**LABORATORIO SWENG**

**GIT**

- fare il **fork** del repository

- per copiare il repository in locale usare **clone**

- creare un nuovo branche: **git flow init**

- creare una nuova feature: **git flow feature start <nome>**

- commit: - **git add**

- **git commit**

- chiudere una feature: **git flow feature finish <nome>**

- rilasciare una release: - **git flow release start <ver>**

- successivamente **git flow release finisch <ver>**

- push su repository online: - **git push origin --all**

- **git push origin master**

Se c’è un bug in una versione già pubblica del programma si crea un **hotfix** per sistemare il problema

**TEST**

con più test in input e uguale output

@ParameterizedTest

@ValueSource(strings = {"", " ", "\n"})

@NullSource

void nomeTest(String program) {

interpreter.input(program);

assertThat(interpreter.toString()).isEqualTo("<- Top");

}

Controllare input e output

@ParameterizedTest

@CsvSource( {“input1, output1”,

“input2, output2”,

“**’**input**\n**3**’**, output3”,})

void testNoInput(String program, String output) {

classe.input(program);

assertThat(classe.toString()).isEqualTo(output);

}

Controllare eccezione

@ParameterizedTest

@CsvSource( {“input1, output1”,

“inputErrato, output2” })

void testInvalidToken(String program, String error) {

assertThatThrownBy(

( ) -> interpreter.input(program) )

.islnstance0f(IllegalArgumentException. class)

.hasMessage(“errore”)

}

Inserire e iterare su una lista

@Test

void newPokerHandTest() {

PokerHand ph = new PokerHand(

**List.of**(

Card.get(Rank.ACE, Suit.CLUBS),

Card.get(Rank. TWO, Suit.CLUBS),

Card.get(Rank.ACE, Suit.DIAMONDS),

Card.get(Rank.JACK, Suit.HEARTS),

Card.get(Rank.QUEEN, Suit.HEARTS)

)

);

assertThat(**(Iterable<Card>)** ph).**containsExactlyInAnyOrder**(

Card.get(Rank.ACE, Suit.CLUBS),

Card.get(Rank.TWO, Suit.CLUBS),

card.get(Rank.ACE, Suit.DIAMONDS),

Card.get(Rank.JACK, Suit.HEARTS),

Card.get(Rank.QUEEN, Suit.HEARTS)

);

}

**libreria** **mockito**

MyClass variabile = **mock**(MyClass.**class**);

when(mockedObj.**methodname(args)**).**thenXXX**(values);

**- args**: values matchers argumentCaptor

**- thenXXX**: thenReturn thenThrows thenAnswer thenCallRealMethod

Quando si ha un metodo che ritorna void:

**doXXX**(values).when(mockedObj).methodname(args)

Per verificare la occorrenza di una chiamata con certi parametri:

**verify**(mockedclass, **howmany**).methodname(args)

**- howmany**: times(n) never atLeast(n) atMost(n)

verifyNoMoreInteractions(mockedClass)

Per verificare l'ordine delle occorrenze delle chiamate:

InOrder inO = InOrder(mock1, mock2)

InO.verify...

Possibile catturare un parametro per farci sopra asserzioni

ArgumentCaptor<Person> arg = ArgumentCaptor.forClass(Person.class);

verify(mock).doSomething(arg.capture());

AssertEquals(“John”, arg.getValue().getName());

@Test

public void testDummy() {

MyClass dummy = mock(MyClass.class);

List<MyClass> SUT = new ArrayList<MyClass> ();

SUT.add(dummy);

assertThat (SUT. size()).isEqualTo(1);

}

@Test   
public void testConStub() {

MyCIass stub = mock(MyCIass.class);

when(stub.getVaIue(0)).thenReturn(4);

when(stub.getVaIue(1)).thenReturn(7, 3);

MyList<int> SUT = new MyList<int>();

SUT.add(stub.getValue(0));

SUT.add(stub.getValue(1));

SUT.add(stub.getValue(1));

res = SUT.somma();

assertThat(res).isEqualTo(14);

}

@Test   
public void testConStub() {

MyCIass stub = mock(MyCIass.class);

when(mock.getVaIue(0)).thenReturn(4);

when(mock.getVaIue(1)).thenReturn(7, 3);

MyList<int> SUT = new MyList<int>();

res = SUT.somma(mock);

assertThat(res).isEqualTo(14);

InOrder io = inOrder(mock);

io.verify(mock).getVaIue(0);

io.verify(mock, times(2)).getVaIue(1);

}

**SEPARARE STRINGA (in automatico da spazi e \n)**

public void input (String program) {

try (Scanner sc = new Scanner(program)) {

whiIe (sc.hasNext()) {

String token = sc.next();

if (token.matches**( "-?[0-9]+")**) { // **regex**: espressione per delimitare il token

internalStack.push(lnteger.vaLueOf(token)) ;

}

}

}

}

**Eliminazione if**

**1. Map**  
 - nei casi di sequenze tipo:

if(x == Tipoconfronto){

operazione che non ritorna nulla e non chiede nulla (add su arraylist, push su uno staks, ecc)

}

Soluzione:

Public MyClass() {

**// creare una variabile di tipo Map (private final)**

Map< Tipoconfronto, Operation> operators = new HashMap<>();

interface Operation { **// creare l’interfaccia “operazione” con metodo op**  
 void op();  
 }

**// si aggiunge ogni operazione all’interno del costruttore**

public MyClass() {

Runnable.put( “+” , new Runnable { **//prima soluzione**

@Ovveride

Public void op(){

//mettiamo che cosa fa con il caso “+”

stack.push(stack.pop() + stack.pop());

}

});

Runnable.put(“2° caso” , new Runnable{

@Ovveride

Public void op(){ codice if }

});

operators.put("+", () -> stack.push(stack.pop() + stack.pop())); **//seconda soluzione**  
 operators.put("\*", () -> stack.push(stack.pop() \* stack.pop())); **//con utilizzo di lambda**  
 operators.put("dup", () -> stack.push( stack.peek()));  
 }

**//all’interno del metodo che deve decidere i vari casi si metterà**

if(runnable.conteinsKesy(token)) {

Runnable.get(token).op();

}

}

**2. sto cazzo-> bho**

**3. chain evaluator**

public enum HandRank { **// creare una classe enum con tutte le categorie**  
 HIGH\_CARD,  
 ONE\_PAIR,  
 TWO\_PAIR,  
 THREE\_OF\_A\_KIND,  
 STRAIGHT,  
 FLUSH,  
 FULL\_HOUSE,  
 FOUR\_OF\_A\_KIND,  
 STRAIGHT\_FLUSH  
}

**// creare un'interfaccia “catena” che abbia all’interno**

**// un metodo per valutare l’oggetto**

Public interface ChainedHandEvaluator {

ChainedHandEvaluator HIGHCARD = ph -> HandRank.HIGH\_CARD; **//condizione ultima (null Object)**

HandRank handEvaluator(PorkerHand ph);

}

public class OnepairEvaluator implements ChainedHandEva1uator { **// implemento la classe di una categoria**

private final ChainedHandEvaluator next; **// per controllare quello specifico caso**

public ChainedHandEvaIuator(@NotNull ChainedHandEvaluator next) {

this.next = next;

}

@Override

public HandRank handEvaIuator(PokerHand Ph) {

EnumSet<Rank> present = EnunSet.noneOf(Rank.class);

for (Card card : ph) {

if (present.contains(card.getRank()))

return HandRank.ONE\_PAIR;

else

present.add(card. getRank());

}

return next.handEvaluator(ph);

}

}

public class PokerHand { **// creo il metodo (all’interno della classe che deve sapere quale è il caso giusto)**

... **// per individuare quale percorso bisogna fare (segnare la catena effettiva)**

Static final private ChainedHandEvaluator **chain** =

new FlushEvaluator (

new OnePairEvaluator (

ChainedHandEvaluator.HIGHCARD ));

Public HandRank getRank() {

return chain.handEvaluator(this);

}

...

}

Implementazione:

- Per prima cosa è necessario una interfaccia catena con dentro il metodo necessarie a valutare

Ad esempio chiedi carta

È necessario quindi avere delle classi strategie varie (che alla fine rispettano il campo if else)

Che implementano l’interfaccia ed introducono le variabili necessaria a funzionare oltre ad una variabile

Strategia Next che servirà nel caso in cui la strategia attuale non sia quella che usiamo (ha fallito nel campo if) e servirà per andare nella prossima classe che implementa strategia

Dentro la classe potrebbe servire un costruttore nel caso avessimo bisogno di stato / variabili

O di un metodo per eseguire il corpo dell if  
 ad esempio:

abbbiamo bisogno di variabili che arrivano da altre classi

giocatore

strategia

nel costruttore chiedereme entrembe

this.giocatore=giocatore;

this.next=strategia;

così da avere la prossima strategia e il giocatore attuale (che passeremo alla prossima strategia nel caso in cui questa non vada bene)

qui dipendentemente da quel che dobbiamo fare potremmo avere solo il metodo dell interfaccia o altri

(penso per lo piu solo quello del interfaccia)

Ad esempio boolean chiediCarte()

//diciamo che chiede care se ne a meno di 3

//ha bisogno di giocatore perché giocatore ha un metodo sizemano()

If(giocatore.sizemano()<3){

return true;

}else{

return next.chiedicarte();

}

return false;

Quindi, supponiamo che se il giocatore ha meno di 3 carte, il risultato della valutazione sia "true". In questo scenario, potremmo implementare un metodo (chiaramente in una classe diversa da strategia)con la seguente logica: "if (strategia == true) pesca carta else rip".

Se il giocatore ha almeno tre carte, procederemo a valutare un'altra condizione definita attraverso un'altra classe, come già discusso in precedenza. Anche questa condizione restituirà un valore booleano, "true" o "false".

In questo contesto, non era necessario implementare un metodo separato, poiché la sua unica funzione è decidere se pescare o meno una carta. In alternativa, avremmo potuto assegnare un nome distintivo a ogni strategia, utilizzando un enum con i nomi delle strategie, e restituire questi valori invece di un booleano. Seguendo questa scelta, avremmo potuto gestire ogni situazione individualmente con un'istruzione "if" (semplificabile ulteriormente con una mappa).

È importante notare che in questa situazione è opportuno introdurre il "null object pattern". Questo consiste nell'utilizzare un oggetto che può essere inserito al posto del caso null e che si comporta in modo conforme alle tue esigenze.

Ad esempio, anziché restituire "null" al posto di "next" nell'ultima strategia, avremmo potuto creare un oggetto che ritornasse sempre "false" al posto di "true". Questo oggetto può essere utilizzato come un terminatore di catena, fornendo una gestione coerente anche in situazioni in cui normalmente ci sarebbe un valore nullo.

**ES code**

public class MediaStrategy implements Strategia {

final private Strategia next;

private final Sfidante sfidante;

public MediaStrategy(Sfidante sfidante, Strategia strategia) {

this.sfidante = sfidante;

next = strategia;

}

@Override

public boolean chiediCarta() {

if(sfidante.getPunti()<=16) {

return true;

} else {

next.chiediCarta();

}

return false;

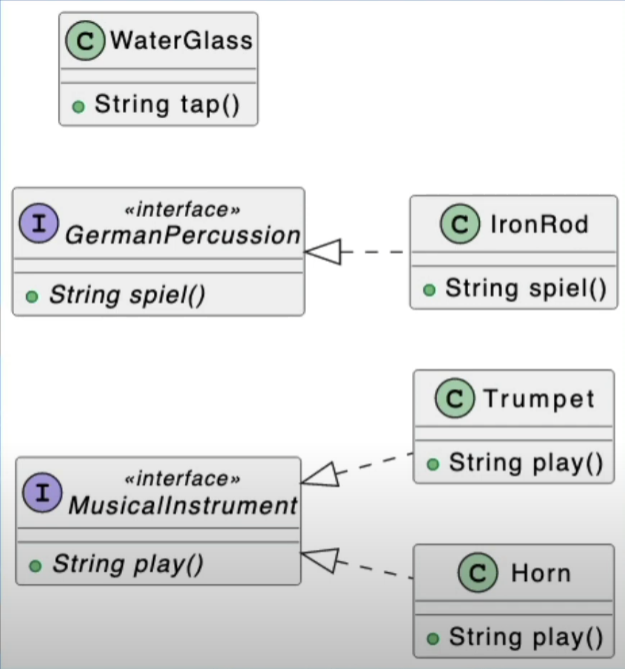
}

}

**PATTERN**

**Adapter Compose Decoretor pattern**

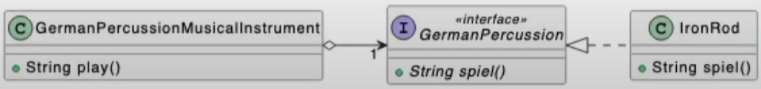
Introduciamo ora altri pattern faremo gli esempi su una struttura generale di questo tipo



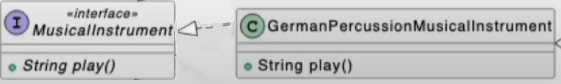
Allora partiamo dal Adapter Pattern ci sono 2 modi di implementarlo in base a se vuoi o meno che sia ancora utilizzabile la vecchia interfaccia:

Facciamo caso che noi vogliamo introdurlo per gli stumenti germanici e per il bicchiere d’acqua

Nel primo caso potremmo volere tenere la vecchia interfaccia GermanPercussion quindi

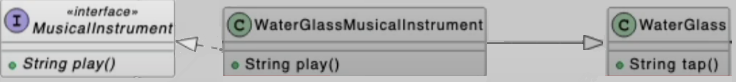


Creiamo una classe (Oject adapter) contiene un unico GermanPercussion(da passare nel costruttore per esser sicuro che ci sia sempre (implementa musicInstru quindi play dove returna il metodo Spiel)) e questo è vedibile come MusicalInstrument



Mentre invece per Water class creiamo unca classe che estende e implementa MusicallInsutrument e che chiamera

Al suo interno il metodo equivalente di Water class  
(per intenderci creiamo una classe WaoterClassMusicalInstrument che eredita ed estende MusicallInsutrument quidni avrà un overide di play e dentre play metteremo return tap stiamo “traducendo” il metodo)



**Composite Pattern (dacontrollare)**

L’idea è quella di creare una classe che rappresenta un insieme (che quindi avra una struttura tipo lista di strumenti) che implementa delle operazioni comune come play nel nostro caso

**Decoretor Pattern (dafinire)**

Serve ad aggiungere funzionalità ad una tipologia di oggetti come in questo caso MusicalInstrument

Mokito Mokkare il costruttore

Quando? -> variabile final e quindi non puoi fare altrimenti

Esempio:

void cotruttoreTest() {

try (MockedConstruction<Tavolo> mocked = mockConstruction(Tavolo.class, (mock, context) -> {

when(mock.inMostra(prima)).thenReturn(true);

})) {

Partita p = new Partita();

assertThat(p.controllaSeCartaPresenteSuTavolo(prima)).isEqualTo(true);

}

}

(mock, context) -> sarebbe MockedConstruction..MockInitializer<tipo della classe da mockare>{

Public void prapare (mock, context) Throwable{ when ()then}

}

Come usare la funzion che aveva dato lui?



Visto che c’è thenAnswer ricrea literatore va usato quando non quando stiamo testando literable (ma mokkare un oggetto vero) lo usiamo quando il mio oggetto per svolgere le sue funzioni doveva iterare su un oggetto esterno

Ad esempio se un metodo prende un iteratore scegli prende un iteratore mano avendo quella funzione se rifacciamo il test resetta quest'ultimo e avremo lo stesso risultato

