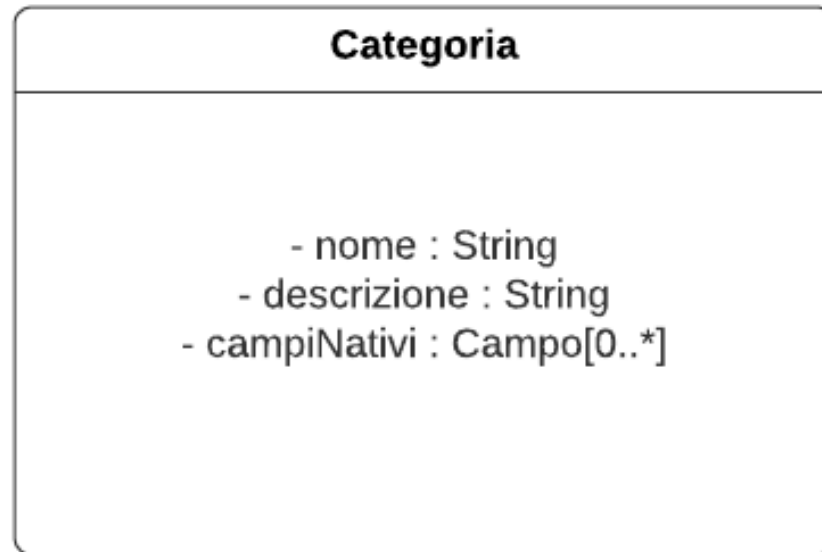
View → Controller



GRASP High Cohesion

GRASP High Cohesion

Esempio di riferimento: classe *Categoria*



Classe inizialmente poco coesa visto che si occupa di diversi compiti poco coerenti ed è qui che entra in gioco High Cohesion.



L'uso di questo pattern ci ha portato ad apportare una correzione nell'implementazione del progetto, con Chain of Responsibility (GoF).



GRASP Protected Variation

GRASP Protected Variation

Esempio di riferimento: *GestoreOfferte* e *OfferteService*



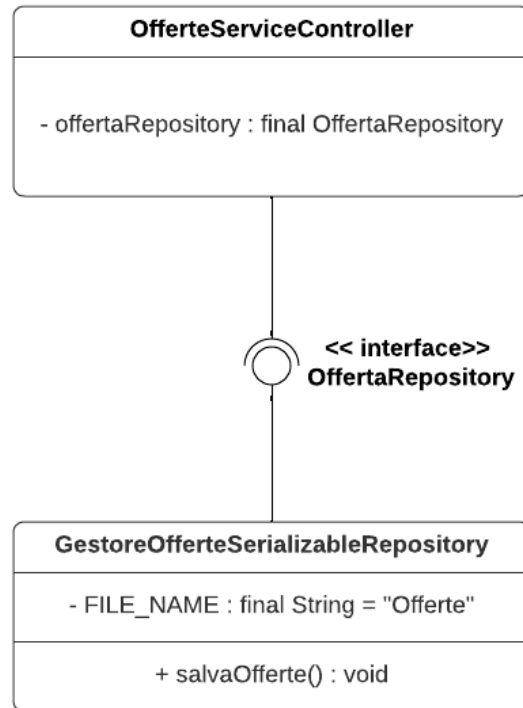
La classe di dominio relativa alla persistenza (**GestoreOfferte**) risulta accoppiata direttamente a una classe utilizzatrice della prima (**OfferteService**)



Ciò però è contro il pattern Protected Variation in quanto un cambio all'interno della classe di dominio (per sua natura instabile) può avere impatti indesiderati sulle classi utilizzatrici.

GRASP Protected Variation

La nostra soluzione



Costruiamo per ogni classe di persistenza un'interfaccia stabile, da cui le classi utilizzatrici ora dipenderanno. L'interfaccia proteggerà queste dalle variazioni delle classi di persistenza.

L'analisi di Protected Variation ci ha portato a valutare, come conseguenza, un'applicazione del pattern SOLID Dependency Inversion Principle come effetto sulle classi di progetto.

SOLID Interface Segregation Principle



SOLID Interface Segregation Principle

L'idea di fondo

Ogni classe di dominio del nostro progetto implementa due interfacce: **Serializable**, per la gestione della persistenza dei dati e **DomainTypeToRender** per la gestione del rendering (mediante Chain Of Responsibility che vedremo in seguito).

```
public interface DomainTypeToRender {  
}
```

*L'interfaccia marker **DomainTypeToRender***

Dato che tutte le suddette classi implementano due interfacce, potremmo far implementare alle classi una sola interfaccia (**DomainType**) che estenda le due interfacce:

```
public interface DomainType extends Serializable, DomainTypeToRender {  
}
```

Violazione di ISP!



Grazie al pattern ISP si è deciso che fosse meglio lasciare queste due interfacce slegate tra di loro perché potessero essere più piccole e più coese riguardanti ciascuna uno specifico comportamento.

SOLID Interface Segregation Principle

Ulteriori note

- ISP suggerisce interfacce più snelle perché non tutti i suoi metodi sono sempre implementati dagli implementatori dell'interfaccia ma sia ***Serializable*** che ***DomainTypeToRender*** sono interfacce marker, quindi non hanno metodi!
- Creare un'unica interfaccia che aggregasse due interfacce, una nel contesto del rendering per i tipi di dominio e l'altra nel contesto della persistenza dei dati sarebbe comodo ma sarebbe anche una chiara violazione del pattern ISP in quanto l'interfaccia unica si occuperebbe di compiti diversi.



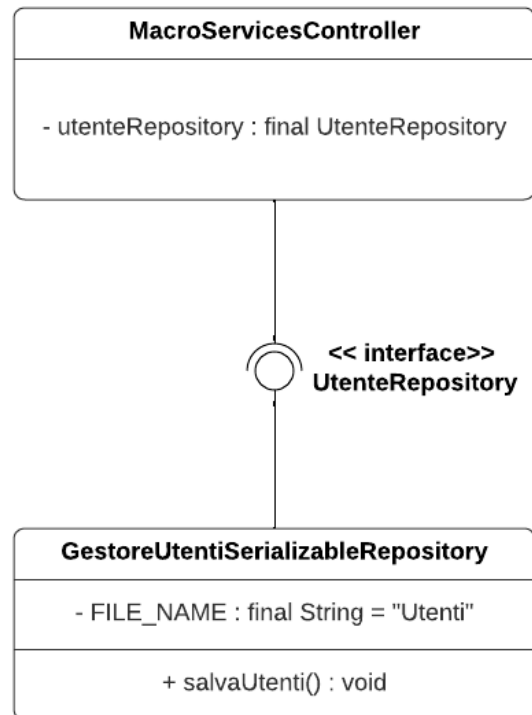
SOLID Dependency Inversion Principle



SOLID Dependency Inversion Principle

Esempio di riferimento: *UtenteRepository*

I moduli di più alto livello e quelli di più basso livello non interagiscono più tra di loro in maniera diretta: così le classi di alto livello non dipendono più dalle classi di basso livello con i loro dettagli implementativi ma entrambi ora dipendono da astrazioni.



Creiamo un'interfaccia apposita di tipo Repository che venga implementata da ogni classe di dominio Gestore. Il controller che utilizzerà l'interfaccia non sarà a conoscenza delle implementazioni dell'interfaccia e le conoscerà solo alla chiamata del costruttore.

SOLID Dependency Inversion Principle

DIP nel codice

Nel metodo `run()` della classe `App` creiamo un nuovo oggetto di tipo **GestoreUtentiSerializableRepository** che verrà passato come parametro al costruttore di **MacroServicesController**. Si può vedere chiaramente l'inversione delle dipendenze:

```
final UtenteRepository gestoreUtenti = new GestoreUtentiSerializableRepository();  
final var macroServicesController = new MacroServicesController(gestoreUtenti);
```

Questo invece è il costruttore di **MacroServicesController** che riceve in ingresso un oggetto di tipo *UtenteRepository*:

```
3 usages  
private final UtenteRepository utenteRepository;  
1 usage new *  
public MacroServicesController(UtenteRepository utenteRepository) {  
    this.utenteRepository = utenteRepository;  
}
```

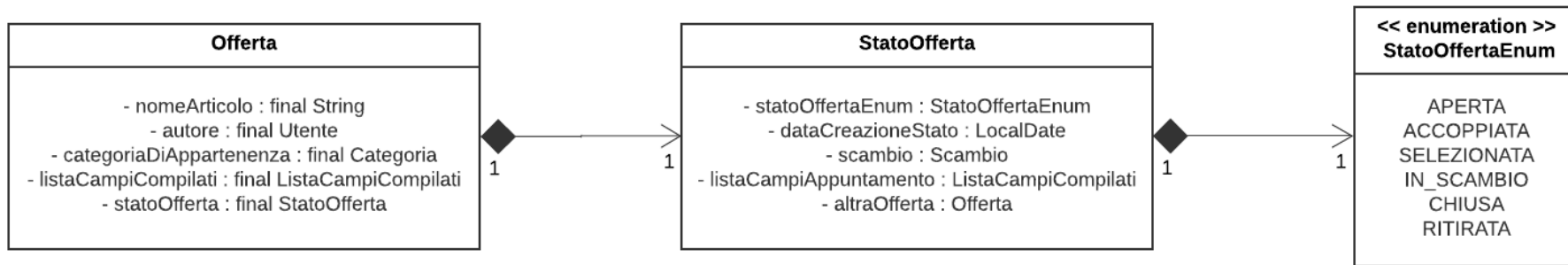
The background features a complex abstract design. On the left, a light gray area contains a sparse collection of black dots. The central portion is a white field overlaid with a dense, intricate network of thin gray lines and various colored circular nodes (pink, blue, green, brown). To the right, there are overlapping translucent green shapes, including a large triangle and several smaller, more complex geometric forms. The overall aesthetic is modern and technical.

GoF State

GoF State

Esempio di riferimento: classe Offerta

Nella classe **Offerta** ci sono molti if che in funzione di **StatoOfferta** eseguono codice diverso (a seconda del diverso stato dell'offerta) per la stessa chiamata al medesimo metodo.



Violazione
di
OCP!

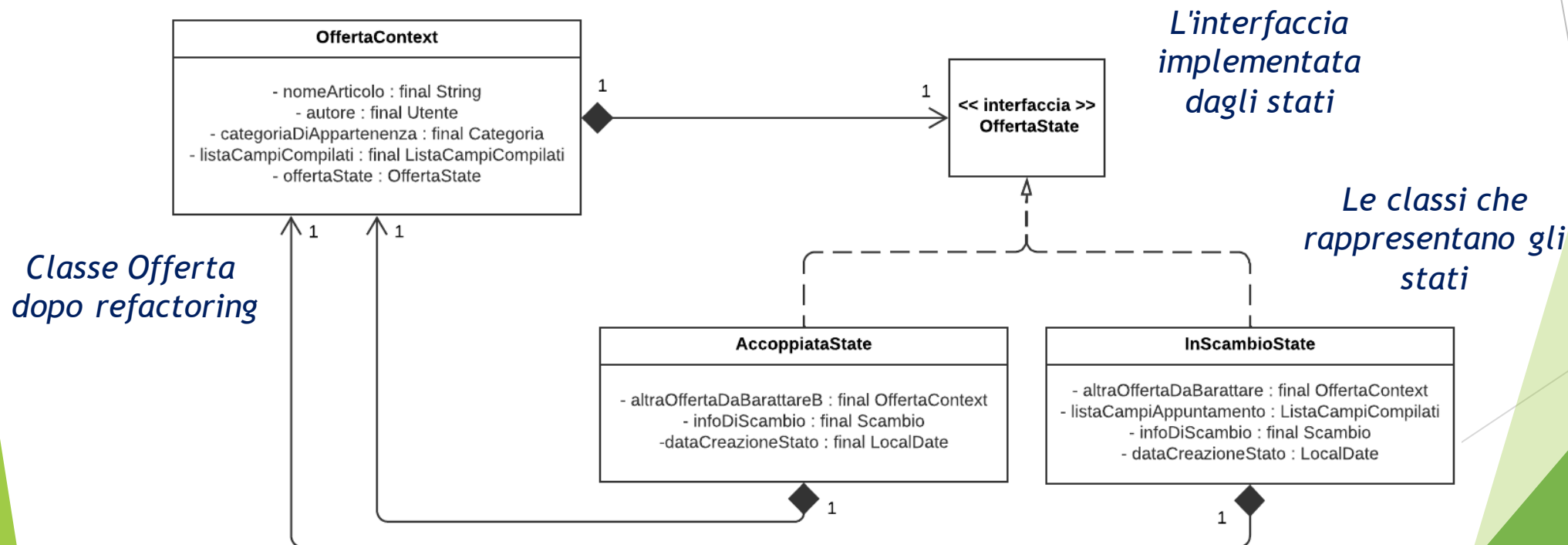


Il problema di questo approccio è che se in futuro ci fosse un'estensione del codice con un nuovo stato dell'offerta, tutti gli if e l'intero codice dei vari metodi andrebbero ogni volta riadattati per supportare il nuovo stato e questa, appunto, sarebbe una violazione di Open Closed Principle.

GoF State

La nostra soluzione

Dato che il comportamento di **Offerta** è condizionato dal suo stato interno, introduciamo un'interfaccia comune che verrà implementata dalle classi, ciascuna delle quali rappresenta uno stato diverso di **Offerta**.





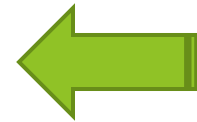
GoF Chain of Responsibility

GoF Chain of Responsibility

Il problema del rendering

Dover renderizzare una classe di dominio con il metodo *toString()* viola MVC e OCP: di conseguenza, realizziamo una classe apposita per ogni classe di dominio che si occupa del suo rendering così in caso di rendering alternativi (HTML, JSON...) basta aggiungere altro codice senza modificare né le classi di dominio né le classi di rendering iniziali.

```
public class UtenteRenderer implements SelectableDomainTypeRenderer {  
  
    /** Metodo che restituisce il rendering dell'oggetto in input ...*/  
    @Aile  
    @Override  
    public String render(DomainTypeToRender domainTypeToRender) {  
        Utente utente = (Utente) domainTypeToRender;  
        return "Utente{" + "username='" + utente.getUsername() + '\'' + '}';  
    }  
  
    /** Metodo che controlla se il renderer può gestire ...*/  
    @Aile  
    @Override  
    public boolean canHandle(DomainTypeToRender domainTypeToRender) {  
        return domainTypeToRender instanceof Utente;  
    }  
}
```



Esempio di classe per il rendering dell'utente: ogni nostra classe renderer implementerà un'interfaccia con due metodi, *render()* e *canHandle()*.

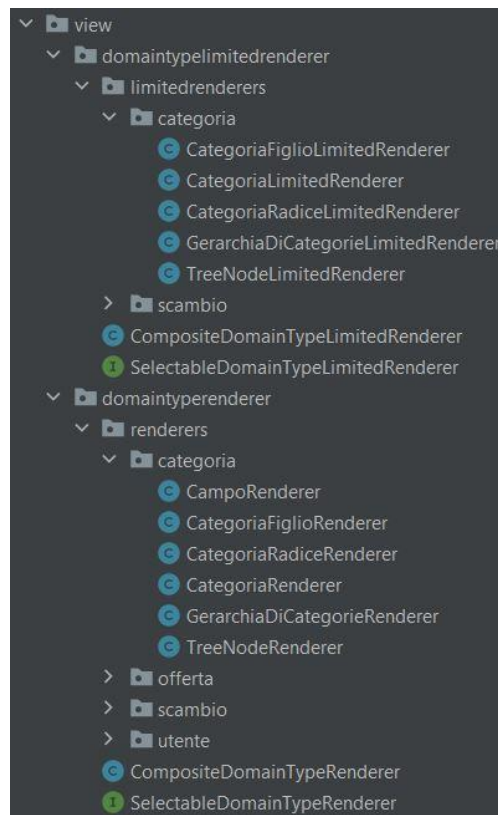
GoF Chain of Responsibility

Strutturazione del codice

Abbiamo applicato questo pattern "2 volte": una per la gestione del rendering classico e un'altra per la gestione del rendering limitato che è possibile avere per le categorie e gli scambi.

Classi per il rendering limitato

Classi per il rendering classico



Tutto il codice per il rendering è stato messo nella View per rispettare MVC

Interfaccia implementata dai renderer limitati

Interfaccia implementata dai renderer classici



Testing

Testing

Testing a un metodo (black-box)

La macro-tipologia di testing scelta per operare sul metodo è quella del testing black-box: il testing consiste nel creare un insieme di condizioni di input per esercitare i requisiti funzionali di interesse, i quali vengono poi verificati tramite il confronto con un output atteso. Il metodo scelto è il seguente (della classe IntervalloOrario):

```
public boolean intersecaAltroIntervalloOrario(IntervalloOrario altroIntervalloOrario)
```

Si struttura il testing per rilevare in particolare:

- Errori di interfaccia
- Presenza di controlli errati o mancanti

Di conseguenza, strutturiamo i test per **classi di equivalenza**: la condizione di input specifica un insieme (c'è intersezione o meno). Generiamo diversi test case per testare le diverse intersezioni tra quelle possibili.

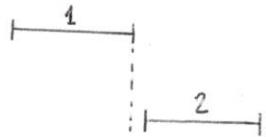
Infine con l'**analisi dei valori limite** strutturiamo test case che utilizzino i valori di frontiera delle classi di equivalenza individuate.

Testing

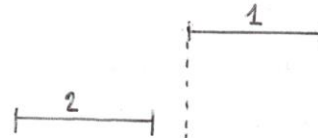
Testing a un metodo: le classi di equivalenza

Le classi di equivalenza individuate per testare il metodo mostrato precedentemente sono due: la prima si ha nel caso non ci sia intersezione tra due intervalli orari e la seconda nel caso ci sia intersezione tra due intervalli orari. Di seguito, mostriamo un disegno esplicativo delle due classi di equivalenza da noi individuate:

1° CLASSE DI EQUIVALENZA: TEST PER CUI NON C'E' INTERSEZIONE (OUTPUT: FALSE)

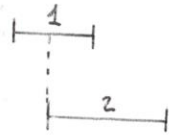


Intervallo 1 viene prima che intervallo 2 inizi

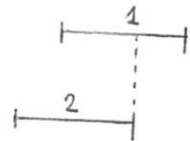


Intervallo 1 viene dopo che intervallo 2 finisce

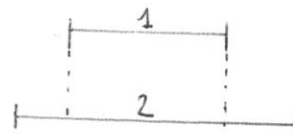
2° CLASSE DI EQUIVALENZA: TEST PER CUI C'E' INTERSEZIONE (OUTPUT: TRUE)



Inizio di intervallo 2 compresa in intervallo 1



Fine di intervallo 2 compresa in intervallo 1



Intervallo 1 compreso strettamente in intervallo 2

Testing

Testing a una funzionalità (white-box)

La macro-tipologia di testing scelta per operare sulla funzionalità è quella del testing white-box: grazie all'accesso al codice strutturiamo i test case direttamente dalla struttura di controllo del codice della funzionalità garantendo che ogni componente del codice sia effettivamente eseguita e risponda a controlli standardizzati per la qualità del software. La funzionalità scelta è la seguente (della classe IntervalloOrario):

```
public List<LocalTime> getListaOrariValidi()
```

Le tipologie di testing white-box scelte sono:

- **Code Coverage**
- **Code Inspection**

Testing

Testing a una funzionalità: Code Coverage

Per la realizzazione di Code Coverage è bastato creare un unico test case che richiamasse una singola volta il codice della funzionalità. Poi, con l'esecuzione del test case tramite il Debugger è stato possibile osservare che tutte le parti del codice da analizzare erano effettivamente richiamate ed eseguite.

I livelli di copertura che analizziamo sono:

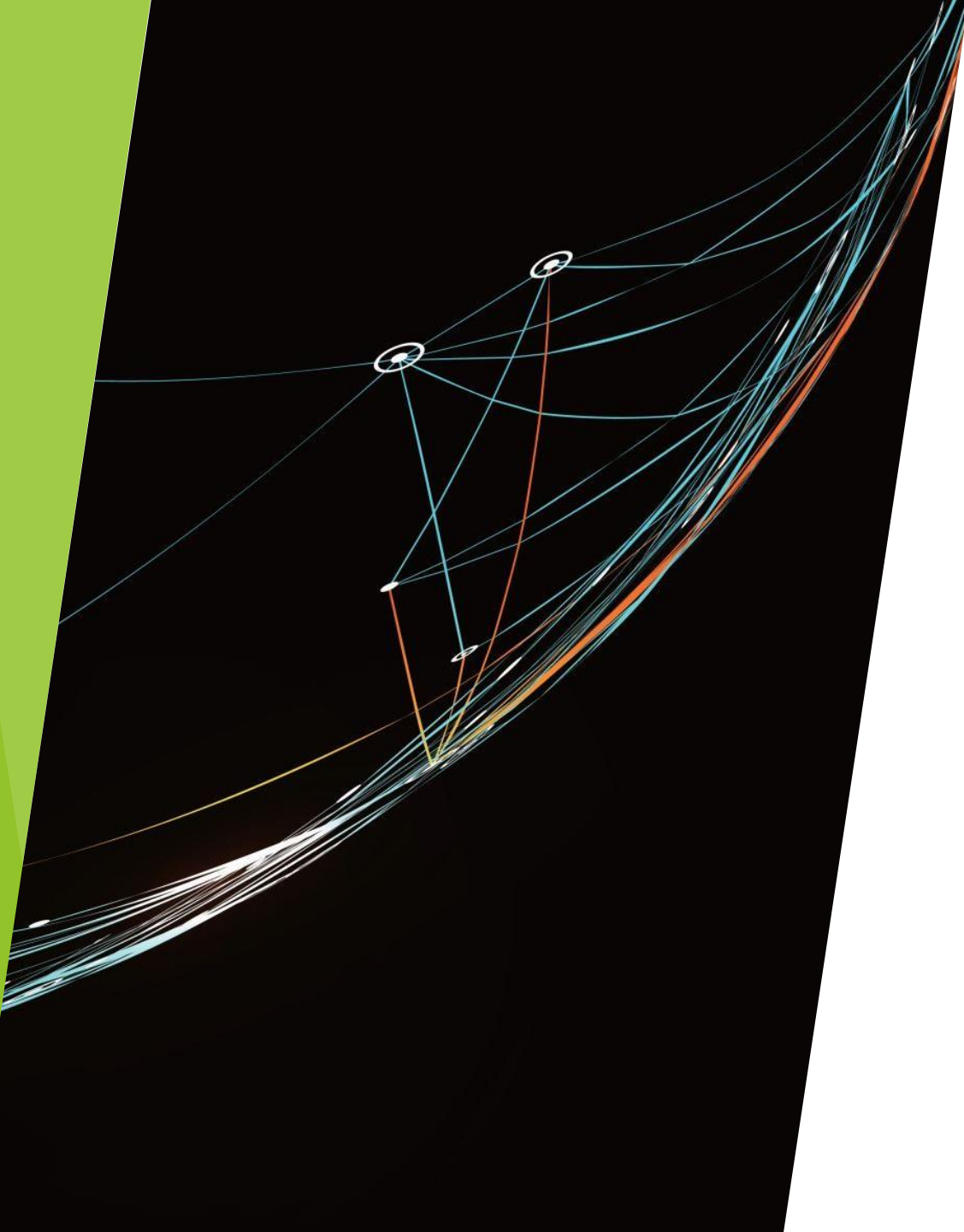
- **Function Coverage:** il metodo termina in quanto il test case passa
- **Statement Coverage:** ogni istruzione del metodo viene eseguita
- **Path Coverage:** l'unica struttura di controllo nel metodo è un while e vengono eseguite correttamente entrambe le vie
- **Condition Coverage:** l'unica istruzione booleana presente nel while è valutata sia quando è falsa che quando è vera

Testing

Testing a una funzionalità: Code Inspection

Code Inspection è una tecnica statica completamente manuale, ovvero senza esecuzione di codice. Analizziamo ad occhio in modo sistematico il codice della funzionalità e si cercano errori e difetti rispetto ad una lista di controlli standardizzati. Abbiamo quindi risposto alle seguenti domande:

Fault class	Inspection check
Data faults	Are all program variables initialized before their values are used? Have all constants been named? Should the lower bound of arrays be 0, 1, or something else? Should the upper bound of arrays be equal to the size of the array or size - 1? If character strings are used, is a delimiter explicitly assigned?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output?
Interface faults	Do all function and procedure calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception mngt faults	Have all possible error conditions been taken into account?



Refactoring

Refactoring

Pattern di refactoring sulla classe IntervalloOrario

Extract Method

Questo pattern è stato applicato due volte in maniera ricorsiva sullo stesso metodo: il metodo in questione è *intersecaAltroIntervalloOrario()*.

Ragioniamo su un solo frammento del codice e non sull'intero metodo:

```
// Inizio del secondo compreso nell'intervallo del primo
if (!altroIntervalloOrario.orarioIniziale.isBefore(this.orarioIniziale) && !altroIntervalloOrario.orarioIniziale.isAfter(this.orarioFinale)) {
    return true;
}

// Inizio del secondo compreso nell'intervallo del primo
if (this.isStartingBeforeOrTogetherWithTheBeginningOf(altroIntervalloOrario) && this.isEndingTogetherOrAfterWithTheBeginningOf(altroIntervalloOrario)) {
    return true;
}

return this.isIntersectedByTheBeginningOf(altroIntervalloOrario);
```


Refactoring

Introduce Assertion

Nell'unico costruttore (quello con tutti i parametri) della classe IntervalloOrario abbiamo le precondizioni che tutti i parametri attuali passati nell'invocazione del costruttore, ovvero *orarioIniziale* e *orarioFinale*, non debbano essere nulli e che *orarioIniziale* preceda temporalmente *orarioFinale*.

```
Ali +1 *  
public record IntervalloOrario(LocalTime orarioIniziale, LocalTime orarioFinale)  
    implements Serializable, DomainTypeToRender {  
  
    Aile  
    public IntervalloOrario {  
        assert !Objects.isNull(orarioIniziale);  
        assert !Objects.isNull(orarioFinale);  
        assert orarioIniziale.isBefore(orarioFinale);  
    }  
}
```

Definizione del record

Asserzioni per le precondizioni

NB: un ulteriore refactoring è stato quello di migrare IntervalloOrario da *Class* a *Record* per poter realizzare meglio il concetto di classe contenitrice di dati immutabili (inoltre supporta la pulizia del codice con la rimozione di boilerplate code).

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern, layered effect. The shapes are concentrated on the left and right sides, leaving a large white central area.

Fine