Con valori "assenti"

Aggiungiamo la carta **Joker** che non ha nè *Suit* nè *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 ritorano 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`

Esempio di uso NullObject Pattern

```
Card x = deck.draw();
if (x != null)
  System.out.print(x.desc());
else
  System.out.print("Carta non esistente");



Card draw() {
  if (isEmpty())
    return null;
  else
    return internal.remove(0);
}
```

```
Card x = deck.draw();
System.out.print(x.desc());



Card draw() {
  if (isEmpty())
    return Card.NULL;
  else
    return internal.remove(0);
}

static Card NULL = new Card() {
  public String desc() { return "Carta non esistente"; }
};
```

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html

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

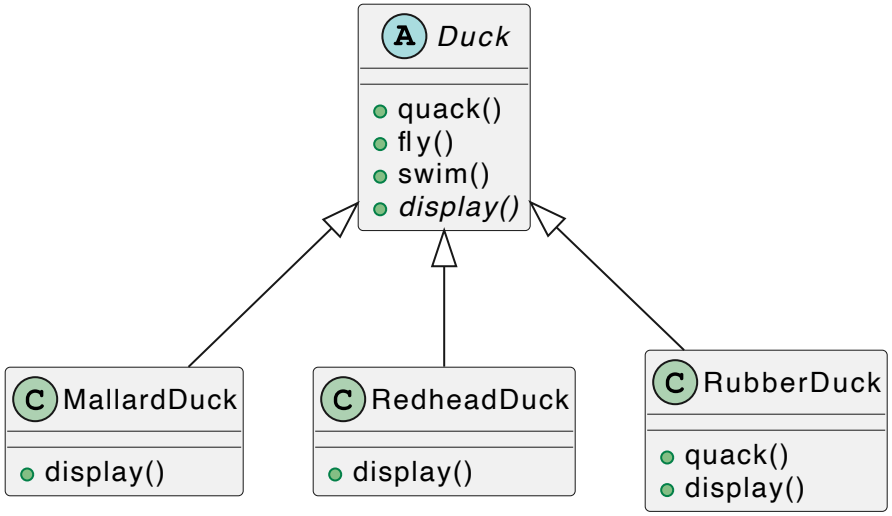
Al posto del costruttore definisce tre metodi statici:

- `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

Duck saga

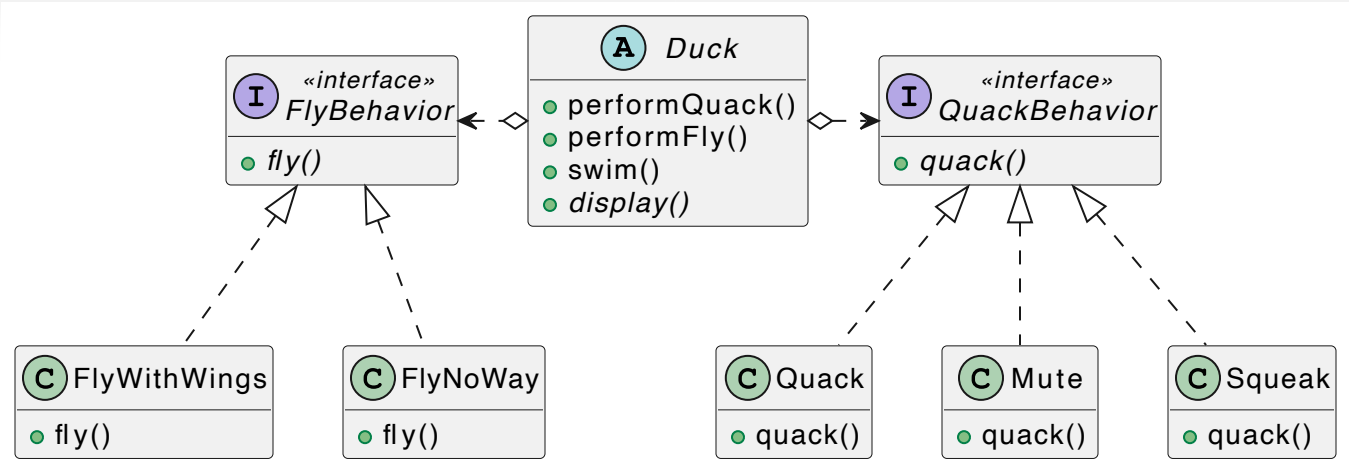


Come risolvo "*problema*" `fly` ?

- override
- interfacce
- delegation

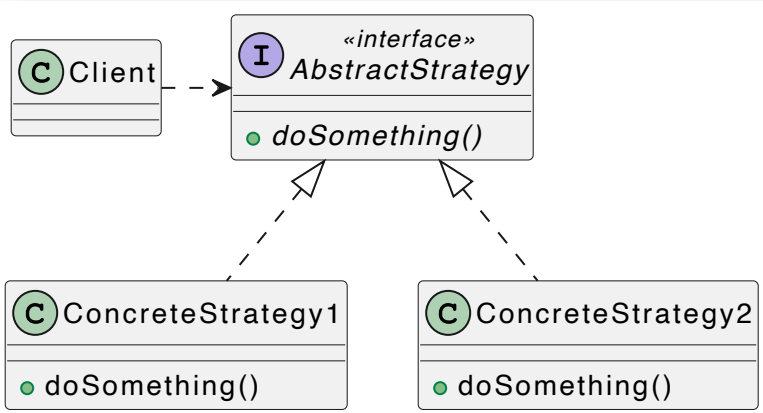
Principi Dependency Inversion e Open Close

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`

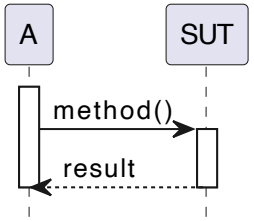
una altra possibilità è il metodo sort con un secondo parametro `Comparator`

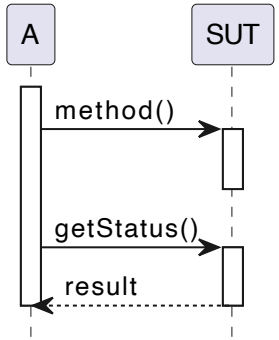
```
static <T> void  sort(List<T> list, Comparator<? super T> c)
```

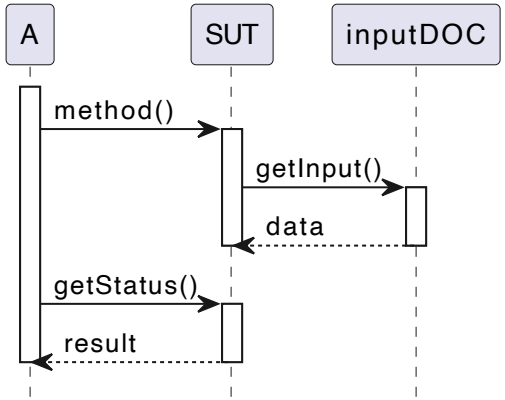
UNIVERSITÀ DEGLI STUDI
DI MILANO

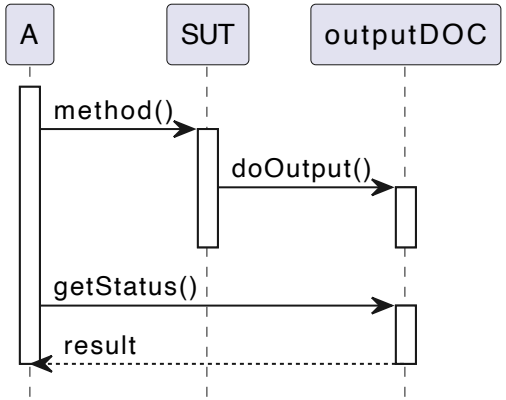
Mocking

Approcci al testing









http://xunitpatterns.com/Test Double.html

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

```java
@Test
public void testDummy()  {
  MyClass dummy =  mock(MyClass.class);

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

  SUT.add(dummy);

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

```java
@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

```java
@Test
public void testConMock()  {
  MyClass mock =  mock(MyClass.class);

  when(mock.getValue(0)).thenReturn(4);
  when(mock.getValue(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).getValue(0);
  io.verify(mock, times(2)).getValue(1);
}
```

```java
@Test
public void testConSpy()  {
  MyClass spy =  spy(new MyClass());

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

  res = SUT.somma(spy);

  assertThat(res).isEqualTo(14);
  InOrder io = inOrder(spy);
  io.verify(spy).getValue(0);
  io.verify(spy, times(2)).getValue(1);
}
```

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 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());
```