

# Software Engineering Fundamentals

## • Definizioni

- Metodi e variabili statiche
  - le uso quando non mi interessa che vengano usate da un'istanza in particolare, bensì voglio essere in grado di usarle senza istanziare un oggetto.
- Metodi puri
  - 1 – it always returns the same result for same argument values.
  - 2 – it has no side effects like
    - modifying an argument (or global variable); outputting something
  - 3 – the only result of calling a pure function is the return value.
  - In parole povere: Un metodo è puro se non ha effetti collaterali (se non solleva eccezioni). In JML assignable vale \nothing, ma preferiamo scrivere “pure” perché ci consente di richiamare il metodo in JML.
- Interfaccia
  - Può essere considerata una classe abstract pura, in quanto fornisce solo una forma, ma nessuna implementazione: `public /*@ pure @*/ class Interfaccia { ... }`. Può ereditare da un'altra interfaccia.

## • ArrayList<T>

- Metodi
  - `E get(i)`
    - ritorna l'oggetto contenuto all'indice `i`.
  - `boolean add(E e)`
    - aggiunge `e` alla fine della lista
  - `void add(int index, E e)`
    - aggiunge `e` in posizione `i`
  - `boolean contains(Object o)`
  - `boolean containsAll(p)`
    - example post condition: `\result.containsAll(points) && points.containsAll(\result)`
  - `int size()`
  - `boolean remove(Object o)`
    - rimuove il primo oggetto che trova uguale a `o`, se esiste.
  - `Object[] toArray()`
    - Ritorna un array contenente gli elementi della lista nella stessa sequenza.

## • Iterator<T>

- Metodi
  - `boolean hasNext()`
  - `Object next()` Throws `NoSuchElementException`
  - `void remove()` → Removes the current element

### ○ Implementazione 1

```
All'interno della classe Videoteca si definiscono il metodo e la classe privata per l'iteratore.
public class Videoteca {
    private ArrayList<Film> film;
    public boolean contains (Film f);
    ...
    public direttoDa(Artista a) { return new VideoIter(this, a); }

    private static class VideoIter implements Iterator<Film> {
        private Videoteca vid;
        private Artista reg;
        private int n;
        private searchNext() {
            for ( ; n<vid.size() && !vid.film.get(n).registra().equals(reg); n++);
        }
        VideoIter(Videoteca v, Artista a) {
            vid=v; reg=a; n=0; searchNext();
        }
        public boolean hasNext() { return n<vid.size(); }
        public Film next() throws NoSuchElementException {
            if (n<vid.size()) throw new NoSuchElementException();
            Film f = film.get(n); searchNext(); return f;
        }
    }
}
```

### ○ Implementazione 2

```
ArrayList al = new ArrayList();
//riempi l'array...
Use iterator to display contents of al
System.out.print("Original contents of al: ");
Iterator itr = al.iterator();
while(itr.hasNext()) {
    Object element = itr.next();
    System.out.print(element + " ");
}
System.out.print();
```

- Metodi di Set<T>

- boolean contains(Object element)
- boolean containsAll(Collection<T> C)
- void clear()
- boolean equals(Object o)
- int size()
- boolean isEmpty()
- Object[] toArray()
- Iterator<E> iterator()
- boolean add(T element) - opzionale
- boolean remove(Object element) - opzionale
- boolean addAll(Collection<T> C) - opzionale
- boolean removeAll(Collection C) - opzionale

- Label (Etichetta) & instanceof

- Label public class **Label** extends **Component** implements **Accessible**.  
A Label object is a component for placing text in a container. A label displays a single line of read-only text. The text can be changed by the application, but a user cannot edit it directly.
- Un figlio è un instanceof di un padre

- Eccezioni

- Eccezioni checked
  - Sottotipo di Exception.
  - Gestite da blocco try/catch.
- Eccezioni unchecked
  - Sottotipo di RuntimeException
  - Possono propagarsi senza essere dichiarate in alcuna intestazione di metodo e senza essere gestite da nessun blocco try/catch
  - Può essere meglio includerle comunque in throws, per renderne esplicita la presenza (ma per il compilatore è irrilevante).

- Ereditarietà

- Procedimento Meccanico
  - Animale a = new balena();  
Animale b = new balena();  
Balena c = new Balena();  
Dinosauro d = new Dinosauro();  
**a.mangia(b)**
    - Ancora il tipo statico all'argomento → **[b] = Animale**
    - Uso il tipo (quindi quello statico o quello dinamico) di a che più si avvicina a implementare a.mangia(Animale b) → **[a] = Balena**
    - **Balena.mangia(Animale)**
  - c.metodo(d)**
    - Ancora il tipo statico all'argomento  
[d] = Dinosauro
    - Se non esiste un metodo mangia(Dinosauro) nè nella classe del tipo statico (**Dinosauro**) nè in quella del tipo dinamico (sempre **Dinosauro**), allora viene cercata la classe mangia(DinosauroPadre) = mangia(Animale)
    - **Dinosauro.mangia(Animale)**
  - In soldoni: **se una classe chiama metodo(figlio1 a) e questo metodo non esiste all'interno della classe, allora viene chiamato metodo (padre a)!**

- Assegnazione polimorfica:

- ```
Animale a = ...;
Gatto mao = ...;           // eredita da Animale
a = mao;                   // OK: assegnazione polimorfica
```

- ClassCastException

Viene sollevata nei seguenti casi

- ```
Cat cat = new Cat();
Dog dog;
dog = (Dog) cat; → Dog e Cat sono due tipi incompatibili.
```
- ```
public class Animal{...};
public class Dog extends Animal{...};
public class InheritanceSample{
    public static void main(String[] args){
        Animal animal = new Animal();
        Dog dog;
        dog = (Dog) animal; // ERRORE: Il casting funziona solo convertendo un figlio in
padre.
        animal = (Animal) new Dog(); // GIUSTO!
    } }
```

ovvero:

**the conversion is valid only when the child class is casted to its parent class**

- Esempio importante di ereditarietà e polimorfismo:

In caso di polimorfismo tengo in considerazione il tipo dinamico.

- ```
Animal a1 = new Dog();
Animal a2 = new DogSon();
Dog dog1 = (Dog) a1;
Dog dog2 = (Dog) a2;
```

**dog2.greeting(dog1);**  
Ancora a dog 1 il suo tipo statico: Dog.  
L'assegnazione fatta a dog2 è lecita in quanto faccio un casting da figlio (DogSon) a padre(Dog), infatti devo tenere in considerazione il tipo dinamico di a2 (DogSon) per fare vedere se il casting è legittimo.  
Ora che so che è lecita mi chiedo quale metodo viene chiamato, l'algoritmo è semplice: scendo l'albero finché posso → il tipo dinamico di a2 è DogSon, se DogSon ha un metodo greeting(tipo Dog) uso quello.  
Garantito dalle regole di ereditarietà e polimorfismo.

- Overloading

- All'interno di una stessa classe possono esservi più metodi con lo stesso nome purché si distinguano per numero e/o tipo dei parametri.  
Attenzione: Il tipo e il valore restituito non basta a distinguere due metodi.  
Metodi overloaded devono avere intestazioni diverse.  
In Java l'intestazione di un metodo comprende il numero, il tipo e la posizione dei parametri;  
Non include il tipo del valore restituito.  
Utile per definire funzioni con codice differente ma con effetti simili su tipi diversi.

- Regole estensioni – criteri di sostituibilità

- Regola della preconditione

- $Precondizione_{super} \rightarrow Precondizione_{sub}$

- La preconditione del metodo ridefinito è più debole di quella del metodo originale, allora in tutti i casi in cui si chiamava il metodo originale si può chiamare anche il metodo ridefinito.

- Regola della segnatura

- Garantisce che il contratto della superclasse sia ancora applicabile, ossia che la sintassi della sottoclasse sia compatibile con la sintassi della superclasse.
      - I metodi non cambiano prototipo (del prototipo fanno parte anche le eccezioni)
      - I figli devono avere al minimo gli stessi metodi dei padri

- Regola dei metodi

- le chiamate ai metodi del sottotipo devono comportarsi come le chiamate ai metodi corrispondenti del supertipo. Il comportamento è anche dettato dal sollevare o meno eccezioni nei metodi.

- Regola delle proprietà

- Un sottotipo deve preservare tutti i public invariant degli oggetti del supertipo.
      - Public invariant:
        - Rappresenta le proprietà invarianti degli stati astratti osservabili, ovvero le proprietà osservabili tramite i metodi **observer**
        - per scrivere il public invariant (in JML) si possono usare solo attributi e metodi public della classe.

- **JML**

- Algoritmo di aggiunta eccezioni

- `//@ requires in >= 0;`  
`//@ ensures Math.abs(\result*\result-in)<0.0001;`  
`public static float sqrt(float in) { ... }`
    - Becomes:
    - `//@ requires true;`  
`//@ ensures in >= 0 && Math.abs(\result*\result-in)<0.0001;`  
`//@ signals(NegativeException) in < 0;`  
`public static float sqrt(float in) { ... }`

- Quantificatori in JML

- $(\sum \text{int } i; 0 \leq i \ \&\& \ i < 5; i) == 0 + 1 + 2 + 3 + 4$
    - $(\text{product int } i; 0 \leq i \ \&\& \ i < 5; i) == 1 \cdot 2 \cdot 3 \cdot 4$
    - $(\max \text{int } i; 0 \leq i \ \&\& \ i < 5; i) == 4$
    - $(\min \text{int } i; 0 \leq i \ \&\& \ i < 5; i - 1) == -1$

- Specifica

- Nelle pre e post-condizioni di un metodo pubblico possono comparire solo gli elementi pubblici del metodo e della classe.  
In particolare, I parametri formali e `\result`, ma anche metodi pubblici puri o attributi pubblici.  
Ma non i metodi pubblici che non sono dichiarati puri!
    - Requires:
      - che i parametri dati in pasto alla funzione siano validi (`!=null` || `</=> X`)
    - Ensures:
      - che I parametri e la funzione facciano il loro dovere
    - Metodo puro
      - Osservatore
        - `/*@pure@*/`
      - Non Osservatore
        - `//@assignable \nothing`
    - Algoritmo per la specifica:
      - Se il metodo è puro (non solleva eccezioni)  $\rightarrow$  `//@assignable \nothing`
      - Precondizioni: valori validi dei parametri: `//@requires parametri !=NULL` (nella maggior parte dei casi basta scrivere solo questo)
      - Postcondizioni: il metodo deve fare effettivamente ciò che gli viene chiesto di fare.

- Esempi

- Funzione di adding
      - `//@requires`  
`//@ d!=null &&`  
`//@ salario>0 &&`  
`//@ salarioOk(salarioStandard(), responsabile(), d) &&`  
`//@ !isDipendente(d) //@ &&`  
`//@ !(\exists Dipendente d1; isDipendente(d1);`  
`//@ d!=d1 && d.persona()==d1.persona());`  
`//@ensures`  
`//@ isDipendente(d) &&`  
`//@ (\forallall Dipendente d1; d1!= d;`  
`//@ \old(isDipendente(d1)) <==> isDipendente(d1));`  
**`public void addDipendente(Dipendente d, int salario)`**

- Esempio intSet

- // Costruttori:  
`//@ensures this.size()==0;`  
`public IntSet() { *inizializza this come insieme vuoto* }`  
  
`//Modificatori:`  
`/*@ensures`  
`@ size()== \old(size()+1) &&`  
`@ this.isIn(x) &&`  
`@ (\forallall int y; x!=y; \old(isIn(y)) <==> isIn(y));`  
`@ */`  
`public void insert(int x){ *inserisce x in this* }`  
  
`/*@ ensures !this.isIn(x) &&`  
`@ size()== \old(size()-1) &&`  
`@ (\forallall int y; x!=y; \old(isIn(y)) <==> isIn(y)); */`  
`public void remove(int x) { *rimuove x da this* }`

- Rep Invariant / Invariante di rappresentazione (Concreto):

- prendo gli attributi della classe e scrivo che questi attributi non possono assumere determinati valori( un ArrayList e i nodi al suo interno non possono essere nulli, gli indici saranno minori e maggiori di un determinato valore, etc...).
- Inoltre scrivo ovviamente qual è il compito della funzione.
- L'invariante di rappresentazione ("rep invariant" o RI) predicato J:  $\text{ConcSt} \rightarrow \text{Boolean}$  È vero solo per oggetti "legali":  $J([1,2,2]) = \text{false}$ ,  $J([1,3,2]) = \text{true}$ .
- Quando, come, cosa scrivere nel rep invariant?
  - Si scrive private invariant... prima di implementare qualunque operazione.
  - Ci scrivo
    - Le proprietà che caratterizzano queglii stati concreti che rappresentano qualche stato astratto.
    - Il rep invariant, insieme alla AF, deve includere tutto ciò che serve per potere implementare i metodi, data la specifica.
- Esempi
  - Alberto implementa i metodi **insert** e **isIn**,
  - Bruno implementa **remove** e **size**, ma i due non si parlano: usano solo il rep invariant!
  - Intset: Per IntSet non basta scrivere **elements !=null**:  
bisogna anche dire che non ci sono duplicati!
  - il rep invariant di un oggetto concreto *c* è:
    - *c.arrayRep* è sempre  $\neq \text{null}$  &&
    - *c.arrayRep* contiene solo elementi  $\neq \text{null}$  &&
    - in *c.arrayRep* non ci sono mai due interi con lo stesso valore

- Invariante Astratto – Public invariant

- Proprietà degli stati astratti osservabili
  - Esempio:
    - *size()* di un IntSet è sempre  $\geq 0$ .
    - il grado di un Poly non è negativo
  - Sono rappresentabili in JML con **public invariant...**  
**Ricapitolando: L'invariante concreto si rappresenta con private invariant, l'invariante astratto con public invariant.**

- **TESTING**

- **Test funzionale (Black Box)**

- Casi di test determinati in base a ciò che il componente deve fare in base alla sua specifica
- Criterio di copertura delle combinazioni proposizionali / Test Combinatorio
  - 1 – Scomporre la specifica
    - Per ciascuna feature prevista, identificare parametri, elementi dell'ambiente.
    - Per ciascun parametro ed elemento dell'ambiente, identificare le caratteristiche elementari (categorie)
  - 2 – Identificare i valori
    - Identificare classi rappresentative di valori per ciascuna categoria. Ignorare le interazioni tra i valori di diverse categorie (vedi prossimo step)
    - I valori rappresentativi si identificano in base alle seguenti classi
      - 1 – Valori normali
      - 2 – Valori di confine/limite
      - 3 – Valori speciali
      - 4 – Valori errati
  - 3 – Introduzione di vincoli
    - Una combinazione di valori per le diverse categorie corrisponde alla specifica di un caso di test.  
Spesso il metodo combinatorio genera un gran numero di casi di test (gran parte dei quali magari sono impossibili (Esempio: valore valido, ma non nel database).
    - Introduzione dei vincoli per
      - Eliminare le combinazioni impossibili
      - Ridurre la dimensione di un insieme di test, se questo è troppo grande
- In brevissimo come può essere riassunto?
  - Testo ogni "categoria"
  - Testo i confini tra le categorie
  - Così facendo non si ha garanzia sull'esattezza del codice, ma l'esperienza dimostra che spesso i malfunzionamenti sorgono ai confini.

- **Test strutturale (White Box)**

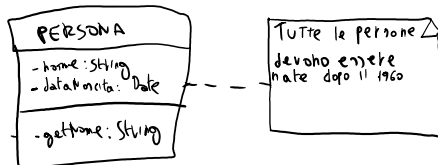
- Casi di test determinati in base a come il componente è implementato in base al codice
- Branch coverage/Copertura delle decisioni - diramazioni:
  - copertura delle diramazioni, degli archi.
  - Consiste nel coprire tutte le possibili frecce/archi del diagramma di flusso.
- Condition coverage/Copertura delle condizioni
  - Consiste nel selezionare un insieme T per cui si percorre ogni diramazione possibile (non sempre tutti i branch sono percorribili) e tutti i possibili valori dei costituenti della condizione che controlla la diramazione sono esercitati almeno una volta → Nel caso in cui si avessero condizioni composte, potrebbe essere che esse vengano coperte tutte solo coprendo tutte le diramazioni.
- Statement coverage/copertura delle istruzioni:
  - Consiste nel coprire tutte le possibili istruzioni, ovvero i riquadri rettangolari e i romboidi, in quanto le condizioni sono comunque istruzioni che restituiscono un boolean.

## • UML

### ○ Class diagram

- Legenda
  - valore[0..1]: int = 0
  - - = private; [0..1] = molteplicità;
  - Dato omissso
    - Visibilità = private;
    - Molteplicità = 1;
    - Tipo = non importa qual è
- Metodi
  - + cambiaVal(nVal: Int)
    - += public
  - Visibilità omessa = public
  - Parametri
    - Per valore e per riferimento
    - Il nome può essere omesso

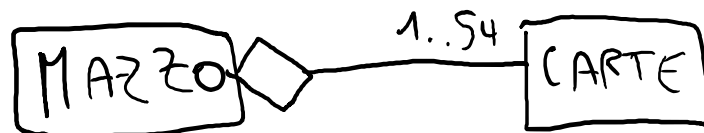
### ▪ Condizioni o Specificazioni



### ▪ Associazioni

Associazioni speciali per indicare che gli oggetti di una classe sono fatti o contengono oggetti di un'altra:

- Aggregazione
  - Una parte è in relazione con un oggetto (part-of).



- Composizione
  - Una relazione di composizione è un'aggregazione forte. Le parti componenti non esistono senza il contenitore. Creazione e distruzione avvengono nel contenitore. Le parti componenti non sono parti di altri oggetti. Un'automobile non esiste senza ruote.



- Specializzazione: sottoclasse



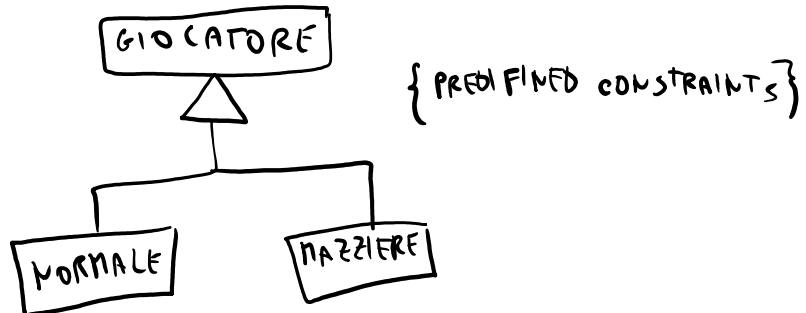


- Navigabilità

- L'associazione è navigabile nelle due direzioni di default, ma se ci interessa solo un verso si può mettere a freccia all'associazione
- Solamente le istanze di mazzo possono mandare messaggi a quelle di carte



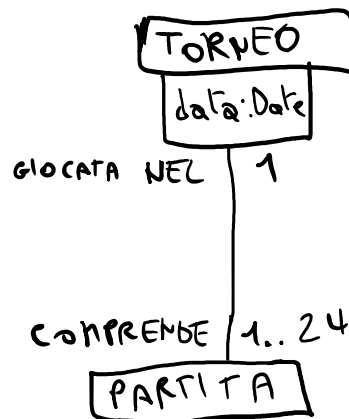
- Specializzazione multipla



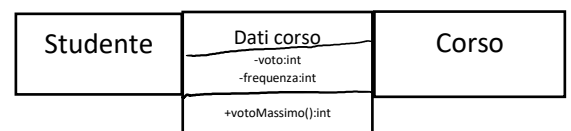
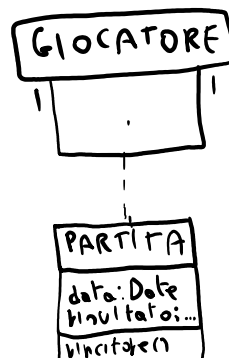
- I predefined constraints sono definiti come di seguito

- Complete – Incomplete
  - Ogni classe è specificata – Non ogni classe è specificata
- Disjoint – Overlapping
  - Sottoclassi sicuramente disgiunte – Sottoclassi possibilmente sovrapposte

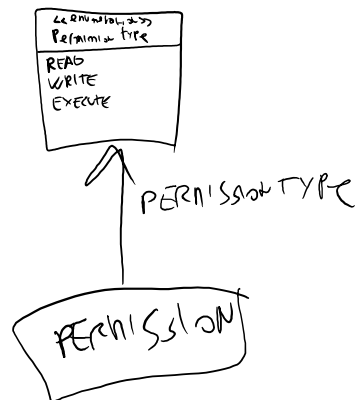
- Association Qualifier



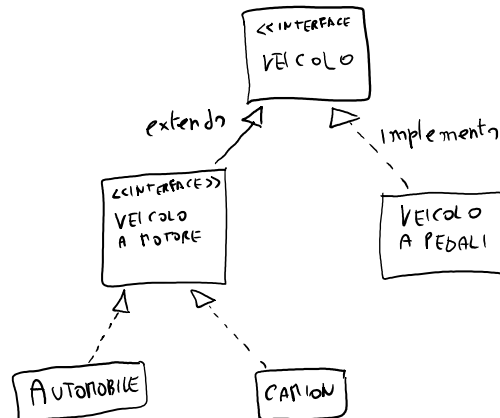
- Association Class



- Enumeration:



- Intefacce in relazione alle ereditarietà



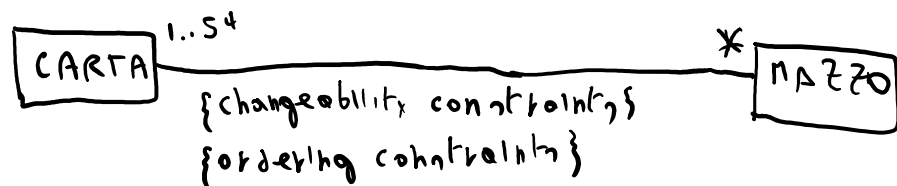
- Constraints di modificabilità

- changeable:
  - le carte associate ad un mazzo possono essere aggiunte e tolte
- frozen:
  - le carte associate ad un mazzo non possono essere aggiunte e tolte.
- addOnly:
  - le carte associate ad un mazzo possono essere solamente aggiunte.
- Se manca è changeable.

- Constraints di ordine

- Ordered:
  - Le carte associate ad un mazzo sono in ordine.
- Unordered:
  - Le carte associate ad un mazzo non sono in ordine.
  - L'ordine non è fissato.

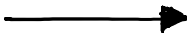



- Se manca è unordered



## ○ OCL (Object Constraint Language)

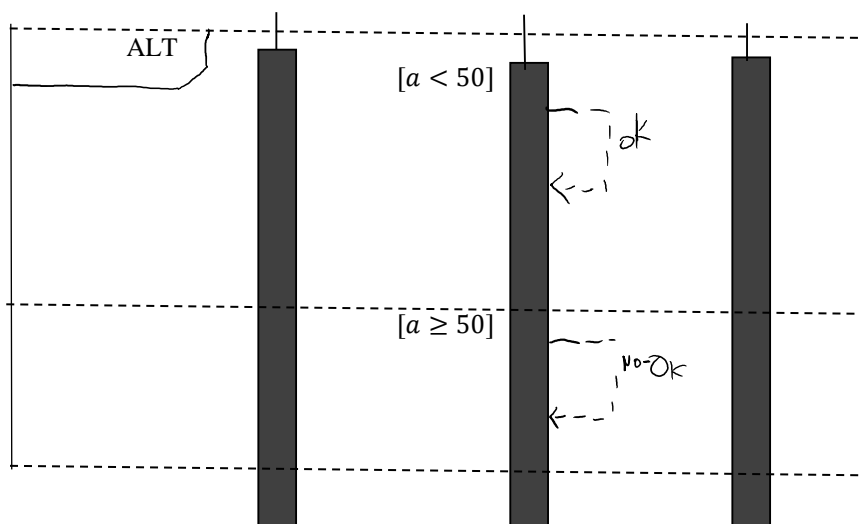
- Se voglio indicare una condizione di una classe scrivo: context Classe inv:
- Esempi
  - Il valore di una carta è compreso tra 1 e 10
    - Context Carta inv: self.valore>0 and self.valore<=11
  - Il vincitore di uno scontro è lo sfidante o lo sfidato
    - context Scontro inv:  
vincitore=sfidante or vincitore=sfidato
  - Una coppia è fatta da due giocatori differenti
    - context c: Coppia inv: primo <> secondo
  - Vince uno scontro la coppia che ha vinto per prima 3 partite  
Ovvero, ci devono essere 3 partite vinte dalla coppia vincitore e le partite sono meno di 6.
    - context Scontro inv:  
partite → size>=3 and partite → size < 6 and partite → exists(P1,P2,P3|  
P1<>P2 and P1<>P3 and P2<>P3 and  
P1.vince = vincitore and  
P2.vince = vincitore and  
P3.vince = vincitore)

## ○ Collaboration Diagram

- Lo salto, mi segno solo i tipo di comunicazione
- Tipi di comunicazione
  - Sincrona
    - A synchronous message is used when the sender waits for the receiver to process the message and return before carrying on with another message
  - Asincrona
    - An asynchronous message is used when the message caller does not wait for the receiver to process the message and return before sending other messages to other objects within the system.
  - Flat
    - Non si precisa se sincrono o asincrono (vado sul sicuro. Principalmente accompagna invocazioni a funzioni)
  - Return
    - Esplicita il ritorno del controllo del flusso al chiamante.

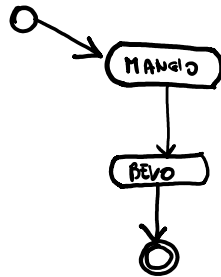
## ○ Sequence Diagram

- I tipi di comunicazione sono gli stessi del Collaboration Diagram
- Lifeline
  - L'oggetto smette di esistere se non aspetta risposta da un altro oggetto
- Condizioni

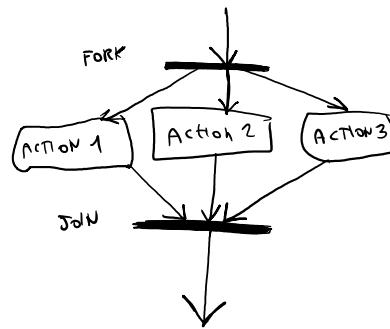


## ○ Activity diagram

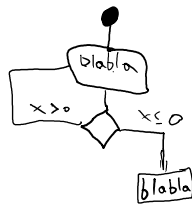
- Decision Point:
  - Rombo splitter che ha la stessa funzione dello XOR di BPMN, ovvero viene eseguita solo la condizione vera ( [condizione] )
- Merge
  - Rombo che riunisce più flussi in un unico
- Esempio



- Fork e Join
  - Servono a descrivere attività in parallelo



- XOR
  - Solo uno dei cammini viene eseguito, in base alla veridicità della condizione.



- **THREAD**

- Metodi

- wait()
  - Rilascia il lock (mutua esclusione) sull'oggetto e sospende il thread che lo invoca in attesa di una notifica.
- notifyAll()
  - Risveglia tutti i thread sospesi sull'oggetto in attesa di notifica. I thread risvegliati competono per acquisire il lock (mutua esclusione) sull'oggetto.
- notify():
  - Risveglia un thread scelto casualmente tra quelli sospesi sull'oggetto in attesa di notifica. Il thread risvegliato compete per acquisire il lock (mutua esclusione) sull'oggetto.
- start()
  - avvia il thread (eseguendo il metodo run)
- join()
  - chiamato su un thread specifico e ha lo scopo di mettere in attesa il thread attualmente in esecuzione fino a quando il thread su cui è stato invocato il metodo join() non termini.
- isAlive()
  - controlla se il thread è vivo (in esecuzione, in attesa o bloccato)
- sleep(int ms)
  - sospende l'esecuzione del thread
- yield()
  - mette temporaneamente in pausa il thread corrente e consente ad altri thread in stato Runnable (qualora ve ne siano) di avere una chance di essere eseguiti.

- I metodi wait(), notify() e notifyAll() devono essere invocati dall'interno di un metodo o blocco sincronizzato.

- Altre nozioni

- I vari thread condividono le risorse/variabili!
- I costruttori non possono essere synchronized
- Altre nozioni

- ```
public class FIFO {
    private boolean empty;
    private String message;
    public synchronized String take() {
        // Wait until message is available
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status
        empty = true;
        // Notify producer that status has changed
        notifyAll();
        return message;
    }
    public synchronized void put(String message) {
        // Wait until message has been retrieved
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status
        empty = false;
        // Store message
        this.message = message;
        // Notify consumer that status has changed
        notifyAll();
    }
}
```

- Uso di synchronized(this)

```
public synchronized void doStuff() {
    System.out.println("synchronized");
}
is equivalent to this:
public void doStuff() {
    //do some stuff for which you do not require synchronization
    synchronized(this) {
        System.out.println("synchronized");
    }
    // perform stuff for which you require synchronization
}
```

In the second snippet you make synchronization lock only for the necessary code rather than making synchronization lock for the entire method.