



UNIVERSITÀ DEGLI STUDI
DI MILANO

Ingegneria del software

Progettazione

Lab 04

Si vuole avere una classe `PokerHand` che permetta di gestire un gruppo di carte (generalmente 5, parametro del costruttore) provenienti da un mazzo specifico (indicato in fase di costruzione degli oggetti `PokerHand`)

Test

```
@Test  
public void newPokerHandTest() {  
    PokerHand pokerHand = new PokerHand( deck, 5 );  
    assertThat(pokerHand).isEqualTo(???);  
}
```

PROBLEMA: Testing con encapsulation

1. Come facciamo se abbiamo bisogno che un oggetto sia in un determinato stato per poter fare il testing?
2. Come facciamo se lo stato su cui dobbiamo fare asserzioni è privato e non accessibile?

SOLUZIONI:

- Controlliamo che non ci sia già o debba essere definito un metodo da cui controllarlo indirettamente (ad esempio una `toString`)
- Creiamo dentro al test un metodo (helper) in grado di ricavare tale informazione o una sua derivata (anche se in maniera inefficiente)
- Aggiungiamo un metodo in produzione per avere accesso all'informazione
- Allarghiamo scope di visibilità per motivi di testing
- Accediamo usando la reflection o introspezione (o i metodi predefiniti di assertj che la usano)



Studio anche le classi esistenti

Classe Card

```
/**  
 * An immutable description of a playing card. This abstraction  
 * is designed to be independent of game logic, so it does  
 * not provide any service that relies on the knowledge  
 * of the rules of any particular game.  
 *  
 * This class implements the Flyweight design pattern:  
 * there can only ever be one instance of a card that  
 * represents a specific real-world playing card (such as ace  
 * of spades). In the absence of serialization and reflection,  
 * this ensures that the behavior of the == operator is identical  
 * to that of the equals method when two card arguments are  
 * provided.  
 */
```

Immutable , flyweight design pattern

FLYWEIGHT pattern

Serve a gestire una collezione di oggetti immutabili assicurandone la unicità.

Quando le istanze equivalenti sono fortemente condivise all'interno del programma diventano auspicabili sia immutabilità che unicità. Ad esempio le nostre Card.

È simile al SINGLETON ma a differenza di quello devo contenere più istanze e spesso non è definibile a priori quante sono.

```
public class Card {  
    private static final Card[][][] CARDS = new Card[Suit.values().length][Rank.values().length];  
    static {  
        for( Suit suit : Suit.values() )  
            for( Rank rank : Rank.values() )  
                CARDS[suit.ordinal()][rank.ordinal()] = new Card(rank, suit);  
    }  
    public static Card get(Rank pRank, Suit pSuit) {  
        return CARDS[pSuit.ordinal()][pRank.ordinal()];  
    }  
}
```

Creo Metodi Helper per punto 1

```
private static Deck createDeckWithCards(Card[] cards) {  
    Deck deck = createEmptyDeck();  
    for (Card card : cards) deck.push(card);  
    return deck;  
}  
  
private static Deck createEmptyDeck() {  
    Deck deck = new Deck();  
    for (int i = 0; i < 52; i++) deck.draw();  
    return deck;  
}  
  
final static private Card[] THREE_OF_A_KIND_CARDS = {  
    Card.get(Rank.ACE, Suit.HEARTS), Card.get(Rank.THREE, Suit.HEARTS),  
    Card.get(Rank.FIVE, Suit.HEARTS), Card.get(Rank.ACE, Suit.CLUBS),  
    Card.get(Rank.ACE, Suit.DIAMONDS)  
};  
  
@Test  
public void newPokerHandTest() {  
    PokerHand pokerHand = createDeckWithCards(THREE_OF_A_KIND_CARDS);  
    assertThat(pokerHand).isEqualTo(???);  
}
```

punto 2: alcune possibilità

- modifco visibilità campo togliendo `private`

```
assertThat(pokerHand.cards).containsExactly(THREE_OF_A_KIND_CARDS);
```

- uso metodi *pericolosi* che sfruttano *reflection*

```
assertThat(pokerHand).extracting("cards", as(LIST)).containsExactly(THREE_OF_A_KIND_CARDS);
```

- aggiungo metodo `toString`

```
@Override public String toString() { return cards.toString(); }
```

```
assertThat(pokerHand.toString())
    .isEqualTo("[ACE of HEARTS, THREE of HEARTS, FIVE of HEARTS, ACE of CLUBS, ACE of DIAMONDS]");
```

- implemento `Iterable` come richiesto dopo

```
@Override public Iterator<Card> iterator() { return cards.iterator(); }
```

```
assertThat(pokerHand).contains(THREE_OF_A_KIND_CARDS);
```

Faccio passare il test

```
final private List<Card> cards = new ArrayList<>();  
  
public PokerHand(Deck deck, int numCards) {  
    for (int i = 0; i < numCards; i++) {  
        cards.add(0, deck.draw());  
    }  
}
```

- **Tutto giusto?** Pensavo a possibile Reference Escaping... ma in effetti difficile che si presentasse in questo caso

Proseguiamo: Iterable

Gli oggetti PokerHand espongono il gruppo di carte di cui sono composti solo tramite un iteratore
(Utilizzare il pattern Iterator, sfruttando l'interfaccia Iterable della libreria standard)

Probabilmente già risolto, comunque:

```
public class PokerHand implements Iterable<Card> {  
    @Override  
    public Iterator<Card> iterator() {  
        return cards.iterator();  
    }  
    ...  
}
```

che avevamo testato con:

oppure più basico

```
assertThat(pokerHand).containsExactly(HIGH_RANK_CARDS);  
  
Iterator<Card> iterator = pokerHand.iterator();  
for (Card card : HIGH_RANK_CARDS)  
    assertThat(iterator.next()).isEqualTo(card);
```



Proseguiamo: getRank

Gli oggetti PokerHand forniscono un metodo `getRank()` per determinare il punteggio di una mano
[...] Non è necessario realizzare tutti i valutatori ma almeno tre a vostra scelta.

- copio l'enumerativo `HandRank` in una classe a parte
- definisco primo test

```
@Test
void getRankHighRankTest() {
    Deck deck = createDeckWithCards(HIGH_RANK_CARDS);
    PokerHand pokerHand = new PokerHand(deck, 5);
    assertThat(pokerHand.getRank()).isEqualTo(HandRank.HIGH_CARD);
}
```

- e funzione

```
public HandRank getRank() { return HandRank.HIGH_CARD; }
```

Come vado avanti?

L'implementazione del metodo `getRank()` rischia di cadere nell'anti-pattern "Switch Statement". Per evitarlo, è possibile organizzare la valutazione del punteggio secondo un pattern "Chain-of-responsibility"

Avendo intuito (o letto) che potrei usare il pattern **CHAIN OF RESPONSABILITY**, ho due problemi da risolvere contemporaneamente:

- capire come riconoscere cosa ho in mano (coppia, doppia-coppia, etc...)
- capire come funziona il pattern (visto che era la prima volta)
- Due possibilità
 - tralascio per il momento il pattern e risolvo il punto 1 usando if e switch per un paio di iterazioni prima di pormi il problema di rifattorizzare
 - prendo l'incapsulamento che mi fornisce il pattern (che magari conosco già un po') e testo un nodo della catena alla volta. Nodi che poi legherò a formare le regole.

Come è fatto un nodo ?

```
interface ChainedHandEvaluator {  
    HandRank handEvaluator(...);  
}  
  
class TrisEvaluator implements ChainedHandEvaluator {  
}
```

Come lo testo?

```
@Test  
void chainedTrisEvaluatorSuccessTest() {  
    Deck deck = createDeckWithCards(THREE_OF_A_KIND_CARDS);  
    PokerHand pokerHand = new PokerHand(deck, 5);  
  
    ChainedHandEvaluator evaluator = new TrisEvaluator(null); // non ho un next da dargli  
    assertThat(evaluator.handEvaluator(pokerHand)).isEqualTo(HandRank.THREE_OF_A_KIND);  
}
```



Faccio passare il test

```
class TrisEvaluator implements ChainedHandEvaluator {  
    final private ChainedHandEvaluator next;  
  
    public TrisEvaluator(ChainedHandEvaluator next) {  
        this.next = next;  
    }  
  
    public HandRank handEvaluator(PokerHand hand) {  
        if (isATris(hand)) return HandRank.THREE_OF_A_KIND;  
        else return next.handEvaluator(hand);  
    }  
  
    private boolean isATris(PokerHand hand) {  
        Map<Rank, Integer> occ = new EnumMap<>(Rank.class);  
  
        for( Card card : hand) {  
            int num = occ.getOrDefault(card.getRank(), 0);  
            if (num == 2) return true;  
            else occ.put(card.getRank(), num + 1);  
        }  
        return false;  
    }  
}
```



Faccio test di triangolazione

cioè controllo che se gli passo una mano che non è un tris ...

```
@Test  
void chainedTrisEvaluatorFailTest() {  
    Deck deck = createDeckWithCards(HIGH_RANK_CARDS);  
    PokerHand pokerHand = new PokerHand(deck, 5);  
  
    ChainedHandEvaluator evaluator = new TrisEvaluator(null);  
    assertThatThrownBy(() -> evaluator.handEvaluator(pokerHand))  
        .isInstanceOf(NullPointerException.class);  
}
```

ma è pericoloso: NullPointerException potrebbe essere stato sollevato da altro.

Test Double

Quando testiamo un oggetto (SUT system under test), la sua esecuzione può dipendere da altri componenti del sistemi da lui richiamati (DOC dependent-on component)

Test Double (controfigura per il testing) è un termine generico per un qualunque tipo di oggetto che viene messo al posto di *DOC* reale al fine di permettere il testing del *SUT* quando il *DOC*:

- non esiste ancora
 - fornirebbe dei dati non deterministici/non prevedibili: il tempo corrente, la temperatura corrente
 - può presentare delle situazioni difficilmente riproducibili (errori di trasmissione, esaurimento della memoria)
 - è lento
- o *semplicemente* perché si vuole testare il SUT senza correre il rischio che il DOC introduca errori

Nullability

Ad una variabile che indica un *riferimento* a un oggetto (quindi in Java **sempre** a parte i *tipi primitivi*), allora possiamo assegnare il valore speciale `null`, per dire che ... *non punta a nulla*.

```
Card card = null; // oppure Card.rank = null;
```

Il problema si manifesta quando proviamo a *dereferenziare* la variabile e non puntando a nulla riceviamo un errore (una `NullPointerException`)

- un parametro può essere `null` o posso assumere che punti a un dato "valido"?

Il "problema" è che `null` viene usato per indicare diverse cose:

- errore
- stato temporaneamente inconsistente
- valore assente

Un codice chiaro non dovrebbe far uso di `null` o almeno limitarlo

Senza valori "assenti"

```
public class Card {  
    private Rank rank; // Should never be null  
    private Suit suit; // Should never be null  
  
    public Card(Rank rank, Suit suit) {  
        this.rank = rank;  
        this.suit = suit;  
    }  
}
```

```
public Card(Rank rank, Suit suit) {  
    if (rank != null && suit != null) {  
        this.rank = rank;  
        this.suit = suit;  
    }  
}
```

```
public Card(Rank rank, Suit suit) {  
    assert rank != null && suit != null;  
    this.rank = rank;  
    this.suit = suit;  
}
```

SBAGLIATO!

Come possiamo "tradurre" in codice questi commenti?

```
public Card(Rank rank, Suit suit) {  
    if (rank == null || suit == null)  
        throw new IllegalArgumentException();  
    this.rank = rank;  
    this.suit = suit;  
}  
  
final @NotNull private Rank rank;  
final @NotNull private Suit suit;  
  
public Card(@NotNull Rank rank, @NotNull Suit suit) {  
    this.rank = rank;
```



Con valori "assenti"

Aggiungiamo la carta **Joker** che non ha né *Suit* né *Rank*

```
public class Card {  
    private Rank rank;  
    private Suit suit;  
    private boolean isJoker;  
    public boolean isJoker() { return isJoker; }  
    public Rank getRank() { return rank; }  
    public Suit getSuit() { return suit; }  
}
```

Cosa ritorno i *getter* se è un Joker?

- null
 - è quello che stiamo sconsigliando
- un valore qualsiasi (tanto controlleremo prima se `isJoker() == true`)
 - confusione e probabili errori di uso
- aggiungo valore enumerativo: `NONE` ... ma ci sarebbero 5 segni e 14 rank

NULLOBJECT pattern

Vogliamo creare un oggetto che corrisponda al concetto "nessun valore" o "valore neutro"

```
public interface CardSource {  
    Card draw();  
    boolean isEmpty();  
  
    public static CardSource NULL = new CardSource() {  
        public boolean isEmpty() { return true; }  
        public Card draw() {  
            assert !isEmpty();  
            return null;  
        }  
    };  
}
```

`CardSource.NULL` è un oggetto valido di un tipo anonimo che aderisce alla interfaccia `CardSource` ma che ha particolari implementazioni per i vari metodi

Serve ad evitare di dover trattare separatamente il caso `== null`, ma se proprio diventasse necessario si può sempre testare `== CardSource.NULL`

Optional Types

Optional is primarily intended for use as a method return type where there is a clear need to represent "no result," and where using null is likely to cause errors. A variable whose type is Optional should never itself be null; it should always point to an Optional instance.

Al posto del costruttore definisce tre metodi statici:

- `static <T> Optional<T> empty()` Returns an empty Optional instance.
- `static <T> Optional<T> of(T value)` Returns an Optional describing the given non-null value.
- `static <T> Optional<T> ofNullable(T value)` Returns an Optional describing the given value, if non-null, otherwise returns an empty Optional.

e poi fornisce metodi (tra cui ad esempio)

- `T get()` If a value is present, returns the value, otherwise throws NoSuchElementException
- `boolean isEmpty()` If a value is not present, returns true, otherwise false
- `boolean isPresent()` If a value is present, returns true, otherwise false
- `T orElse(T other)` If a value is present, returns the value, otherwise returns other

Completiamo la chain e usiamo getRank()

Facciamo *refactoring* per inserire pattern pur continuando a gestire solo **HIGH_RANK**.

```
public interface ChainedHandEvaluator {  
    @NotNull PokerHand.HandRank handEvaluator(@NotNull PokerHand pokerHand);  
    ChainedHandEvaluator HIGH_RANK = pokerHand -> PokerHand.HandRank.HIGH_CARD;  
}  
  
public @NotNull HandRank getRank() {  
    ChainedHandEvaluator rules = ChainedHandEvaluator.HIGH_RANK;  
    return rules.handEvaluator(this);  
}
```

Aggiungiamo caso di test

```
@Test  
void getRankHighRankTest() {  
    Deck deck = createDeckWithCards(THREE_OF_A_KIND_CARDS);  
    PokerHand pokerHand = new PokerHand(deck, 5);  
  
    assertThat(pokerHand.getRank()).isEqualTo(PokerHand.HandRank.THREE_OF_A_KIND);  
}
```

E lo facciamo passare

```
public HandRank getRank() {  
    ChainedHandEvaluator rules =  
    new FlushEvaluator(  
        new TrisEvaluator(  
            ChainedHandEvaluator.HIGH_RANK));  
  
    return rules.handEvaluator(this);  
}
```

