

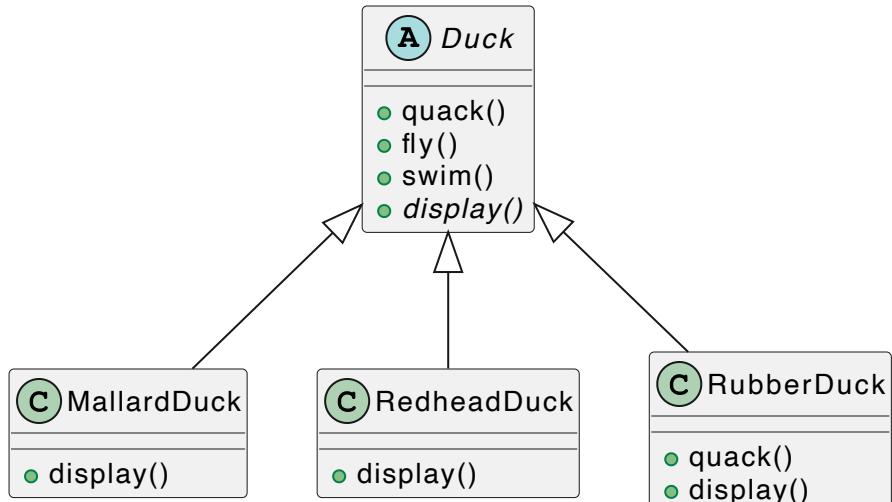


UNIVERSITÀ DEGLI STUDI  
DI MILANO

# Ingegneria del software

7 novembre 2022

# Duck saga

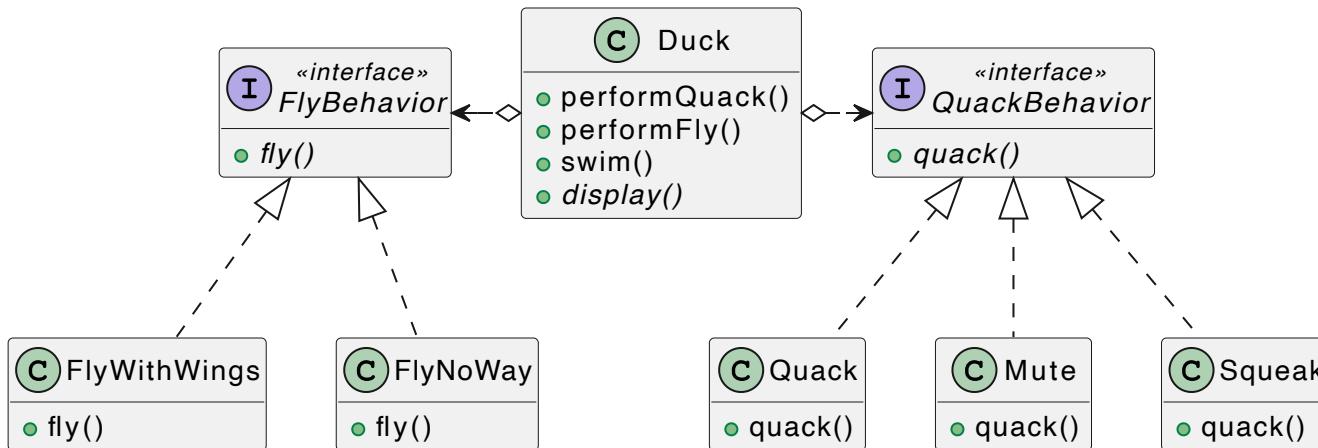


Come risolvo  
"**problema**" `fly` ?

- override
- interfacce
- delegation

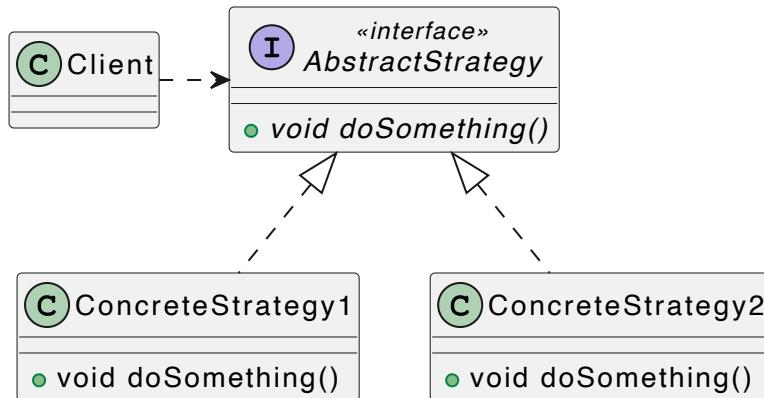
# Principi Dependency Inversion e Open Close

Identificare gli aspetti della applicazione che cambiano e separarli da ciò che rimane fisso



# Delegation/Strategy

Definisce una famiglia di algoritmi, e li rende (tramite encapsulation) tra di loro intercambiabili

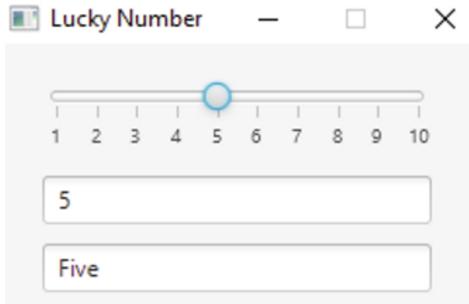


Abbiamo visto `sort` di `Collections` che richiedeva un parametro `Comparable`, una altra possibilità è il metodo `sort` con un secondo parametro `Comparator`

```
static <T> void sort(List<T> list, Comparator<? super T> c)
```

# Esempi

di situazioni che risolveremo con pattern Observer



```
public final class Card
{
    // Indexed by suit, then rank
    private static final Card[][] CARDS;

    // Create the flyweight object
    static
    {
        private final Rank aRank;
        private final Suit aSuit;

        private Card(Rank pRank, Suit pSuit)
        {
            aRank = pRank;
            aSuit = pSuit;
        }
    }

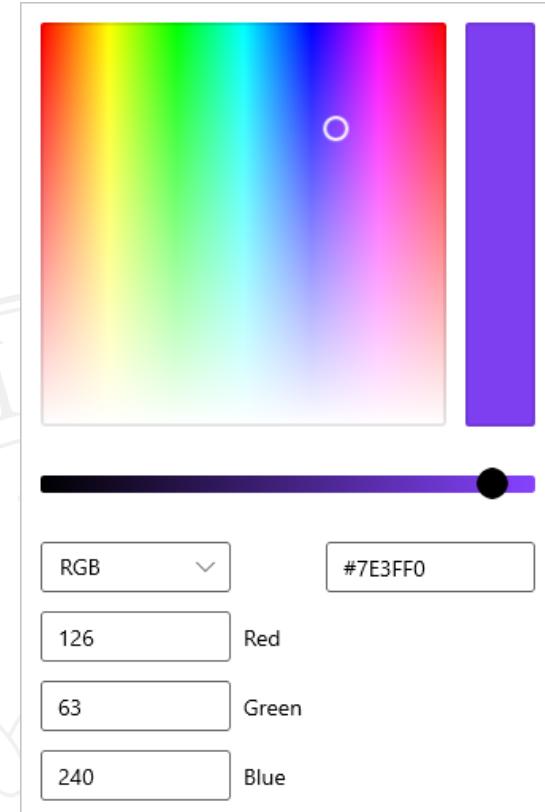
    public Card(Rank rank, Suit suit)
    {
        aRank = rank;
        aSuit = suit;
    }

    public String getIDString()
    {
        return aRank + " of " + aSuit;
    }

    public Rank getRank()
    {
        return aRank;
    }

    public Suit getSuit()
    {
        return aSuit;
    }

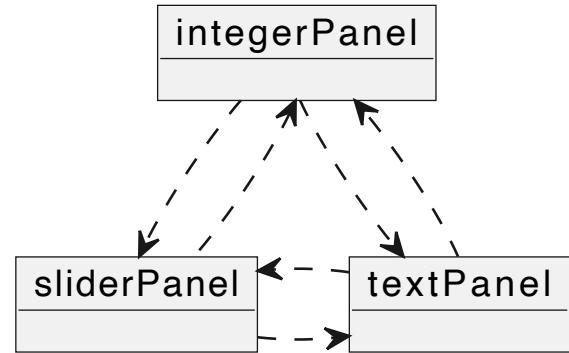
    public String toString()
    {
        return IDString();
    }
}
```



# Problemi

## PAIRWISE DEPENDENCIES

- Forte accoppiamento
  - ogni vista deve conoscere le altre viste
- Bassa espandibilità
  - complicato aggiungere e togliere altre viste

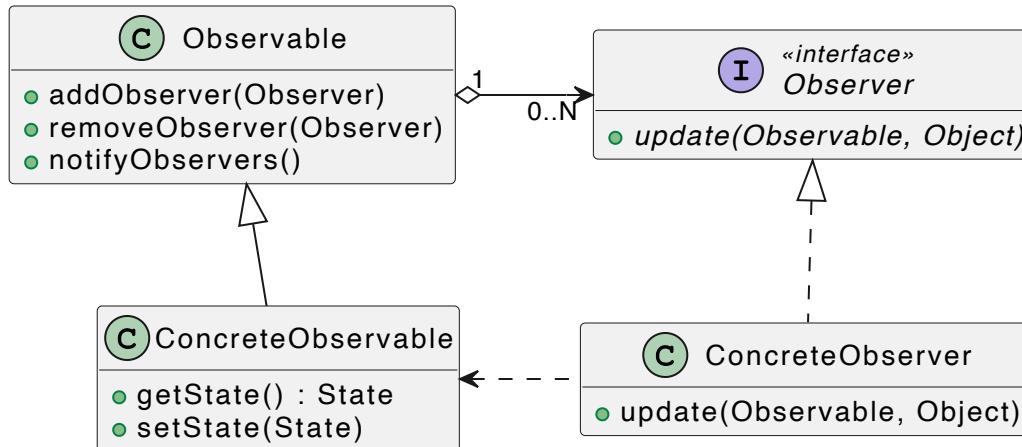


# Soluzione

- Estraiamo la parte comune: lo stato
- La mettiamo in un oggetto a parte (*Subject*)
- che verrà osservato dagli altri (*Observer*)
- in Java c'erano delle classi nelle librerie standard per realizzare questo pattern
  - **interfaccia** `java.util.Observer`
  - **classe** `java.util.Observable`

# OBSERVER pattern

- Come colleghiamo Observable e Observer?
- Come scoprono gli Observer lo stato dell'Observable?
- Quando l'Observable viene notificato?

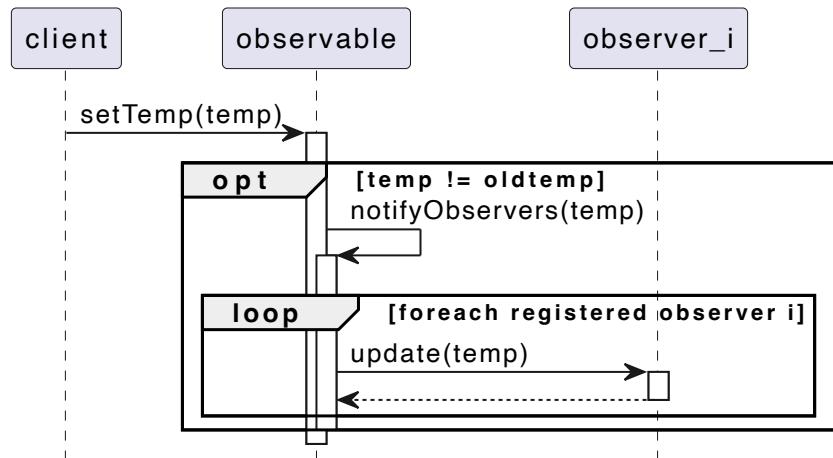


# OBSERVER: push

Lo stato modificato viene passato come argomento alla *callback*

```
//OBSERVABLE
@Override
public void notifyObservers(){
    for (Observer observer : observers) {
        observer.update(null, state);
    }
};

//OBSERVER
@Override
public void update(Observable model,
                   Object state) {
    if (state instanceof Integer intValue)
        doSomethingOn(intValue);
}
```

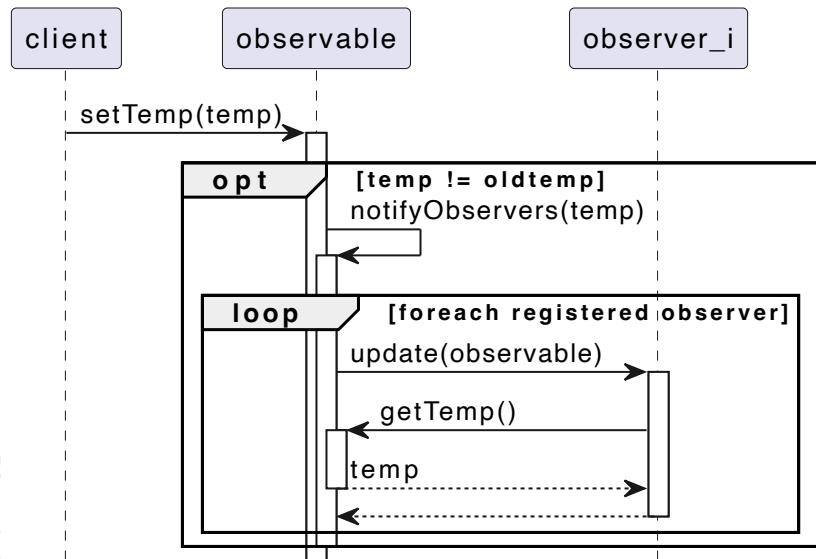


# OBSERVER: pull

Lo stato modificato viene passato come argomento alla *callback*

```
//OBSERVABLE
@Override
public void notify0bservers(){
    for (Observer observer : observers) {
        observer.update(this, null);
    }
};

//OBSERVER
@Override
public void update(Observable model,
                   Object state) {
    if (model instanceof Concrete0bservable cModel)
        doSomethingOn(cModel.getState());
}
```

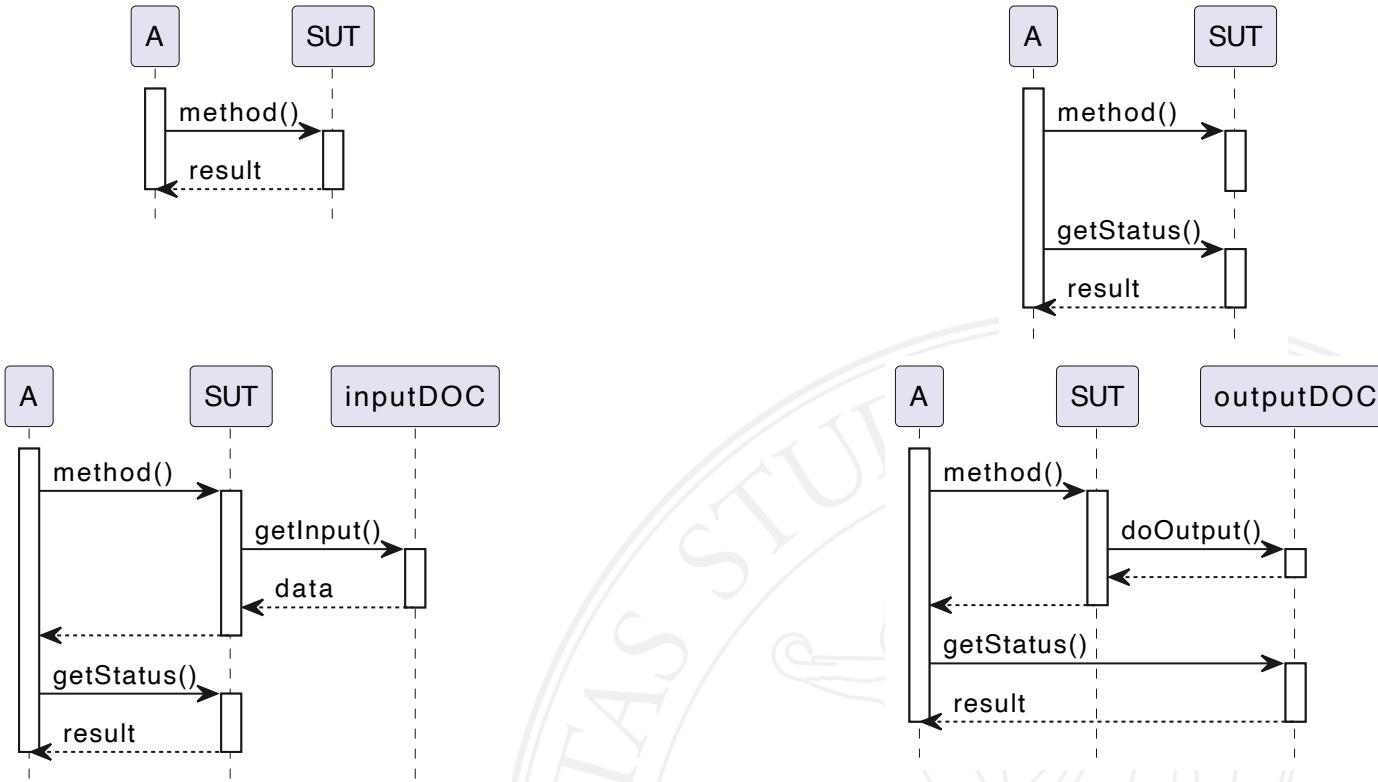




UNIVERSITÀ DEGLI STUDI  
DI MILANO

# Mocking

# Approcci al testing



# Dummy objects

- oggetti che sono passati in giro ma mai veramente usati
  - non posso passare *null*
  - potrei avere solo una interfaccia e non una classe
  - potrei avere solo costruttori complessi

```
@Test
public void testDummy() {
    MyClass dummy = ?? ;

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

    SUT.add(dummy);

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

# Stub Objects

- oggetti che forniscono risposte preconfezionate alle sole chiamate fatte durante il testing

```
@Test
public void testConStub() {
    MyClass stub = ?? ;

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

    SUT.add(stub.getValue(0)); // deve ritornare 4
    SUT.add(stub.getValue(1)); // deve ritornare 7
    SUT.add(stub.getValue(1)); // deve ritornare 3

    res = SUT.somma();

    assertThat(res).isEqualTo(14);
}
```

# Mock objects

- oggetti che instrumentano e controllano le chiamate

```
@Test
public void testConMock() {
    MyClass mock = ?? ;

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

    res = SUT.somma(mock);

    assertThat(res).isEqualTo(14);
    // assert che getValue è stata chiamata 3 volte
    // prima una volta con parametro 0 e poi...
}
```

# Spy objects

- oggetti che instrumentano e controllano le chiamate di oggetti reali (di cui possono usare i metodi e lo stato)

```
@Test
public void testConSpy() {
    MyClass spy = ?? ; // esiste classe reale MyClass

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

    res = SUT.somma(spy);

    assertThat(res).isEqualTo(14);
        // assert che getValue è stata chiamata 3 volte
        // prima una volta con parametro 0 e poi...
}
```

# Fake objects

- oggetti che implementano il DOC ma usando qualche scorciatoia, in maniera non realistica o non installabile
  - database in memoria invece di database reale
  - soluzione inefficiente per casi di dimensione significativa



# Mockito



Libreria (framework) per costruire mock objects (e non solo!)

# Esempi di prima

```
@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() {  
    MyClass stub = mock(MyClass.class);  
    when(stub.getValue(0)).thenReturn(4);  
    when(stub.getValue(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);  
}
```

# Esempi di prima

```
1  @Test
2  public void testConMock() {
3      MyClass mock = mock(MyClass.class);
4
5      when(mock.getValue(0)).thenReturn(4);
6      when(mock.getValue(1)).thenReturn(7,3);
7
8      MyList<int> SUT = new MyList<int>();
9
10     res = SUT.somma(mock);
11
12     assertThat(res).isEqualTo(14);
13     InOrder io = inOrder(mock);
14     io.verify(mock).getValue(0);
15     io.verify(mock, times(2)).getValue(1);
16 }
```

```
1  @Test
2  public void testConSpy() {
3      MyClass spy = spy(new MyClass());
4
5      MyList<int> SUT = new MyList<int>();
6
7      res = SUT.somma(spy);
8
9      assertThat(res).isEqualTo(14);
10     InOrder io = inOrder(spy);
11     io.verify(spy).getValue(0);
12     io.verify(spy, times(2)).getValue(1);
13 }
```

# stubbing

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

- *args*: values | matchers | argumentCaptor
- *matchers*: anyInt(), argThat(is(closeTo(1.0, 0.001)))
- *thenXXX*: thenReturn | thenThrows | thenAnswer | thenCallRealMethod
- *values*

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

sembra uguale ma quella prima non funziona quando metodi ritornano void

# Verifying

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 in0 = inOrder(mock1, mock2, ...)  
in0.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());
```



# Observer pattern mocking

- test del model con mock degli Observers

```
@Test
void modelTest {
    //SETUP
    Model model = new Model();
    Observer obs = mock(Observer.class);
    Observer obs1 = mock(Observer.class);

    //EXERCISE
    model.addObserver(obs);
    model.addObserver(obs1);
    model.setTemp(42.0, scale);

    //VERIFY
    verify(obs).update(eq(model), eq("42.0"));
    verify(obs1).update(eq(model), eq("42.0"));
}
```

# Observer pattern mocking (cont)

- test di un observer con un modello non generico ma di cui ho solo interfaccia di cui fornisco una versione dummy

```
@Test
void observerTest {
    //SETUP
    abstract class MockObservableIModel extends Observable implements Model {};
    MockObservableIModel model = mock(MockObservableIModel.class);
    when(model.getTemp()).thenReturn(42.42);

    //EXERCISE
    observer.update(model, null);

    //VERIFY
    verify(model).getTemp();
    assertThat(observer.getVal()).isCloseTo(42.42, Offset.offset(.01));
}
```

# Un po' di codice laboratorio 5

```
1  @Test
2  void turnoConNessunaCartaSuTavoloTest() {
3      //SETUP
4      Partita partita = new Partita();
5      Tavolo tavolo = partita.getTavolo();
6      svuotaTavolo(tavolo);
7
8      Giocatore mattia = new Giocatore("Mattia", partita);
9      mattia.daiCarta(Card.get(Rank.ACE, Suit.DIAMONDS));
10     mattia.setStrategia(SelettoreCarta.FIRSTCARD);
11
12     //EXERCISE
13     mattia.turno();
14
15     //VERIFY
16     assertThat(tavolo.inMostra(Card.get(Rank.ACE, Suit.DIAMONDS))).isTrue();
17 }
18
19 private void svuotaTavolo(Tavolo tavolo) {
20     Deck d = new Deck();
21     while (!d.isEmpty()) tavolo.prendi(d.draw());
22 }
```

# Riscriviamolo con Mockito

```
1  @Test
2  void turnoConNessunaCartaSuTavoloTestConMockito() {
3      Tavolo t = mock(Tavolo.class);
4      when(t.inMostra(any())).thenReturn(false);
5
6      Partita p = mock(Partita.class);
7      when(p.getTavolo()).thenReturn(t);
8      Iterator<Giocatore> it = mock(Iterator.class);
9      when(p.iterator()).thenReturn(it);
10     when(it.hasNext()).thenReturn(false);
11
12     Giocatore mattia = new Giocatore("Mattia", p);
13     mattia.daiCarta(Card.get(Rank.ACE, Suit.DIAMONDS));
14     mattia.setStrategia(SelettoreCarta.FIRSTCARD);
15
16     mattia.turno();
17
18     verify(t).metti(Card.get(Rank.ACE, Suit.DIAMONDS));
19 }
```

# Giocatore

```
public void turno() {  
    // STRATEGIA  
    Card card = strategia.scegli(mano, partita);  
  
    // ATTUAZIONE MOSSA  
    mano.remove(card);  
    if (partita.getTavolo().inMostra(card)) {  
        prendiDaTavolo(card);  
    } else if (!rubiMazzetto(card))  
        partita.getTavolo().metti(card);  
}
```

```
private void prendiDaTavolo(Card card) {  
    partita.getTavolo().prendi(card);  
    punti += 2;  
    mazzettoTop = card.getRank();  
}  
  
private boolean rubiMazzetto(Card card) {  
    for (Giocatore giocatore : partita) {  
        if (giocatore != this &&  
            giocatore.getMazzettoTop() == card.getRank()) {  
            punti += 1 + giocatore.cediMazzetto();  
            mazzettoTop = card.getRank();  
            return true;  
        }  
    }  
    return false;  
}  
  
private int cediMazzetto() {  
    int vincita = getPunti();  
    punti = 0;  
    mazzettoTop = null;  
    return vincita;  
}
```



# distribuisci mano

```
public void distribuisciMano(int num) {  
    // PRE CONDIZIONI  
    assert num <= 3;  
  
    int max = deckSize() / giocatori.size();  
    for (int i = 0; i < num && i < max; i++) distribuisciCarta();  
  
    // POST CONDIZIONI  
    for (Giocatore giocatore : giocatori) {  
        assert giocatore.numCards() <= 3 :  
            "non si possono avere più di tre carte in mano";  
        assert giocatori.get(0).numCards() == giocatore.numCards() :  
            "non è stato dato stesso numero di carte a tutti";  
        assert giocatore.numCards() == 3 || deckSize() < giocatori.size() :  
            "si possono avere meno di tre carte solo se nel mazzo non ce ne sono abbastanza per fare un altro giro";  
    }  
}
```

AVVERTENZA: questa implementazione soddisfa le postcondizioni nei casi in cui viene effettivamente usata, ma non le garantisce in generale (andrebbero aggiunte delle precondizioni tipo che il numero di carte in possesso ai vari giocatori sia lo stesso per tutti e minore o uguale a 3 e che num sia minore di 3 - numero carte in possesso inizialmente)

```

class SelettoreRubaMazzettoTest {
    @Test
    void scegliSuccess() {
        Giocatore player1 = mock(Giocatore.class);
        Giocatore player2 = mock(Giocatore.class);
        Giocatore me = mock(Giocatore.class);

        when(player1.getMazzettoTop()).thenReturn(Rank.FIVE);
        when(player2.getMazzettoTop()).thenReturn(Rank.TEN);

        Partita p = mock(Partita.class);
        MockUtils.whenIterated(p, me, player2, player1);

        List<Card> mano = List.of(Card.get(Rank.ACE, Suit.CLUBS), Card.get(Rank.KING, Suit.DIAMONDS), Card.get(Rank.TEN, Suit.HEARTS));
        SelettoreCarta FAILED = mock(SelettoreCarta.class);

        SelettoreCarta strategy = new SelettoreRubaMazzetto(FAILED, me);

        assertThat(strategy.scegli(mano, p)).isEqualTo(Card.get(Rank.TEN, Suit.HEARTS));
    }

    public static <T> void whenIterated(Iterable<T> p, T... d) {
        when(p.iterator()).thenAnswer((Answer<Iterator<T>>) invocation -> List.of(d).iterator());
    }
}

```

```
1  @Test
2  void scegliFail() {
3      Giocatore player1 = mock(Giocatore.class);
4      Giocatore player2 = mock(Giocatore.class);
5      Giocatore me = mock(Giocatore.class);
6
7      when(player1.getMazzettoTop()).thenReturn(Rank.FIVE);
8      when(player2.getMazzettoTop()).thenReturn(Rank.QUEEN);
9
10     Partita p = mock(Partita.class);
11     MockUtils.whenIterated(p, me, player2, player1);
12
13     List<Card> mano = List.of(Card.get(Rank.ACE, Suit.CLUBS), Card.get(Rank.KING, Suit.DIAMONDS), Card.
14     SelettoreCarta FAILED = mock(SelettoreCarta.class);
15     when(FAILED.scegli(any(), any())).thenReturn(mock(Card.class));
16
17     SelettoreCarta strategy = new SelettoreRubaMazzetto(FAILED, me);
18
19     strategy.scegli(mano, p);
20
21     verify(FAILED).scegli(mano, p);
22 }
```

testImplementation 'org.mockito:mockito-inline:4.8.0'

# Selettore Ruba Mazzetto

```
public class SelettoreRubaMazzetto implements SelettoreCarta {  
    @NotNull private final SelettoreCarta next;  
    @NotNull private final Giocatore me;  
  
    public SelettoreRubaMazzetto(@NotNull SelettoreCarta next, @NotNull Giocatore me) {  
        this.next = next;  
        this.me = me;  
    }  
  
    @Override  
    @NotNull  
    public Card scegli(@NotNull List<Card> mano, @NotNull Partita partita) {  
        for (Card card : mano) {  
            for (Giocatore giocatore : partita) {  
                if (giocatore != me && giocatore.getMazzettoTop() == card.getRank())  
                    return card;  
            }  
        }  
        return next.scegli(mano, partita);  
    }  
}
```

# Selettore Proteggi Mazzetto

```
public class SelettoreProteggiMazzetto implements SelettoreCarta {  
    @NotNull private final SelettoreCarta next;  
    @NotNull private final Giocatore me;  
  
    public SelettoreProteggiMazzetto(@NotNull SelettoreCarta next, @NotNull Giocatore me) {  
        this.next = next;  
        this.me = me;  
    }  
  
    @NotNull  
    @Override  
    public Card scegli(@NotNull List<Card> mano, @NotNull Partita p) {  
        for (Card card : mano) {  
            if (card.getRank() == me.getMazzettoTop())  
                return card;  
        }  
        return next.scegli(mano, p);  
    }  
}
```