Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

Approfondimento Design Patterns: Mediator

Studente/i
Aleksandar Stojkovski

Relatore
-

Correlatore
-

Committente
Giancarlo Corti

Corso di laurea
Bachelor Ingegneria Informatica

Modulo
Ingegneria e sviluppo del software 2

Anno
2020

19.06.2020

STUDENTSUPSI

Indice

STUDENTSUPSI

Abstract

Il seguente documento è un approfondimento del design pattern Mediator, descritto e spiegato inizialmente nel libro "*Design Patterns: Elements of Reusable Object-Oriented Software*" da Gamma et al. L'obiettivo del documento è quello di capire ed analizzare il pattern proposto, evidenziando eventuali punti deboli e punti forti.

Il pattern verrà successivamente messo alla prova attraverso un'implementazione pratica in un progetto di laboratorio universitario. Il progetto tratta il gioco di carte del BlackJack e viene implementato in linguaggio Java sfruttando il framework JavaFX per quanto riguarda l'interfaccia grafica.

Verrà inoltre presentato un problema che necessiterà l'utilizzo del pattern in questione per essere risolto. Grazie a questo progetto sarà quindi possibile osservare come il pattern risolverà i problemi che si presenteranno nel corso dello sviluppo.

Introduzione

Il Capitolo 1 "Motivazione e contesto" descrive il contesto e il motivo che mi ha spinto a proporre questo lavoro. Nello specifico viene presentata una breve descrizione del progetto di laboratorio del modulo di "Ingegneria del Software 2".
Il Capitolo 2 "Problema" introduce un problema che ho dovuto affrontare durante lo sviluppo del progetto.
Il Capitolo 3 "Review del Design Pattern scelto" approfondisce il Pattern Mediator attraverso una selezione della letteratura disponibile.
Il Capitolo 4 "Implementazione e difesa" propone una possibile soluzione al problema presentato nel Capitolo 2, sfruttando il pattern scelto.
Il Capitolo 5 "Conclusioni" conclude l'analisi riassumendo i vantaggi e svantaggi della soluzione proposta.

STUDENTSUPSI

1. Motivazione e contesto

Questo lavoro viene sottomesso come parziale requisito per la certificazione del modulo di "Ingegneria e Sviluppo del Software 2" nel Bachelor di Ingegneria Informatica della Scuola Universitaria Professionale della Svizzera Italiana.

Siamo stati sottoposti ad un progetto a gruppi di tre persone, nel quale ci è stato chiesto di realizzare un software desktop a nostra scelta, sfruttando e utilizzando metodologie agili per lo sviluppo, curando dettagliatamente il codice e cercando di sfruttare i design patterns appresi durante l'anno accademico.

Il nostro progetto tratta il gioco di carte "BlackJack". Abbiamo scelto questo gioco poiché le regole risultano essere non troppo complicate; questo ci permette di focalizzarci sugli aspetti legati alla programmazione agile, piuttosto che alle logiche interne del gioco stesso.

Il progetto è stato realizzato utilizzando Java 11 come linguaggio di programmazione, JavaFX per quanto riguarda l'interfaccia grafica, TeamCity[1] come piattaforma per l'integrazione continua e Redmine[2] per la gestione e il tracking del progetto.

Di seguito (Fig. 1) è possibile osservare uno screenshot rappresentante l'ultima versione dell'applicazione:

Fig. 1 - Graphical User Interface dell'ultima versione dell'applicazione

[1] TeamCity è una soluzione multifunzione pre-integrata per l'integrazione continua e la distribuzione continua - Sorgente: jetbrains.com

[2] Redmine è un software gratuito e open source, per la pianificazione di progetti e per il tracciamento delle segnalazioni di bug tramite interfaccia web - Sorgente: wikipedia.org

2. Problema

I linguaggi di programmazione di tipo object oriented favoriscono la distribuzione delle responsabilità su più classi che interagiscono fra di loro. Questo principio prende il nome di "Single Responsability" e fa parte dei principi di programmazione SOLID[3].

Secondo Martin (2003), "A class should have only one reason to change" (p. 95). [1]

Ogni responsabilità è allo stesso tempo un possibile motivo di cambiamento. Quando i requisiti cambiano, essi si manifestano in cambiamenti di responsabilità, di conseguenza, se una classe ha più di una responsabilità, avrà più di un motivo per cambiare.
Un cambiamento ad una singola funzionalità potrebbe impattarne delle altre, cambiandone il comportamento e in alcuni casi causare problemi seri all'applicazione.

L'utilizzo del principio "Single Responsability" è infatti atto a risolvere questo tipo di problemi, cerando di assegnare ad ogni classe, solo e unicamente una singola funzionalità, aumentando riusabilità e coesione del codice.

A questo punto però si pone un problema; aver distribuito le responsabilità su così tante classi comporta tante interconnessioni tra di esse.

Secondo Gamma et al. (1995) "distribution of behavior among objects can result in an object structure with many connections between objects; in the worst case, every object ends up knowing about every other." (p. 273). [2]

Se da un lato la scomposizione di un sistema in tante classi aumenta la riusabilità, dall'altro lato aumenta le interconnessioni fra di loro. Molte interconnessioni fanno sì che il sistema sia strettamente accoppiato; un'oggetto riesce difficilmente a svolgere la sua funzione senza il supporto degli altri. Il risultato è un sistema molto complesso, difficile da mantenere e da estendere.

Siccome sin dall'inizio abbiamo suddiviso le responsabilità della nostra applicazione, ci siamo trovati presto ad avere tanti oggetti che necessitavano di comunicare fra di loro; le interazioni erano ben definite ma allo stesso tempo molto complesse.

STUDENTSUPSI

[3] In informatica, e in particolare in programmazione, l'acrostico SOLID si riferisce ai "primi cinque principi" dello sviluppo del software orientato agli oggetti descritti da Robert C. Martin in diverse pubblicazioni dei primi anni 2000 - Sorgente: wikipedia.org.

Il nostro errore più grande è stato quello di sviluppare le interazioni di queste classi seguendo e imitando la realtà, ovvero, nel caso del BlackJack, pensando a come funziona un vero casinò. Ci siamo presto accorti che non è stata una grande idea in quanto le interazioni erano diventate molto complesse.

Di seguito (Fig. 2) è possibile osservare un diagramma rappresentante le dipendenze delle principali classi del nostro backend. Questo diagramma è uno snapshot dello stato della nostra applicazione prima di aver utilizzato un design pattern appropriato.

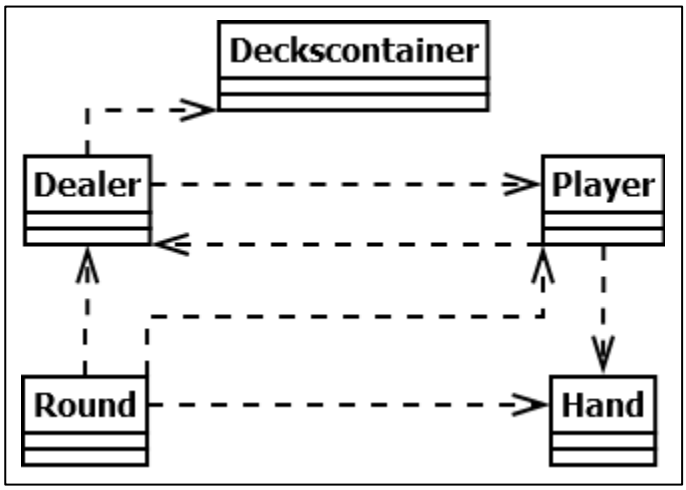

Fig. 2 – Diagramma rappresentate le dipendenze di alcune classi del backend

Il Player (giocatore):

- Parla con il Dealer per potergli chiedere una nuova carta oppure comunicargli che non ne vuole più ricevere.
- Parla con un'oggetto di tipo Hand che mantiene le carte della mano e la puntata relativa alla mano (ogni Player ha una sua Hand).

Il Dealer:

- Parla con i Players per poter dare loro le carte qual ora le richiedessero.
- Parla con un'oggetto di tipo DecksContainer che rappresenta il contenitore di carte, dal quale può mischiare e pescare le carte.

Infine, il Round mantiene lo stato relativo alla partita e quindi ha dipendenze su quasi tutte le altre classi.

Vista l'evidente complessità, sono andato alla ricerca di un pattern che potesse permettermi di disaccoppiare questi oggetti, mantenendo però le loro relazioni intatte. Il pattern che ho deciso di utilizzare per risolvere questo problema è il Mediator design pattern che descriverò nel prossimo capitolo.

STUDENTSUPSI

3. Review del design pattern scelto

3.1. Introduzione

Il Mediator design pattern è uno dei noti pattern descritti e spiegati nel libro "*Design Patterns: Elements of Reusable Object-Oriented Software*" di E. Gamma, R. Helm, R. Johnson e J. Vlissides[4].
L'intento del pattern è quello di ridurre complessità e dipendenze tra oggetti strettamente accoppiati che comunicano direttamente fra di loro (Fig. 3).

Secondo Freeman et al. (2004) "Use the Mediator pattern to centralize complex communications and control between related objects." (p. 622). [3]

Questo viene ottenuto grazie ad una classe intermedia che si occupa di mediare le comunicazioni tra gli oggetti (Fig. 4). Conseguentemente tutte le comunicazioni avverranno attraverso il Mediator, favorendo così il disaccoppiamento, in quanto gli oggetti non dovranno più interagire tra di loro, ma avranno un unico punto di riferimento. In questo modo anche la riusabilità di questi oggetti aumenta.

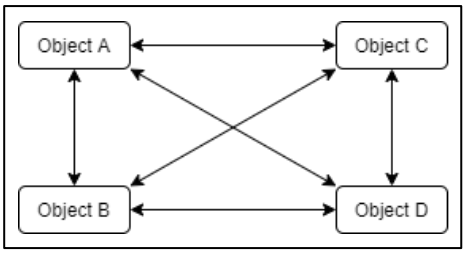

Fig. 3 – Comunicazione senza mediator

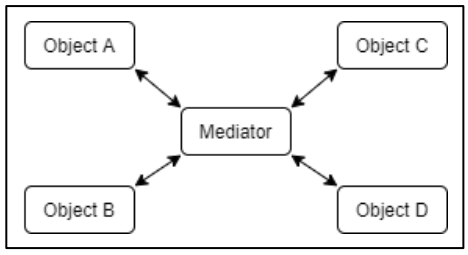

Fig. 4 – Comunicazione con mediator

3.2. Partecipanti

Prima di analizzare la struttura del pattern è necessario conoscere quali sono i componenti (partecipanti) coinvolti. Il pattern fa una netta distinzione tra classi "Colleague" e la classe "Mediator". Le prime sono tutte quelle classi che necessitano di comunicare fra di loro e che quindi utilizzeranno il mediator. Mentre il Mediator sarà l'interfaccia che i "Colleagues" utilizzeranno per comunicare fra di loro.
Più nel dettaglio abbiamo che:

Mediator:
- Definisce l'interfaccia utilizzata per la comunicazione tra oggetti di tipo Colleague.

ConcreteMediator:
- Implementa l'interfaccia Mediator e contiene la logica per la comunicazione tra oggetti Colleague.
- Conosce e mantiene le referenze agli oggetti di tipo Colleague.

Classi Colleague:
- Ogni oggetto di tipo Colleague conosce il suo Mediator.
- Le comunicazioni tra oggetti di tipo Colleague avvengono solo e unicamente mediante Mediator.

[4] Spesso referenziati come Gang of Four (GoF).

3.3. Struttura

La figura sottostante (Fig. 5) rappresenta la struttura base del pattern. Dalla figura è possibile notare che non esiste nessun tipo di collegamento diretto tra ConcreteColleague1 e ConcreteColleague2; siccome entrambe le classi implementano la classe astratta Colleague, ereditano la referenza all'oggetto mediator, che permette loro di comunicare, senza dipendere l'una dall'altra direttamente.

Ci sono diversi modi per far si che le classi Colleague ricevano la referenza al mediatore; il primo, come mostrato in figura, consiste nell'inserire la referenza del mediatore in una classe astratta. Un altro modo invece potrebbe essere quello di passare la referenza attraverso il costruttore dei colleague.

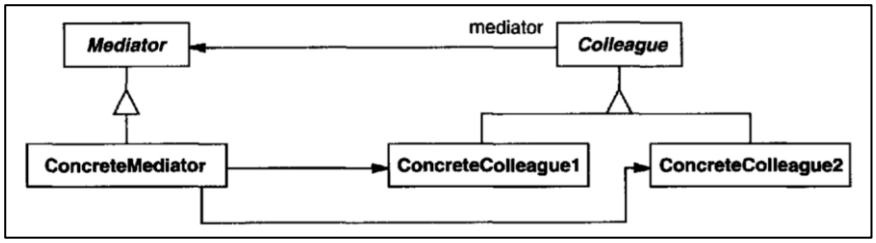

Fig. 5 - Struttura base del pattern Mediator

È possibile osservare (Fig. 6) che la comunicazione con il mediator può essere sia bidirezionale che unidirezionale; un colleague di tipo client, ad esempio, comunicherà con il Mediator, ma non necessariamente avverrà il contrario.

Questa è anche la principale caratteristica che differenzia un Mediator da un Façade. In caso di Façade infatti, il sottosistema è totalmente ignaro dell'esistenza della classe "facciata", di conseguenza la comunicazione con essa è unidirezionali, mentre nel caso del pattern Mediator, il sottosistema conosce e può comunicare con il Mediator.

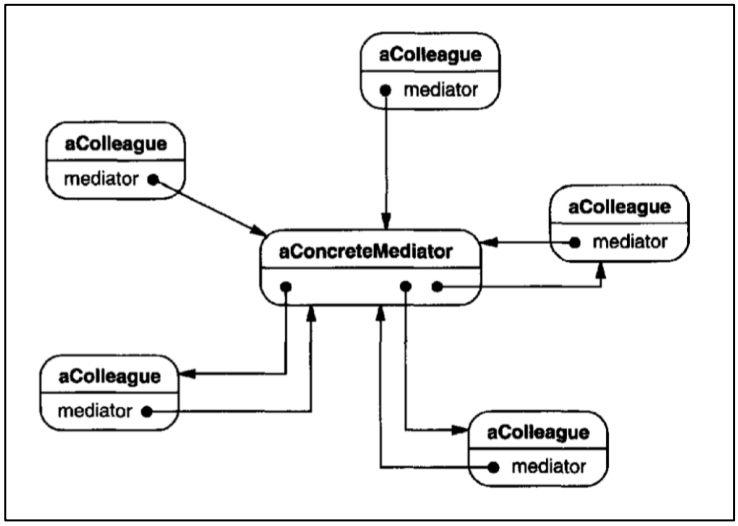

Fig. 6 - Diagramma delle comunicazioni del pattern

STUDENTSUPSI

3.4. Punti di forza

Il pattern presenta diversi punti di forza che possono essere riassunti come segue:

- **Promuove il disaccoppiamento**: le classi supportate dal Mediator comunicano tra di loro in modo indiretto, di conseguenza non dipendono l'una dall'altra.

- **Controllo centralizzato**: la complessità delle interazioni viene incapsulata nel mediator, questo consente una visione complessiva del sistema ed una gestione più efficiente delle modifiche.

- **Limita le sottoclassi**: l'aumento della riusabilità delle classi supportate dal Mediator risulta in una diminuzione di eventuali sottoclassi che altrimenti sarebbero necessarie.

3.5. Punti deboli

Pur avendo molti aspetti positivi, il pattern presenta pure alcuni aspetti negativi. Questi ultimi vanno gestiti per cercare di limitare il più possibile gli effetti collaterali che potrebbero insorgere. I punti deboli si possono quindi riassumere in:

- **Complessità**: siccome la classe Mediator racchiude tutte le interazioni, la sua complessità può diventare importante; per questo motivo è necessario pensare ad un design appropriato.

- **Single Point of Failure (SPOF):** il ruolo centrale della classe Mediator implica che nel caso di un malfunzionamento di quest'ultima, tutto il sistema cede, isolando le classi supportate dal mediatore.

STUDENTSUPSI

4. Implementazione e difesa

4.1. Struttura base

In questo capitolo descriverò la vera implementazione del pattern all'interno del nostro progetto, il quale ci permetterà di risolvere i problemi descritti nel capitolo 2 di questo documento.
Innanzitutto, è importante individuare quali sono i partecipanti al pattern, per questo motivo ho voluto fare un mapping (Fig. 7) tra quello che è il pattern in generale, ed il pattern applicato al nostro progetto.

Pattern in generale		Pattern applicato al nostro progetto
Mediator	→	RoundHandler
ConcreteMediator	→	RoundMediator
Colleagues	→	Dealer, Player, Hand, DecksContainer, GameModel

Fig. 7 – Mapping tra pattern in generale e pattern applicato al nostro progetto

L'interfaccia RoundHandler (Fig. 8) rappresenta il contratto implementato dalla classe concreta del mediatore. I colleagues potranno utilizzare questo contratto per comunicare fra di loro qual ora fosse necessario. L'implementazione di questa interfaccia è molto semplice in quanto contiene solamente i prototipi dei metodi. Nel nostro caso, il pattern è stato implementato in concomitanza con il pattern State; di conseguenza all'interno di questa interfaccia vi sono anche i metodi per la gestione dello stato dell'applicazione (setState(), getState() e goNextState()).

```
public interface RoundHandler {

    void setState(RoundState state);
    RoundState getState();
    void goNextState();
    void startRound();
    void exitRound();
    void nextRound();

    ...

    void playerBet(int amount);
    void playerConfirmBet();
    boolean isBetConfirmed();
    void openRound();
    void playerHit();

}
```

Fig. 8 – Frammento dell'interfaccia RoundHandler

Il grado di complessità del sistema determinerà il numero di metodi da definire all'interno dell'interfaccia. Alcune implementazioni di questo pattern definiscono un unico metodo, mentre altre implementazioni sono basate su più metodi.

STUDENTSUPSI

L'implementazione dei prototipi dell'interfaccia RoundHandler viene appunto fatta all'interno della classe concreta RoundMediator. Questa classe incapsulerà tutta la logica di comunicazione degli oggetti colleague, di conseguenza è possibile notare (Fig. 9) che avrà dipendenze su quasi tutte le altre classi coinvolte.
Il fatto che questa classe abbia molte referenze alle classi colleague è normale, l'importante è che le classi colleague non abbiano referenze tra di loro.

```
public class RoundMediator implements RoundHandler {

    private final Map<Player,Hand> playerHandMap = new HashMap<>();
    private final List<Player> playersOnly = new ArrayList<>();
    private final DecksContainer decksContainer;
    private final GameModel gameModel;
    private final Dealer dealer;
    private RoundState state;

    ...

}
```

Fig. 9 – Frammento della classe RounMediator, rappresentante le multiple dipendenze che la classe contiene

4.2. Refactoring classi esistenti

Implementata la struttura base del pattern, sarà ora necessario iniziare a rifattorizzare tutte le classi colleague, assicurandosi che queste ultime non si referenzino tra di loro, ma bensì, se necessario, utilizzino il mediator. Siccome le classi coinvolte nella rifattorizzazione sono molte, ne descriverò solamente due che ho trovato particolarmente interessanti (Player e Dealer).

Refactoring di Player

La prima classe che ho deciso di rifattorizzare è la classe Player. Il problema di questa classe (Fig. 10) è che ha una dipendenza diretta sulla classe Hand; questo significa che un Player (giocatore) ha sempre una mano associata. Quando il Dealer vuole dare una carta ad un giocatore, chiama il metodo addCard() sul giocatore in questione. Lo stesso avviene quando il Dealer si deve riprendere le carte attraverso il metodo discardCards().

```
public class Player {

    protected final Hand hand;

    public void addCard (Card newCard){
        hand.addCard(newCard);
    }

    ...

    public void discardCards() {
        hand.discardCards();
   }

}
```

Fig. 10 – Frammento della classe Player rappresentante l'accoppiamento tra Player e Hand

STUDENTSUPSI

Se in un futuro si volesse estendere l'applicazione in modo da poter giocare ad altri giochi da casinò (ad esempio la roulette), associare una mano ad ogni player sarebbe sbagliato in quanto non tutti i giochi fanno uso di carte, di conseguenza, non tutti i giocatori necessitano di avere una "Hand". Il giocatore inoltre potrebbe sedersi al tavolo da blackjack come spettatore senza realmente giocare, e anche in questo caso non necessiterebbe di una "Hand".

Per questo motivo ho deciso di disaccoppiare le due classi (Player e Hand) utilizzando il mediatore.
Come prima passo, ho eliminato tutte le referenze a Hand dalla classe Player; non sarà più il player a gestire la sua mano ma quest'ultima verrà gestita all'interno del mediatore.

Il mediatore invece si occuperà di gestire le mani di tutti i giocatori; questo viene fatto (Fig. 11) attraverso ad una HashMap<Player, Hand> contenente la mano associata ad ogni player. La mappa viene popolata nel costruttore del mediatore, creando tutti giocatori e le rispettive mani (Fig. 10).

Quando un giocatore, attraverso il frontend, premerà il bottone per richiedere una nuova carta, verrà chiamato il metodo hit() sul mediatore, che cercherà all'interno della mappa la mano del giocatore in questione e, una volta trovata, vi inserirà una nuova carta. Attraverso il pattern Observer la view verrà notificata del cambiamento e provvederà e mostrare la carta appena inserita.

```
public class RoundMediator implements RoundHandler {

  ...

  private final Map<Player,Hand> playerHandMap = new HashMap<>();

  ...

  public RoundMediator(GameModel gameModel, String pNick, String dNick){

    Player humanPlayer = new Player(pNick);
    Dealer dealer = new Dealer(dNick);

    playerHandMap.put(humanPlayer, new Hand());
    playerHandMap.put(dealer, new Hand());

    ...
  }

  public void hit(Player player) {

    Card card = dealer.giveCard();
    Hand hand = playerHandMap.get(player);
    hand.addCard(card);

    if(player instanceof Dealer){
        gameModel.firePropertyChange(new DealerHandUpdateEvent(this, hand, state));
    }else {
        gameModel.firePropertyChange(new PlayerHandUpdateEvent(this, hand));
    }

  }

  ...

}
```

Fig. 11 – Frammento della classe RoundMediator dopo la rifattorizzazione

STUDENTSUPSI

Di seguito (Fig. 12 e 13) è possibile vedere il diagramma di queste tre classi prima e dopo l'applicazione del Mediator pattern. Player e Hand sono ora completamente disaccoppiate, questo significa che la classe Player può essere riutilizzata potenzialmente anche per altri giochi, non necessariamente di carte. Il Player sarà associato ad una Hand solo e unicamente quando inizierà a giocare a BlackJack, questa associazione viene fatta dal Mediator.

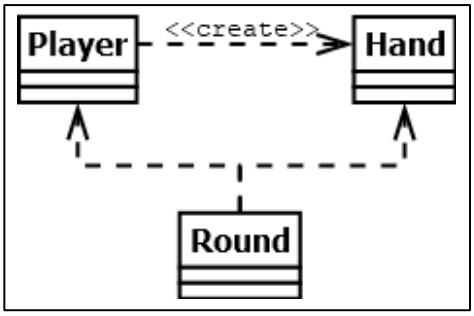

Fig. 12 – Diagramma Pre-Refactoring di Player e Hand

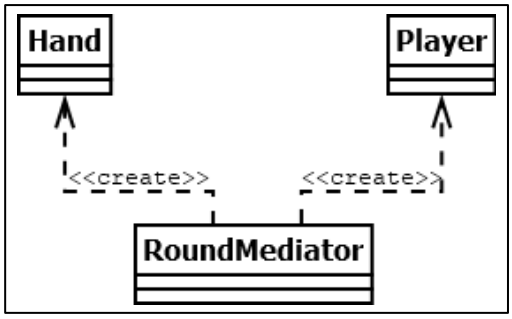

Fig. 13 – Diagramma Post-Refactoring di Player e Hand

Refactoring di Dealer

Un problema simile a quello di Player è presente anche all'interno della classe Dealer. Avendo pensato la logica delle classi ispirandosi alla realtà, il Dealer quando deve distribuire le carte, le prende da un oggetto di tipo DecksContainer, che rappresenta il contenitore di carte (Fig. 14). Questo significa che tutti i Dealer hanno "in pancia" un oggetto di tipo DecksContainer; non esiste un Dealer che non ce l'ha.

- Cosa accade se si vuole riutilizzare il Dealer per un altro gioco che non prevede l'utilizzo di carte?
- Cosa accade se vi sono più dealer che si danno il cambio?

Queste sono state le domande che mi hanno fatto riflettere. Ho quindi capito che avere un accoppiamento così forte tra i due oggetti potrebbe risultare problematico in caso di sviluppi futuri dell'applicazione.

```
public class Dealer extends VirtualPlayer {

    private final DecksContainer decksContainer;

    public Dealer(DecksContainer decksContainer) {
        super("Dealer");
        this.ai = new DealerAI(this);
        this.decksContainer = decksContainer;
        this.decksContainer.shuffle();
    }

    public Card giveCard(){
        return decksContainer.getCard();
    }

}
```

Fig. 14 – Frammento della classe Dealer, rappresentante l'accoppiamento tra Dealer e DecksContainer

STUDENTSUPSI

Il primo passo è quindi stato quello di spostare la referenza a DecksContainer dalla classe Dealer alla classe RoundMediator. Il metodo giveCard() non servirà più in quanto non sarà più il dealer a dare le carte ma sarà direttamente il mediatore.

In questo modo la classe Dealer si riduce solamente ad un costruttore (Fig. 15).

```
public class Dealer extends VirtualPlayer {

    public Dealer() {
        super("Dealer");
        this.ai = new DealerAI(this);
    }

}
```

Fig. 15 – Classe Dealer dopo il disaccoppiamento

Mentre alla classe RoundMediator si aggiunge la referenza a DecksContainer. Il metodo hit() non chiederà più una carta al Dealer (come visto in Fig. 11) ma bensì la chiederà direttamente al contenitore di carte (Fig. 16).

```
public class RoundMediator implements RoundHandler {

  private DecksContainer decksContainer;

  ...

  public RoundMediator(GameModel gameModel, String pNick, String dNick){

    try {
        decksContainer = new DecksContainer.Builder().numberOfDecks(3).build();
    } catch (InvalidDecksContainerSizeException e) {
        e.printStackTrace();
    }

    ...

  }

  public void hit(Player player) {

    Card card = decksContainer.getCard();

    ...

  }

}
```

Fig. 16 – Classe RoundMediator dopo il refactoring

STUDENTSUPSI

4.3. Risultato dell'implementazione

Ho applicato lo stesso principio a tutte le altre classi del backend cercando di rimuovere tutte le dipendenze interne utilizzando il mediator. Di seguito ho fatto generare all'IDE IntelliJ Idea due diagrammi; uno prima del refactoring e uno dopo. Dalla prima figura (Fig. 17) è possibile notare come, prima di applicare il pattern, eravamo in una situazione in cui quasi tutte le classi dipendevano l'una dall'altra, causando un forte accoppiamento e una bassa coesione del codice.
Dopo aver applicato il pattern invece (Fig. 18), l'unica dipendenza che queste classi si ritrovano ad avere è quella del Mediator.

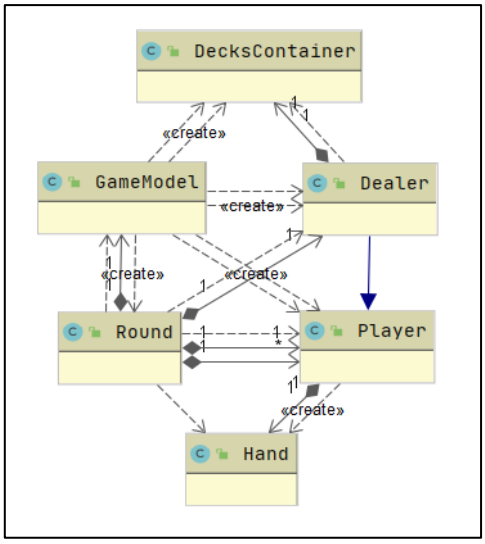

Fig. 17 - Diagramma prima l'utilizzo di Mediator

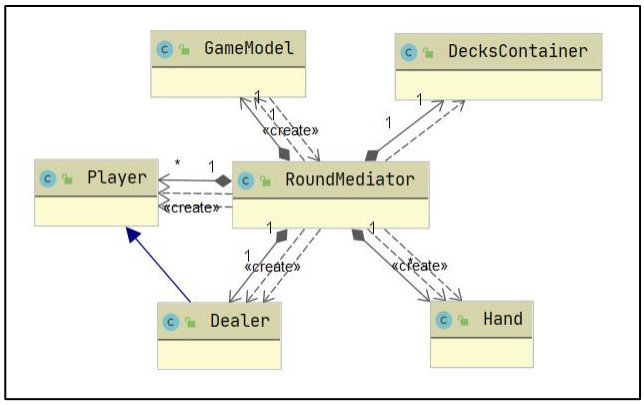

Fig. 18 – Diagramma dopo l'utilizzo di Mediator

STUDENTSUPSI

5. Conclusioni

Il Mediator design pattern è stato in grado di risolvere il nostro problema, questo non significa però che se ha risolto il nostro, sarà in grado di risolverli tutti.
Come ogni design pattern, presenta i suoi pregi e i suoi difetti, per questo motivo è necessario effettuare un'attenta analisi per capire se l'implementazione è veramente necessaria e può essere d'aiuto nel caso specifico.

Il pattern presenta un'iniziale overhead dovuto all'implementazione di tutte le componenti (interfacce e classi) ma una volta implementate, la sua gestione risulta essere molto semplice.
È tuttavia necessario pensare ad una buona struttura per quanto riguarda la classe del mediatore, in quando, siccome conterrà tutta la logica di comunicazione delle classi colleague, la sua complessità può diventare importante nel tempo, oscurando i pregi del pattern ed evidenziando tutti i suoi difetti.

In questo progetto abbiamo messo in pratica molti concetti imparati durante il corso e allo stesso tempo ne abbiamo appresi di nuovi. Ci siamo ritrovati a dover gestire il progetto sia dal punto di vista manageriale affrontando quella che è la gestione dei task, dei problemi e del tempo, e sia dal punto di vista tecnico cercando di prendere decisioni adeguate per quanto riguarda i pattern e le librerie da utilizzare all'interno del progetto.

In generale siamo molto soddisfatti del lavoro che abbiamo svolto, poiché ci ha permesso di ampliare le nostre conoscenze professionali. A questo proposito vogliamo ringraziare Giancarlo Corti per i consigli e le informazioni fornite durante gli incontri in queste settimane, i quali ci hanno permesso di portare a termine questo progetto con successo.

6. Bibliografia

[1]
Titolo: Agile Software D'evelopment, Principles, Patterns, and Practices
Autori: Robert C. Martin, Jan M. Rabaey, Anantha P. Chandrakasan, Borivoje Nikolic
Pubblicazione: 2003
ISBN: 978-0135974445

[2]
Titolo: Design Patterns: Elements of Reusable Object-Oriented Software
Autori: Erich Gamma, John Vlissides, Richard Helm, Ralph Johnson
Pubblicazione: 1995
ISBN: 978-0201633610

[3]
Titolo: Head First Design Patterns
Autori: Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra
Pubblicazione: 2004
ISBN: 978-0596007126