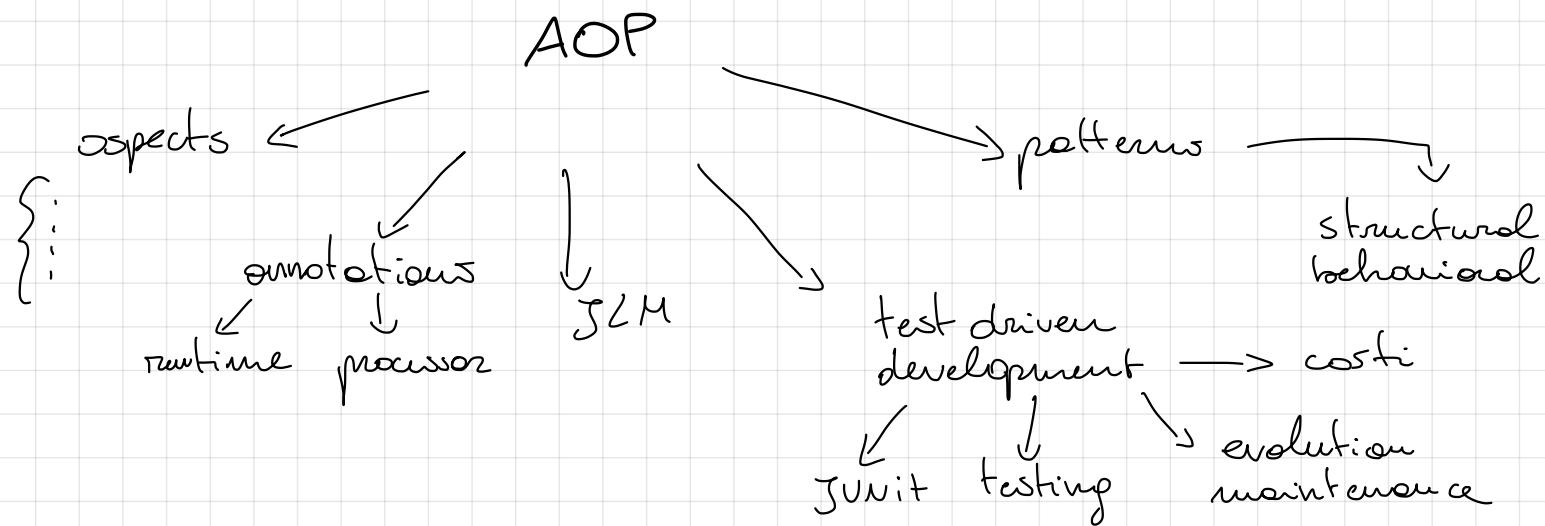
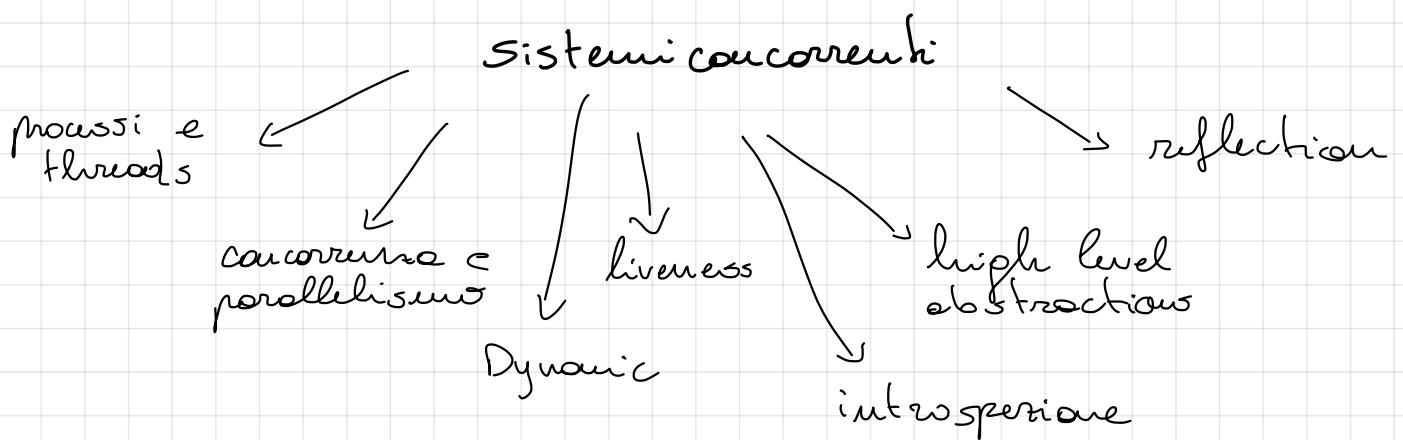
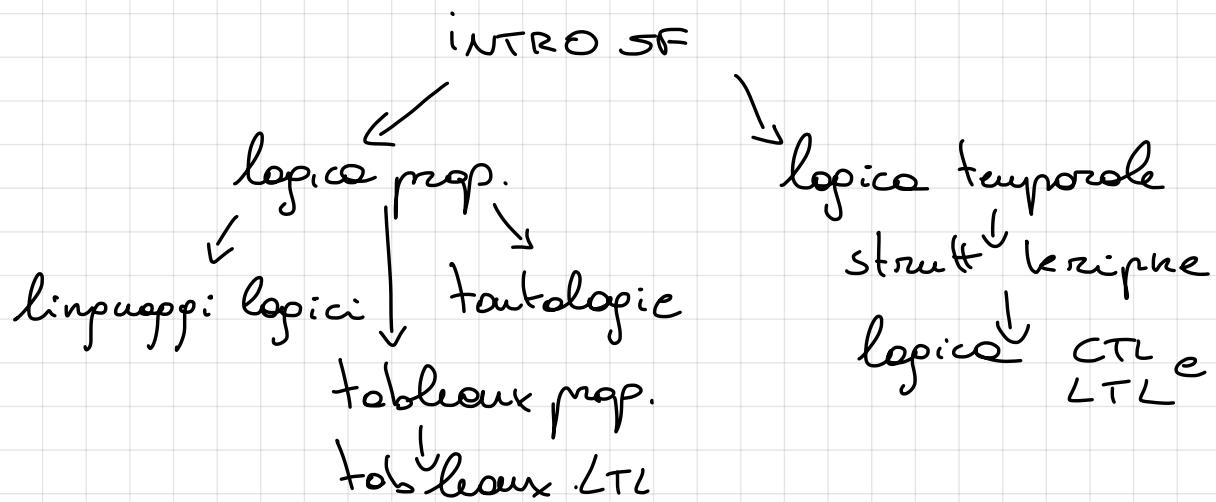


Appunti di Ingegneria del Software

~ Tommaso Pellegrini



PROGRAMMA DEL CORSO



Teoria:

1. Intro SF

Logica proposizionale

- Sintassi: 2 componenti → un alfabeto di simboli
 → un insieme di sequenze di simboli
 AKA linguaggio

* formula ben formata → sintatticamente corretta

↳ simbolo di proposizione

↳ se A è fbf anche $\neg A$ lo è

↳ se A e B sono fbf allora anche $A \wedge B$, $A \vee B$, $A \rightarrow B$ e $A \leftrightarrow B$ lo sono

↳ nient'altro lo è

* Alfabeto: costituito da → insieme di simboli di prop. (p, q, \dots)
 → simboli dei connettivi logici (\neg, \wedge, \dots)
 → parentesi per ambiguità

* connettivi: AND \wedge

OR \vee

NOT \neg

implic. \rightarrow

doppia impl. \leftrightarrow

proposizione: simboli + connettivi

letterale: simbolo prop o la sua

top T e bottom L

- Semantica: vengono associati valori di verità alle formule

tavola di verità

P	Q	$\neg Q$	$P \wedge Q$	$P \vee Q$	$P \leftrightarrow Q$	$P \rightarrow Q$
F	F	V	F	V	V	V
F	V	F	F	V	F	V
V	F	V	F	V	F	F
V	V	F	V	F	V	V

- Soddisfacibile se esiste una valutazione v tc. $v(A) = V$

- Contraddizione se non è soddisfacibile

- Tautologia se per ogni valutazione v si ha $v(A) = V$

$p \wedge \neg p$ è contraddizione, $p \vee \neg p$ è tautologia, p è soddisfacibile

• Logica temporale lineare (LTL)

- Sintassi: dato dai suoi operatori

↓

sintassi minima $\times \text{LTL}$:

$$\varphi := T \mid p \mid -\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid X \varphi$$

X (○)	→ next
F (◊)	→ future
G (□)	→ globally
U (U)	→ until
P (P)	→ precedes
W (W)	→ unless

- Semantica: S insieme finito di stati, so stato iniziale, R relazione di raggiungibilità, P prop. stazionarie, V: $P \times S$ funz. valutaz.

• tableaux

- tableaux semantico:

- tableaux propositionale:

- tableaux LTL

• Struttura di Kripke:

E' una quintupla $K = \langle S, I, R, P, \angle \rangle$ con:

$I \subseteq S$ è un set non vuoto di dati

$R \subseteq S \times S$ è una relazione di accessibilità

P è un set contabile di simboli prop \rightarrow formano Prop [P]

Un percorso π in una K da uno stato $s_0 \in S$ è uno stato infinito di sequenze. Le strutture di K possono essere combinate, lo stato delle componenti si evolve in sincrono.

• Logica CTL:

E' una logica temporale che va oltre lo LTL \rightarrow CTL c'è branching time. Introduce qualificatori di percorso, non permessi in LTL.

$E \ G \ P \rightarrow \exists$ un percorso (da current world) in cui P è sempre vero.

$E \ F \ P \rightarrow \exists$ " in cui P diventerà vero.

$E \ X \ P \quad \exists$ in cui P è vero nel prossimo mondo.

$E \ P \cup Q \quad \exists$ in cui P è vero finché Q diventa vero.

-

$A \ G \ P \rightarrow$ in tutti i comuni successivi, P è sempre vero.

$A \ F \ P \rightarrow$ " P diventerà vero.

$A \ X \ P \rightarrow$ " P è vero fino al prossimo mondo.

$A \ P \cup Q \rightarrow$ " P è vero finché Q diventa vero.

2. Sistemi SW concurrenti

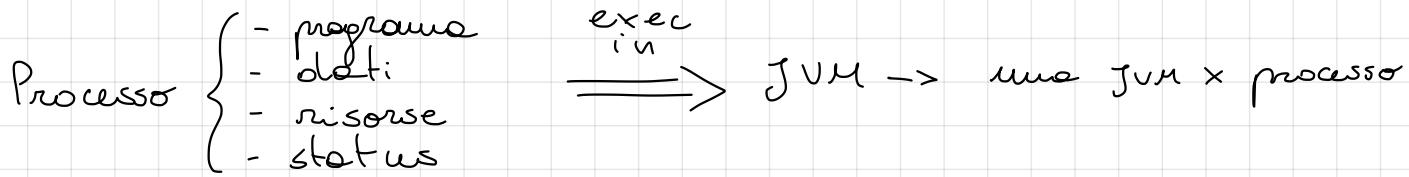
o Processi e threads:

- Processo concorrente → programma concorrente in exec.

↓
V
↳ da statico che descrive il comportamento di un processo a runtime

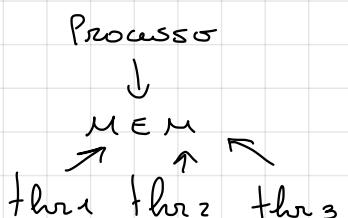
Entità dinamica che esegue un programma usando dati e risorse.

2 o più processi → programma anche con risorse diverse.



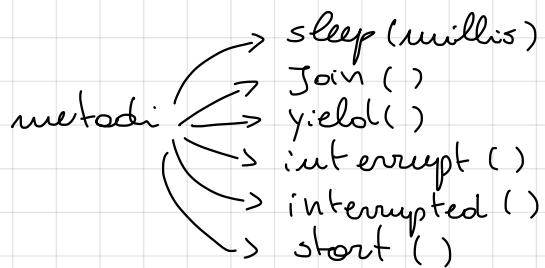
- Thread → execution flow di un processo

- ogni thread è associato a un proc.
- + threads possono \Rightarrow e in solo proc.
- un set di threads è dinamico.
- un thread alloca dinamicamente dati e risorse.
- ha anche dati privati e status.



Hanno le seguenti proprietà:

- Initio in punto specifico del progr. → main thread all'inizio.
- esegue ordinatamente seq. predefinite
- esegue indipendentemente da altri thread.
- appaiono in parallelo anche se interrotti.



o Java memory model

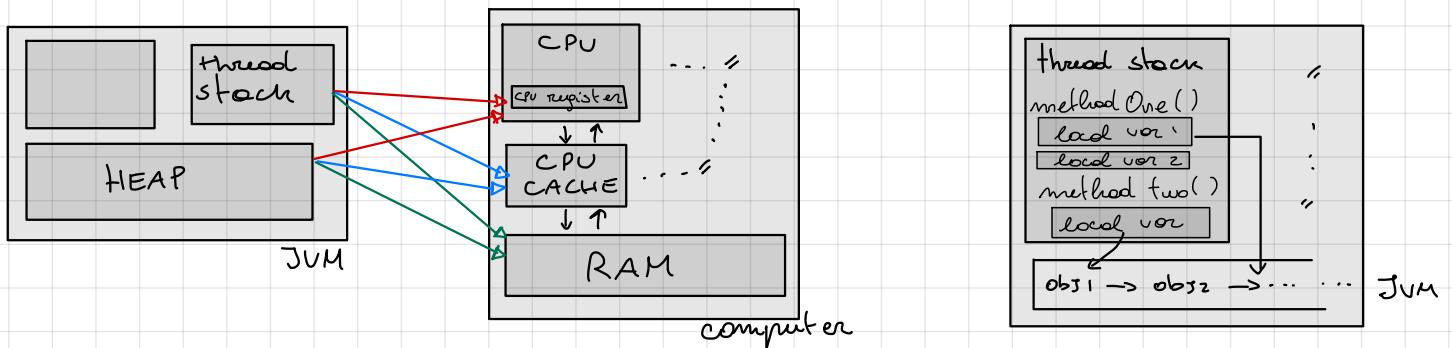
thread → space privato x invocaz. metodi
↳ local-storage privato
↳ condividono heap dell'oggetto

JMM: descrive come i threads accedono alle loro tre memorie

NB: Non per forza i threads sono sullo stesso CPU → sys. distribuito

threads su multi-CPU → memory coherence → accesso a mem controllato

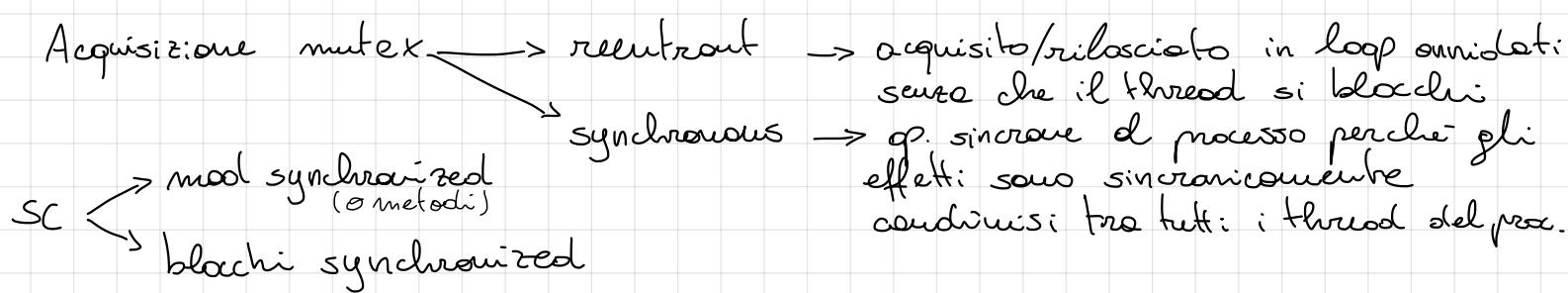
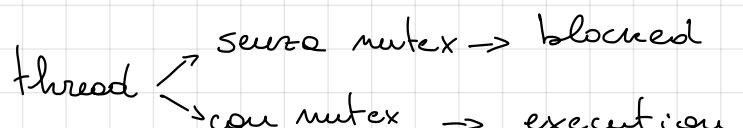
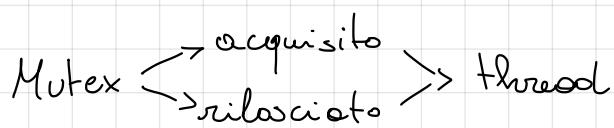
può causare perdite di tempo (busy waiting)



o Mutuo esclusione:

- * Solo un thread alla volta può eseguire sezioni del programma in M
- * I thread che non possono eseguire in M aspettano e riprendono ASAP

Mutuo esclusione \rightarrow Sezione critica $\xrightarrow{(sol.)}$ Mutex \Rightarrow controllo l'accesso.

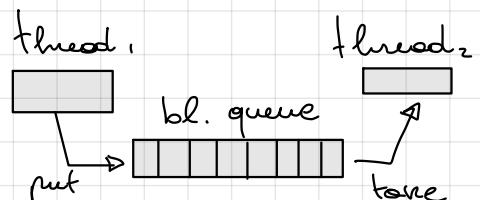


Lo sc evita memory coherence e thread indifference (race condition)

NB: problemi progr. concorrente \rightarrow deadlock, livelock, starvation

o Astrazioni di alto livello:

- **blocking queues**: è una coda FIFO, con oggetti che cambiano dinamicamente
 Operazioni di base: - creazione
 - distruzione
 - enqueue [bloccano con] coda piena/muta
 - dequeue [coda piena/muta]
 - is-full
 - is-empty



- Lock and conditions:

lock \rightarrow astrazione per assicurare mutuo esclusione. lock \equiv mutex (sort of)
 (\hookrightarrow può essere acquisito/rilasciato)
 (\hookrightarrow solo un thread alla volta lo può possedere)
 (\hookrightarrow un thread è bloccato mentre cerca di acquisire il lock.
 viene restartato appena lo prende)

condizione → astrazione per fare waiting per un evento interessante

thread → segue che una condizione è diventata true

→ può andare in waiting finché una condizione viene segnalata
non sempre necessario per fare da guardia a una condizione

lock associato condizione \Rightarrow waiting / signaling su una condizione sotto del thread che possiede il lock

cond. segnalata \rightarrow thread rilascia lock \leftarrow thread che segnala

thread busy in waiting su cond ↑ quando segnalata

- **Atomic references**: astrazione che incapsula una referenza verso un oggetto e ne gestisce la mutua esclusione.

Op. fornite: deref. AR

assegnare AR

deref AR da R e ass. a f(R) ritornando R

"

} tutte op atomiche

" f(R)

- **pool of resources**: prob concorrente $\xrightarrow{\text{cause}}$ risorse condivise $\xrightarrow{\text{sol}}$ gestione corretta

È un insieme di risorse uguali \rightarrow quando create/distrutte sono anche

acquisite/rilasciate tutte insieme
vengono assegnate x richieste \rightarrow accesso controllato garantito

- **thread pools**: come per le risorse \rightarrow creazione / distruzione

acquisiz / rilascio

controllano la creazione e attivazione dei thread \rightarrow

Quando creato \rightarrow attivato immediatamente \rightarrow assicura concorrenza
 \rightarrow immediatamente disponibili

- **executors**: astrazione per far girare più task in maniera concorrente

thread pool $\xrightarrow{\text{assegna}}$ executor $\xrightarrow{\text{exec}}$ threads

executor \rightarrow enqueued di task e fornisce metodi x ritorno risultati o
sono sincroni

3 modi x exec:
one way \rightarrow no modo x garantire exec
exec with callback \rightarrow ritorna risultato x utilizzo
exec with future \rightarrow permette gestione del risultato
per utilizzo futuro

task callback: eseguite quando un executor complete task precedente
→ riceve risultato ed esegue succ. nello stesso thread

Sono implementate come oggetti associati ad una richiesta di exec
Sono dinamicamente associati al risultato delle task

future e future pools: gestisce → risultati di task asincrone
→ eccezioni di task asincrone

Sono implementati come le callback

future $\xrightarrow{\text{block}}$ thread se i risultati embedded non sono ancora pronti,
non bloccante altrimenti

L'accesso ad un set di futures è sequenziale.

Un pool di futures può essere trattato come 1 future con diversi valori
 \hookrightarrow n futures → letto n volte

Correzione esame 11/11/22

- 1) Descrivere il design pattern command, quando serve utile, cosa fa.
Creare un class diagram per semplificare la descrizione.
Scrivere il sorgente di classi e interface definiti nel class diagram.

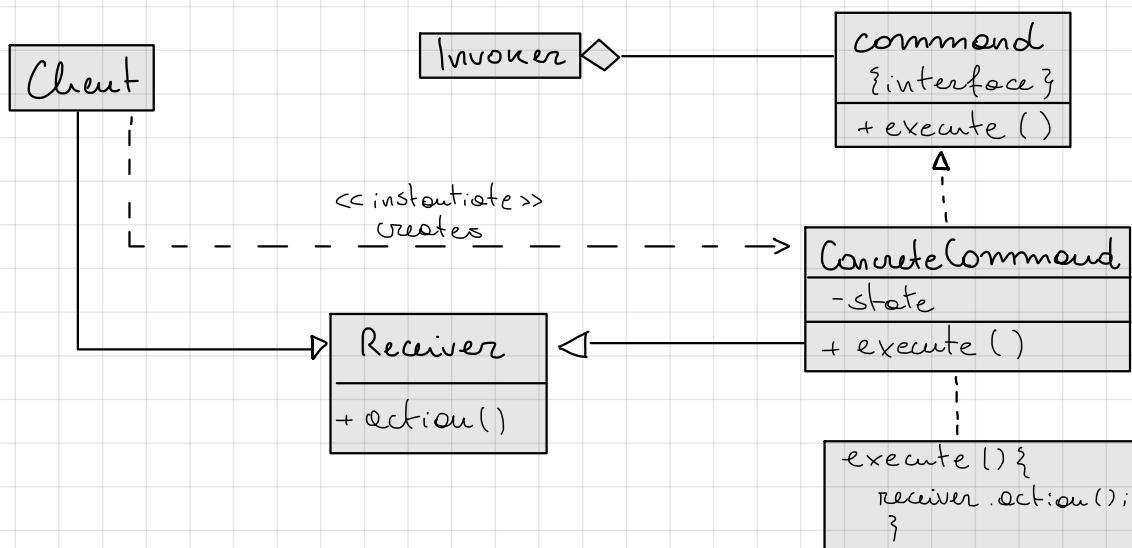
o Command → behavioral design pattern

Command crea un oggetto che incapsula azioni e parametri.

Detto semplicemente, mira ad incapsulare in un oggetto tutti i dati necessari per eseguire un'azione (comando), includendo quali metodi chiudere, gli argomenti e l'oggetto a cui appartengono.

Questo design pattern è utile per disaccoppiare oggetti che producono comandi dai loro consumatori. È infatti conosciuto come producer-consumer pattern.

Diagramma di classe:



Sorgente del diagramma di classe:

```
interface Command {
    public void execute()
}
```

```
class CreateCommand implements Command {
    private ConcreteCommand string;
    public void execute () { receiver.action (); }
}
```

```
class Receiver {
    public void action ();
}
```

```

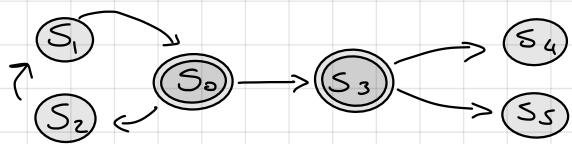
class Client extends Receiver {
    ConcreteCommand CC = new ConcreteCommand ();
}

class Invoker {
    private Command command;
    public Invoker (Command command) {
        this.command = command;
    }
    public void execute () {
        this.command.execute ();
    }
}

```

- Command dichiara un'interfaccia per comandi astratti: tipo execute () ;
- Receiver sa come eseguire un comando particolare .
- Invoker tiene ConcreteCommand , che deve essere eseguito .
- Client crea ConcreteCommand , e assegna Receiver
- ConcreteCommand fa binding fra Command e Receiver

3) Kripke, dato uno stato con simboli { a,b,c } e vero .



- S_0 vero a
- S_1 vero a,b
- S_2 vero c
- S_3 vero a,b,c
- S_4 vero b
- S_5 vero c

Eprimere la quintupla $K = \langle S, I, R, P, L \rangle$, le relazioni di accessibilità, la funzione di labelling.

Dire quali proposizioni CTL* sono vere per K

{ ... }

2) Scrivere il codice sorgente necessario a fare branch testing della seguente classe :

```

class Sorter {
    ...
    ...
    ...
}

```

• Class diagram : (UML)

- Upper section: nome della classe
- Middle section: attributi classe
- Lower section: metodi o operazioni

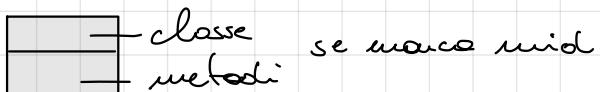
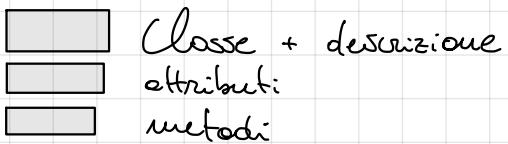
(+) public, (-) private, →
(#) protected, (~) package

ClassName
attributes
methods

Relationships:

- > realizzazione => realizza l'oggetto
- > dipendenza => classi separate con dip. semantiche
- > generalizzazione => superclass -----> subclass
- > associazione => connessione statica/din. fra oggetti
- * multiplicità => più oggetti in dip. con uno
- ◇----- aggregazione => la subclass può esistere senza superclass
- ◆----- composizione => la subclass non può esistere senza superclass

— Bergi version



Con {abstract} può essere
sia interfaccia che classe
estesa

- estende o implementa (upside circle)
- > dipendenza implicita (dipendenza)
- > dipendenza esplicita (associazione)

<<instantiates>> → stereotipo - aggiunge caratteristiche

◇----- la classe del rango contiene oggetti della classe senza rango

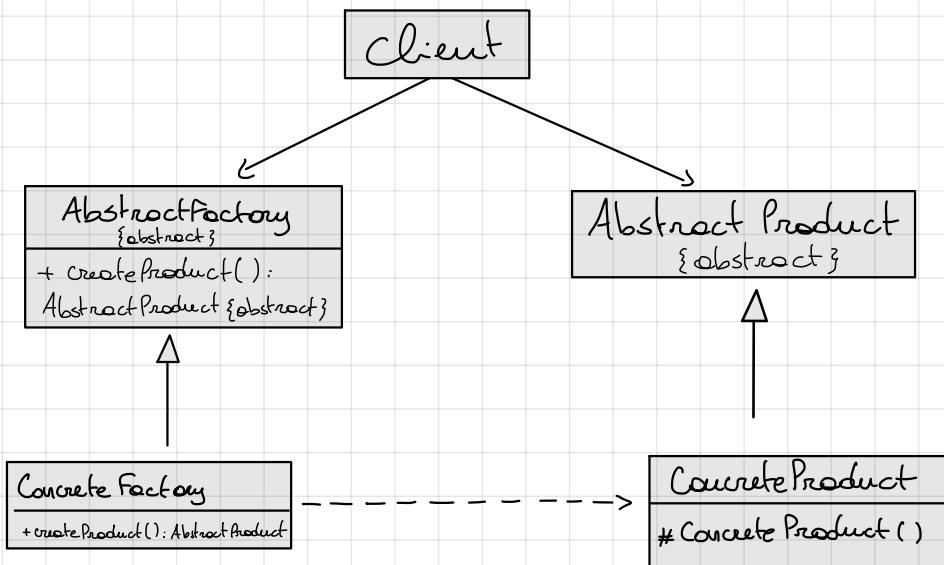
Design Patterns - GOF

- Creational Patterns:

o Abstract factory:

studenti } abstract
libri } factory
biblioteca

factories di classi: gruppo insieme object factories che hanno un tema in comune. Non specifica esplicitamente le loro classi. Un factory object è un oggetto usato per creare altri oggetti.

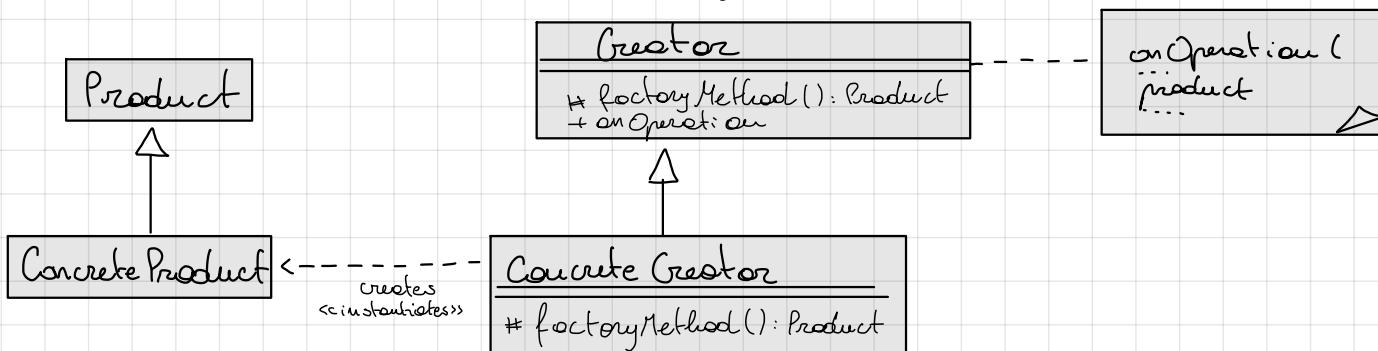


AbstractFactory: è una classe astratta (metodo createProduct abstract) che crea oggetti non di classe factory → genera abstract product che sono usati da concrete factory per creare concrete products.

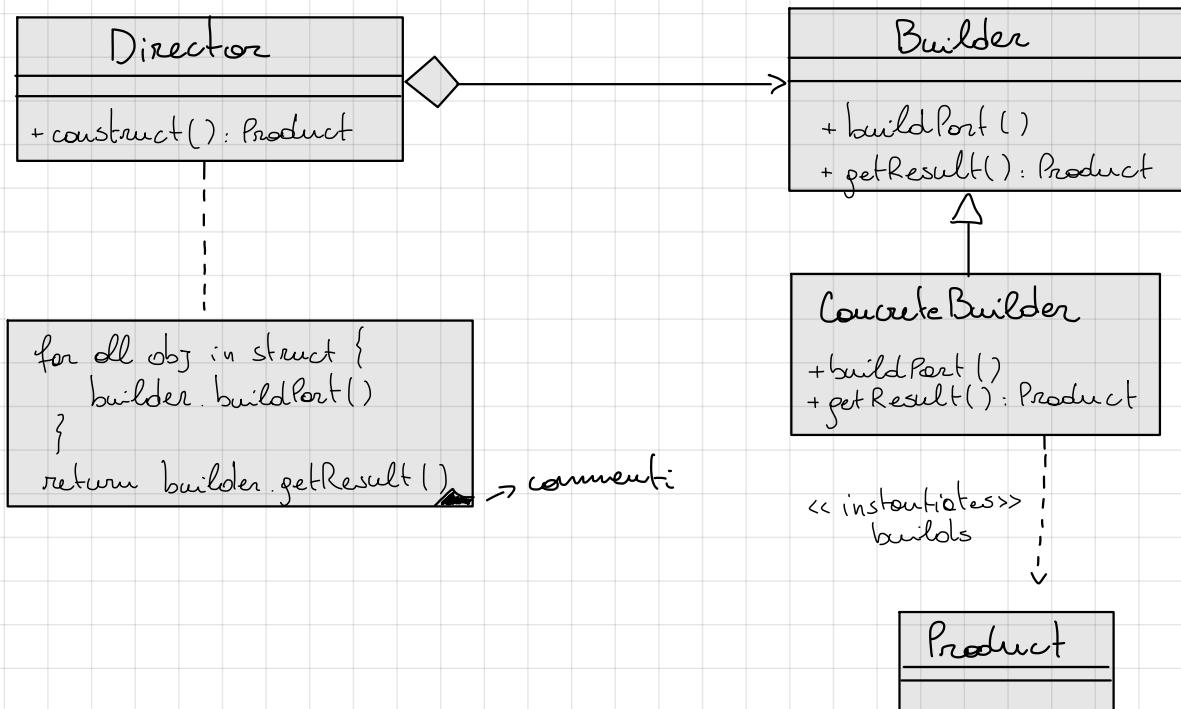
o Factory Method:

crea un oggetto senza specificare la classe esatta che l'oggetto deve avere. È utile per evitare di fare delle new che sono ridondanti. È protetto.

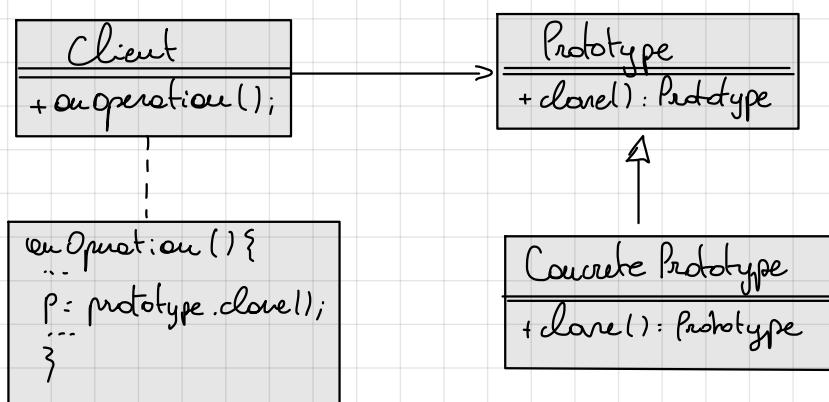
Nascondiamo un metodo per creare oggetti e la classe creator definisce



- **Builder**: costruisce oggetti complessi separando il momento di costruzione dalle loro rappresentazioni.
Sostituiamo l'ereditarietà con la delegation → sost. contenimento



- **Prototype**: È un altro modo per creare oggetti. Anzi che factory o new posso de un prototipo e poi lo clone. A seconda del prototipo nel client lo clone crea oggetti diversi



- **Singleton**:

```

class Singleton {
    -instance: Singleton
    -Singleton()
    +getNewInstance(): Singleton
}
  
```

la classe singololetto è qualcosa che usiamo quando vogliamo che ci sia un solo oggetto di quel tipo.

```

public class Singleton {
    private static volatile Singleton instance;
    public static Singleton getInstance() {
        if (instance == null) {
            ...
        }
    }
}
  
```

```

synchronized ( Singleton . class ) {
    if instance == null
        instance = new Singleton ();
    }
}

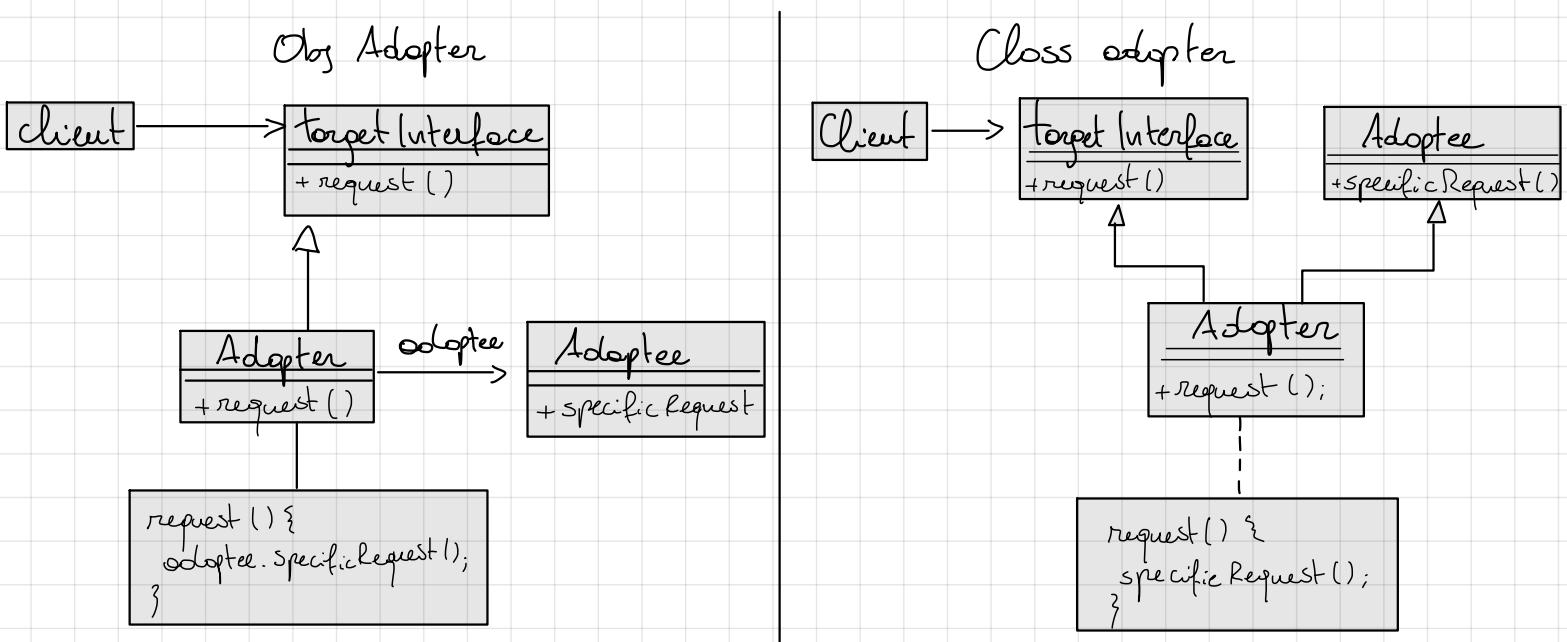
return instance;
private Singleton ();

```

Structural Patterns - GOF

Catégorie delle mini architetture di classi pensate per il problema del riuso staticamente.

- **Adapter:** Conosciuto anche come wrapper, consente a classi con interfacce differenti di lavorare insieme fornendo wrap delle proprie interfacce attorno.

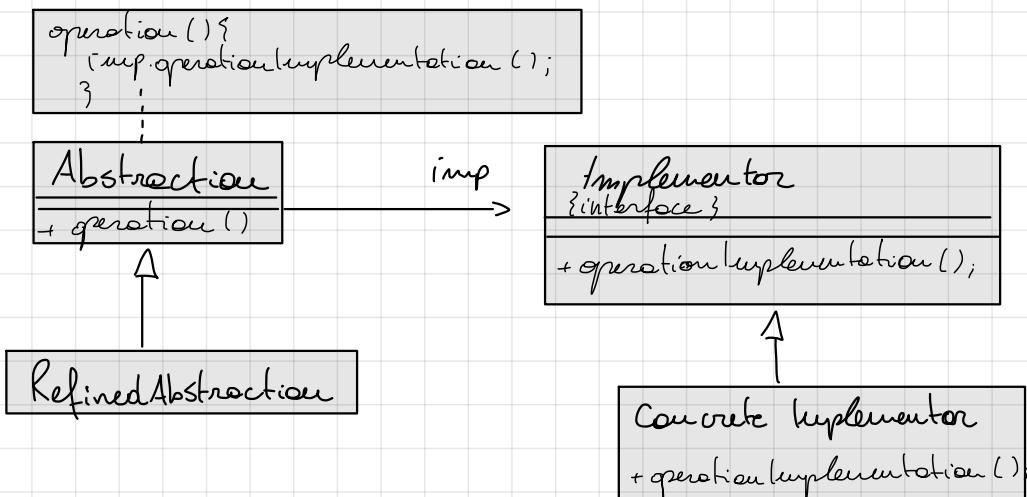


Abbiamo classe con interf, vogliamo combinarne mettiamo un **adptor** che implementa l'originale e gira ogni richiesta di target ad **adoptee**

■ Costruiamo **adptor**, impl **target interface** ed estende **adoptee**.

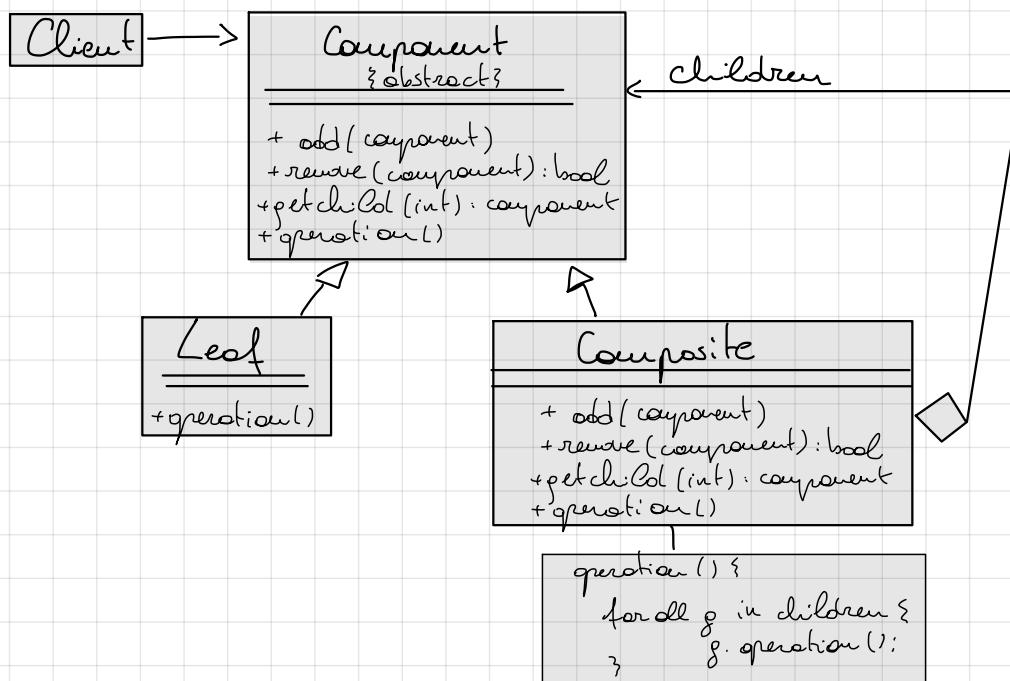
↑ adattore ↑
oggetto ↓ interfaccia

- **Bridge**: simile ad adapter, più flessibile → non vincola le relazioni fra adaptatee e adaptato. Fa le stesse cose ma senza ereditarietà. Abbiamo astrazione → abbiamo impl. → possiamo scegliere a runtime quale impl vogliamo usare.

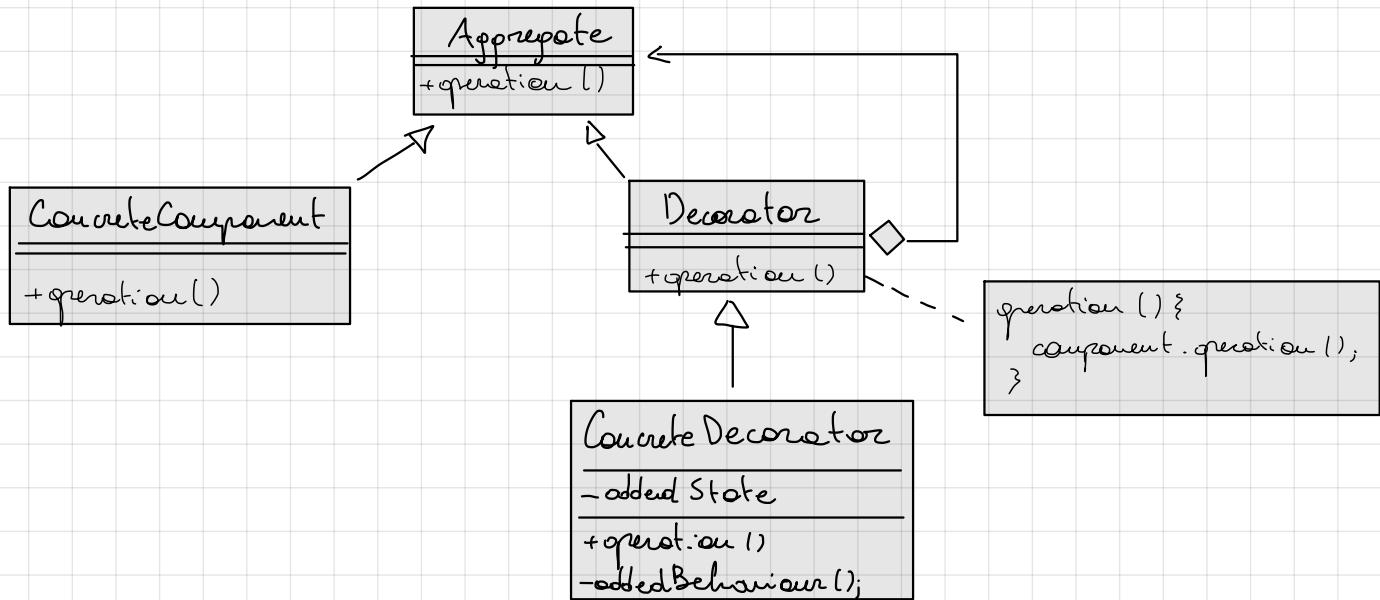


interfaccia verso utente

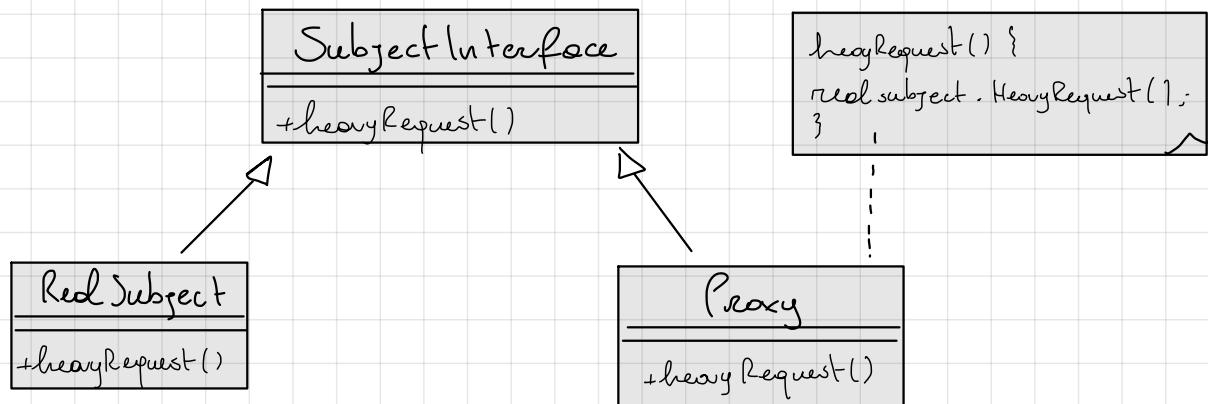
- **Composite**: dati organizzati come alberi → compone zero o più oggetti simili per usarli come un solo oggetto.



- **Decorator**: Simile al composite → al posto di costruire il contenitore ed i contenuti costruiscono i componenti e li decorano con proprietà. Aggiunge o fa override del behavior in un metodo o oggetto esistente.

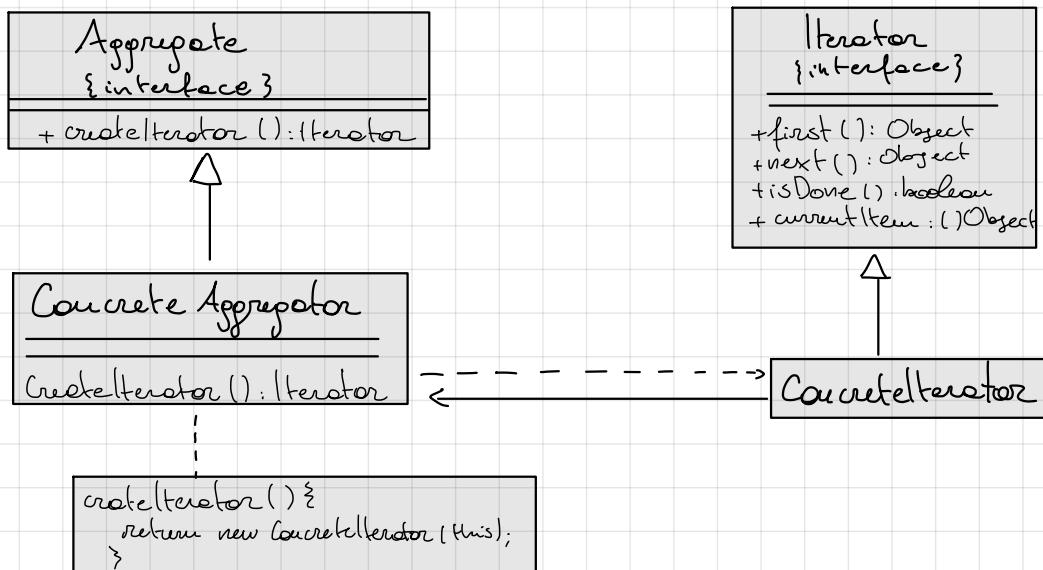


- **Proxy**: fornisce un placeholder per un altro oggetto per controllarne l'accesso e ne riduce la complessità → rende gli oggetti accessibili in modo mediato.



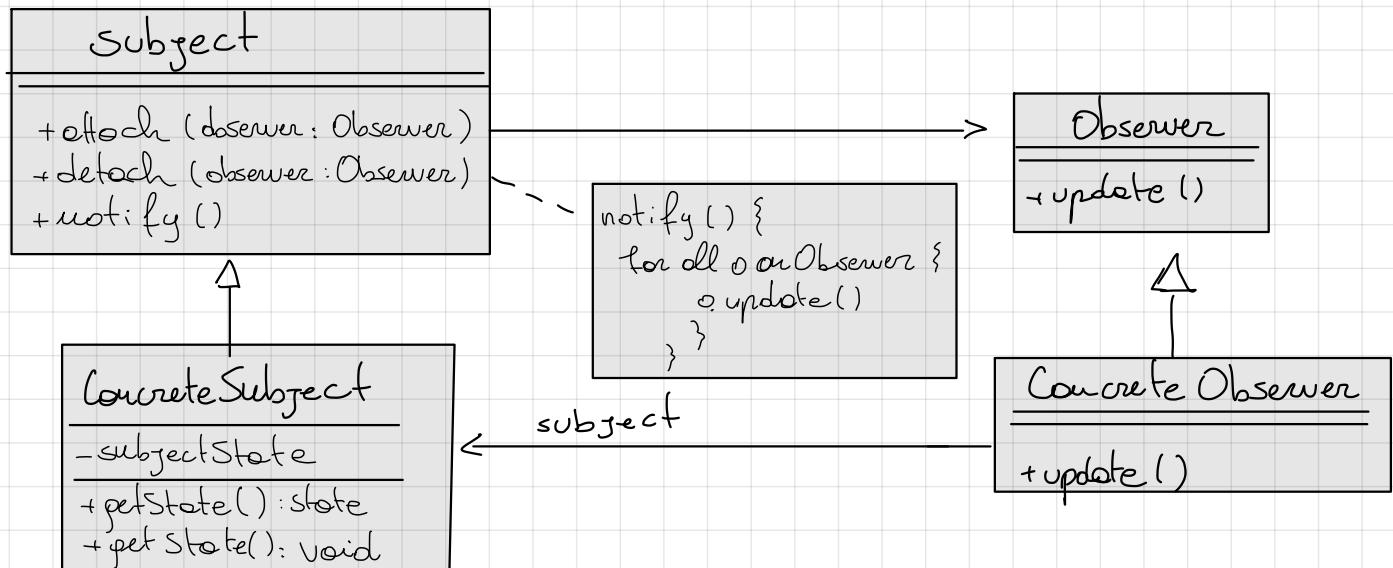
Behavioral Patterns - GOF

- **Iterator**: è un oggetto che permette di attraversare (secondo una data strategie) gli elementi di un oggetto aggregato (alberi ecc).

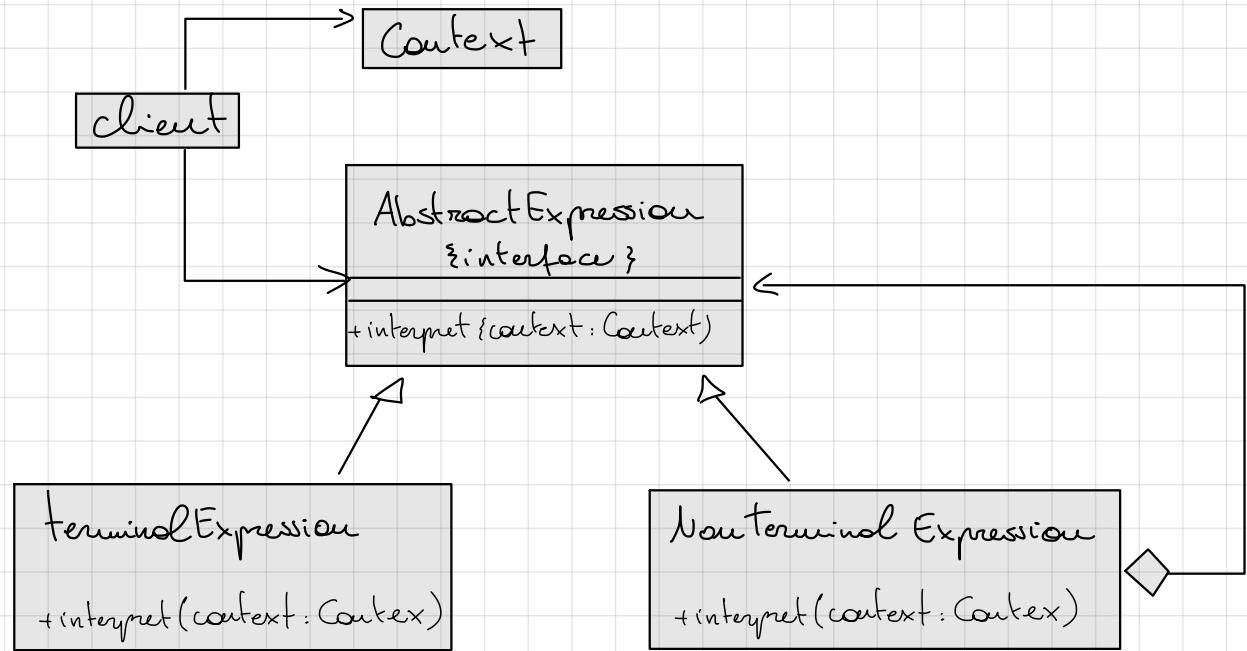


- **Command** → visto sopra

- **Observer**: quando vogliamo osservare lo stato di un oggetto e avere la possibilità di mettere del codice quando lo stato cambia. Permette a un observer di vedere una modifica e reagire.



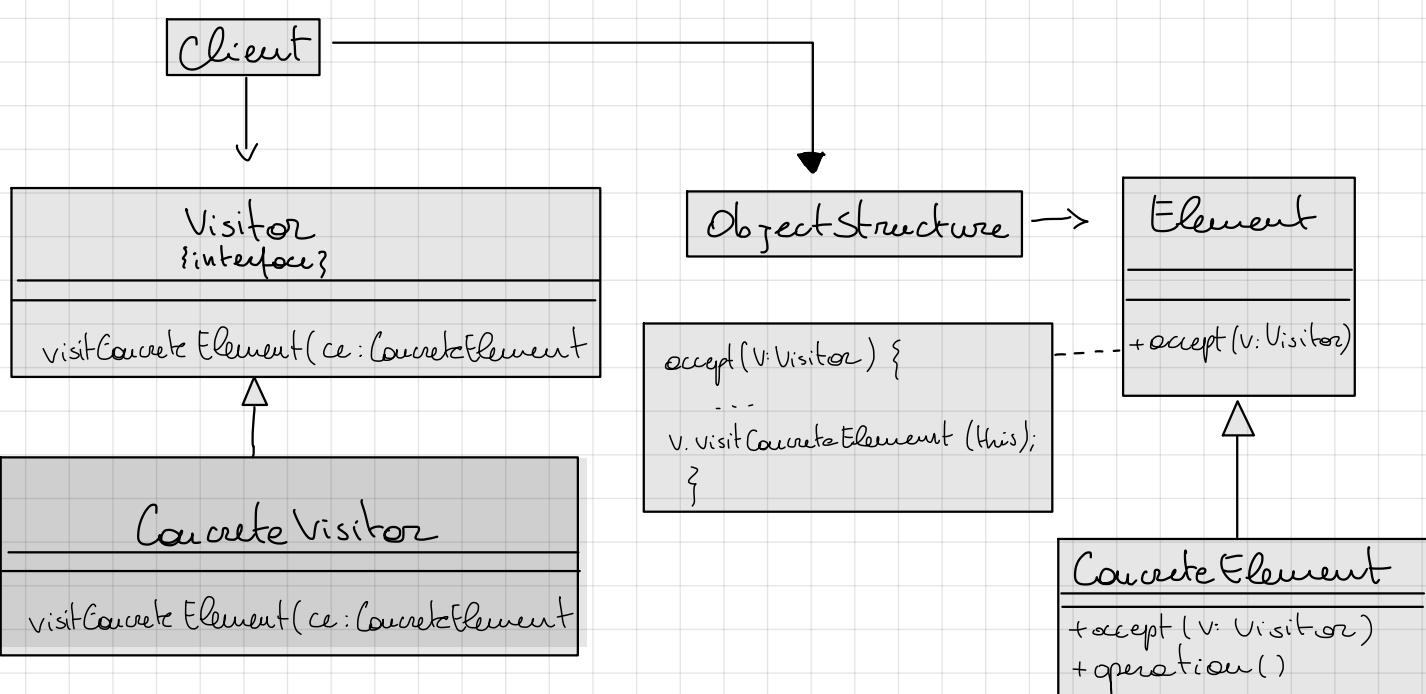
o **Interpreter**: ci dice come fare degli interpreti di linguaggio. Implementa un interprete per un linguaggio specializzato, spesso un DSL. Sono sempre alberi → DMF → testo in albero di sintassi astratta → int



o **Visitor**: realizza una classe che ci permette di visitare una struttura dati ad albero e che ci permette di svincolare la struttura da quello che forse il visitatore.

Separare un algoritmo da una struttura dati muovendo la gerarchia dei metodi in un oggetto.

Avere una struttura che possiamo visitare come vogliamo.



JUNIT - branch and condition testing

- **Statement testing** se una anomalia viene generata da una linea di codice, non vediamo l'anomalia finché non passiamo da quella linea. Le linee di codice che fanno qualcosa sono statement. Vogliamo essere sicuri che tutto il codice che fa calcoli venga controllato.
- **branch testing** spesso una anomalia soltanto fuori uso de uns statement sbaglierà una delle scelte di un certo percorso - branch. Usiamo il branch test per fare incepi di percorrere tutti i percorsi.
- **branch and condition testing** così facendo non solo passiamo per tutti i percorsi ma controlliamo tutte le condizioni che sono presenti all'interno di ogni condizione. Quando facciamo questo tipo di test stiamo facendo il più possibile per controllare che le classi funzionino.

JUNIT → ci permette di creare delle classi che non saranno mai eseguiti dal programma principale, ma testano le funzionalità delle classi che andiamo a testare. Solo classi non interfacce.

```
import static org.junit.Assert.assertEquals;  
import org.junit.Test;
```

```
public class Test {  
    @Test  
    public void getNameNotNull () {  
        Person tester = new Person(); ← check per metodo setter  
        assertEquals("John", tester.getName()); ← creiamo la nuova persona  
        assertEquals("John", tester.getName()); ← controlliamo il metodo  
    }  
}
```

Esempio:

```
public interface Book extends Clonable, Serializable {  
    public int getID();  
    public String getAuthor();  
    public String getTitle();  
    public Book clone();  
}
```

Interfaccia
per SimpleBook

```

public final class SimpleBook implements Book {
    private static final long serialVersionUID = -944174200930040774L;

    private int ID;
    private String author;
    private String title;

    public SimpleBook(int ID, String author, String title) {
        ② if (ID < 1)
            throw new IllegalArgumentException("ID < 1");

        ③ if (author == null || author.length() == 0)
            throw new IllegalArgumentException("author == null || author.length() == 0");

        ④ if (title == null || title.length() == 0)
            throw new IllegalArgumentException("title == null || title.length() == 0");

        this.ID = ID;
        this.author = author;
        this.title = title;
    }

    @Override
    public int getID() {
        return ID;
    }

    public void setID(int ID) {
        this.ID = ID;
    }

    @Override
    public String getAuthor() {
        * ⑤
        return author;
    }

    public void setAuthor(String author) {
        if (author == null || author.length() == 0)
            throw new IllegalArgumentException("author == null || author.length() == 0");
        this.author = author;
    }

    @Override
    public String getTitle() {
        * ⑤
        return title;
    }

    public void setTitle(String title) {
        if (title == null || title.length() == 0)
            throw new IllegalArgumentException("title == null || title.length() == 0");
        this.title = title;
    }

    @Override
    public boolean equals(Object other) {
        if (!(other instanceof SimpleBook))
            return false;
        SimpleBook otherBook = (SimpleBook) other;
        return ID == otherBook.ID && title.equals(otherBook.title) && author.equals(otherBook.author);
    }

    @Override
    public SimpleBook clone() {
        return new SimpleBook(ID, author, title);
    }

    @Override
    public int hashCode() {
        return ID + title.hashCode() + author.hashCode();
    }

    @Override
    public String toString() {
        return "ID=" + ID + ", author=" + author + ", title=" + title;
    }
}

```

- Test Class -

```

package "stesso package"
import static org.junit.Assert.*;

public class SimpleBookTest {

```

lo controlliamo con instance of
← Implementazione

Vogliamo creare il codice di test, affidato al codice che vogliamo testare.

estrai solo il codice che vogliamo testare e lo facciamo in un progetto parallelo con gli stessi pacchetti.

→ New JUnit test case

2 estrazioni: abbiamo tutti i test case e li riorganizziamo in test suites

SimpleBookTest
stesso pacchetto

@Test

② @Test(expected = IllegalArgumentException.class)

```
public void close idLessThanOne() {
```

```
    Book book = new SimpleBook(0, "Auth", "title");  
}
```

③ e ④ Scranno uguali ma con Author e poi title messi a null .

Messi e " "

Perché controlliamo sia null che OR empty + 2 condizioni diverse.

(5) @ Test

```
public void getID() {
```

Book book = new SimpleBook(1, "A", "+"), and we can
assertEqual("A", book.getAuthor()); get author's title

1

→ Creiamo altre 2 classi: dove Author sono: Null e nullo

6

@test

```
public void setAuthor() {
```

sare Null e vuoto

```
SimpleBook book = new SimpleBook(1, "A", "T");  
book.setAuthor("B");  
assertEquals("B", book.getAuthor());
```

14

```
@test(expect = Illegal Argument Exception class)
```

public void setAuthorName(?)

SimpleBook book = new SimpleBook(1, null, "T");

hook . set Aut hor (null) ;
in un altro classe → (" ") :

2

17

buon fine :

@Text

public void equals() {

Book book1 = new SimpleBook(1, "A", "T");

Book book 1 = new Simple Book ("A", "T");
Book book 2 = new Simple Book ("A", "T").

assert Equals (book1, book2);

}

Diversi: @Test

```
public void equalsObject () {  
    Book book1 = new SimpleBook (1, "A", "T");  
    assertEquals (book1, new Object());  
}
```

Le altre 3

@Test

```
public void equalsNotDiffer () {  
    Book book1 = new SimpleBook (1, "A", "T");  
    Book book2 = new SimpleBook (2, "A", "T");  
    assertEquals (book1, book2);  
}
```

→ Uguale con A1 A2
poi T1 T2

Strutture di kripke

$$K = (S, I, R, P, L)$$

$S \rightarrow$ insieme degli stati

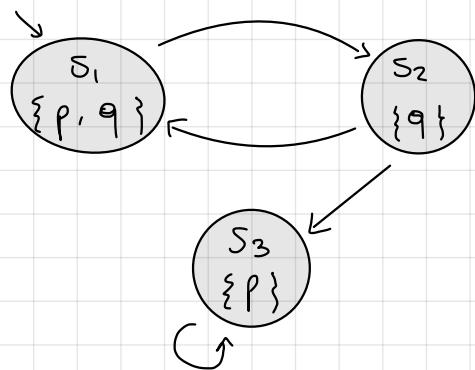
$I \rightarrow$ insieme degli stati iniziali

$R \rightarrow$ insieme delle possibili transizioni fra stati

$P \rightarrow$ insieme delle proposizioni atomiche

$L \rightarrow$ funzione di labeling $L \subseteq S \times P$

Esempio:



$$S = \{S_1, S_2, S_3\}$$

$$I = \{S_1\}$$

$$R = \{(S_1, S_2), (S_2, S_1), (S_2, S_3), (S_3, S_3)\}$$

$$P = \{p, q\}$$

$$L = \{(S_1, \{p, q\}), (S_2, \{q\}), (S_3, \{p\})\}$$

K può produrre un cammino $Q = S_1, S_2, S_1, S_2, S_3, S_3, \dots \in w = \{p, q\},$

$\{q\}, \{p, q\}, \{q\}, \{p\}, \{q\}, \{p\}, \dots$ è l'esecuzione di parole del cammino.

K può produrre esecuzioni $\in \text{ol } L (\{p, q\}, \{q\})^*(\{p\})^\omega \cup (\{p, q\}\{q\})^\omega$

Esempio JUNIT:

AssertTrue (obj instanceof interface) \rightarrow check if implements works

AssertNotTrue () \rightarrow ???

AssertEquals (obj1, obj2) \rightarrow assert per ugualità fra tipi di dato

AssertNotEquals (obj1, obj2) \rightarrow primitive, anche chiavi di fare

@Test (expect IllegalArgumentException.class) {...} \rightarrow caso errato di proposito

↑
apertura f x Junit

LOGICA CTL* + Kripke

Operatori temporali:

- quantificatori di cammino:

$A\phi$: All: ϕ tiene su tutti i cammini dello stato corrente

$E\phi$: Eventually: $\exists!$ 1 cammino dello stato corrente dove ϕ tiene

- quantificatori specifici di cammino:

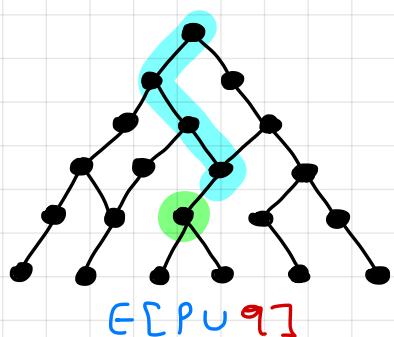
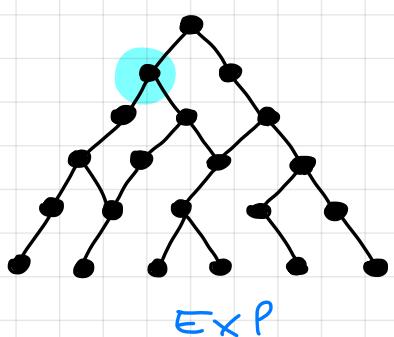
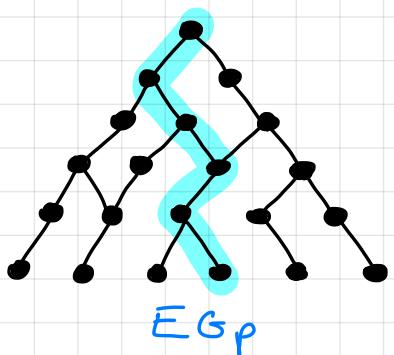
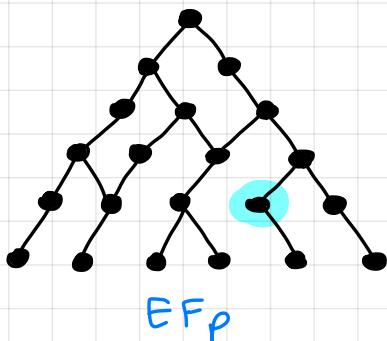
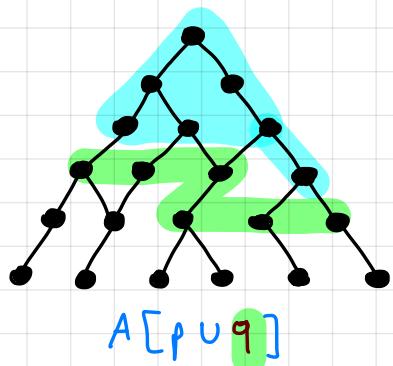
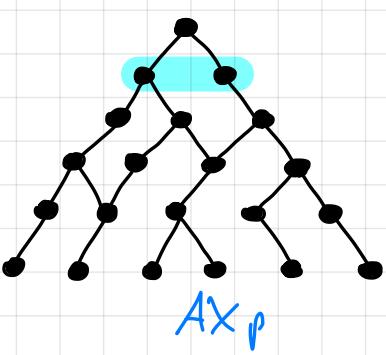
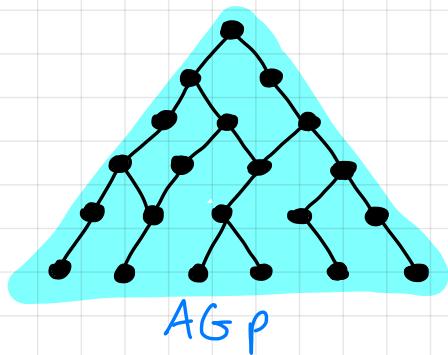
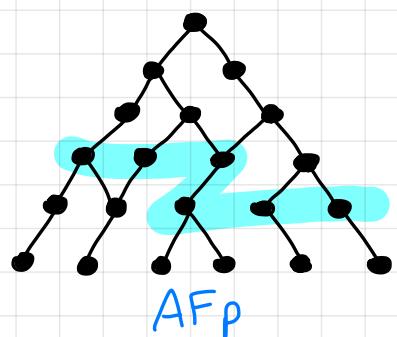
$X\phi$: Next: ϕ deve tenere almeno fino al prossimo stato.

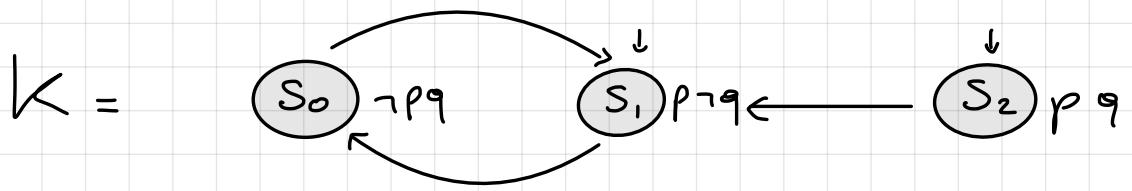
$G\phi$: Globally: ϕ deve tenere per tutti gli stati del prossimo cammino.

$F\phi$: Finally: prima o poi ϕ ferma durante il cammino.

$\phi \cup \psi$: Until: ϕ deve tenere almeno finché ψ inizia a tenere.

$\phi \wedge \psi$: Wore: ϕ deve tenere finché anche ψ tiene.





1. $K \models A F \neg q$ F
2. $K \models E G \neg q$ F
3. $K \models A(p \cup q)$ T
4. $K \models E(p \cup \neg q)$ T

5. $K \models p \cup q$ T
6. $K \models G(\neg p \rightarrow F \neg q)$ T
7. $K \models Gp \rightarrow Gq$ T
8. $K \models FGp$ F

Small Java Concurrency course:

①

- We don't need multicore for concurrency
- Performance → single core → illusion of parallelism (concurrency)
→ multi core → true parallelism

Multithreading programming ≠ single thread programming

Process: isolated from others, contains code, methods, heap, 1+ threads



1+ threads/s → stack → local vars ~ functions
→ instruction pointer

①.2

There are many processes → each containing 1+ threads.

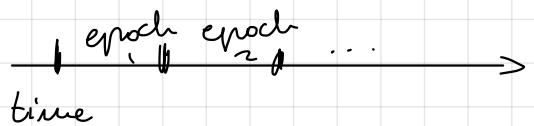
Context switch ⇒ stopping one thread and starting another one.
too many threads → thrashing (wasting time)

OS thread scheduling:

Music player → vi thread, music thread

Text editor → vi thread, save thread

1 core, 4 threads: who runs first?



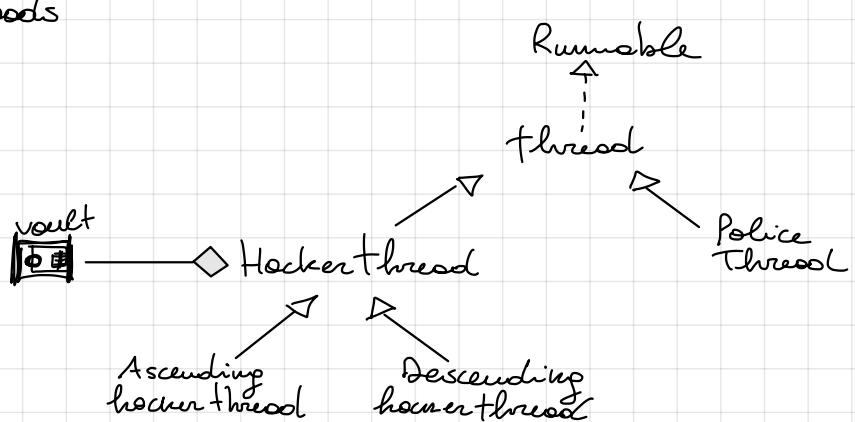
Dynamic priority = static priority + bonus
↓ ↓
user epoch
decided

③ Coding tips

Presente materiale / Downloads

④ + ⑤

Esercizi multithreading



⑥ exercise

⑦ thread termination - why and when

threads consume resource → mem and kernel
→ CPU cycle and cache mem

thread finished working + app still running → clear thread resources

thread misbehaving → stop thread

application won't stop if 1+ threads are running.

- Thread.interrupt():

1. thread executes a method that throws InterruptedException
2. thread code is handling interrupt signal explicitly

if (Thread.currentThread().isInterrupted()) { // controlla interruzione
 System.out.println("interr.")
}

o Daemon threads:

background threads that do not prevent the application from exiting if the main thread terminates.

examples: bg task, not main focus of applications.

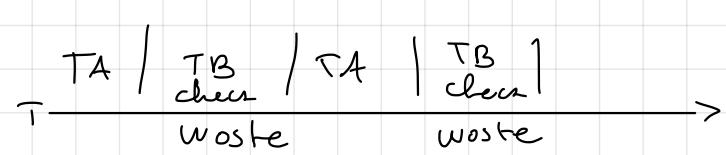
worker thread using external libraries
↳ no control, we don't want it to be a blocker

⇒ Thread.setDaemon(true)

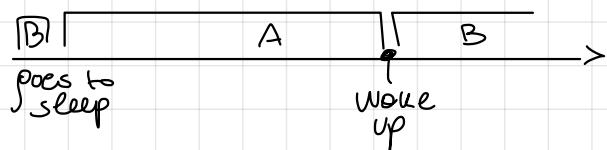
⑧ JOINING THREADS

By default threads run independently and order of execution is out of our control → different scenarios everytime.

thread A → thread B
out in
↑
busy wait check
if done

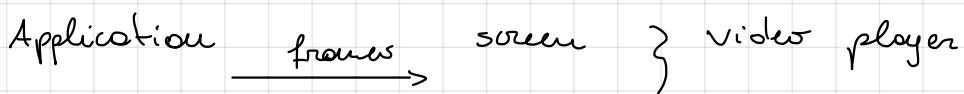
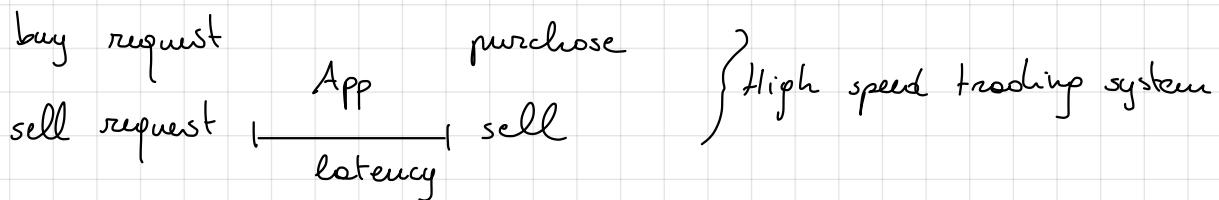


What we went:



(10)

- Performance



Data \rightarrow ML \rightarrow prediction } machine learning

Generally $\begin{cases} \text{latency} = \text{time of completion of a single task} \\ \text{throughput} = \text{amount of task completed in a given period} \end{cases}$

- Latency:
task 1 $N = \text{close to core's number on computer}$
task 2 Adding too many threads is counter-productive.
⋮
task N latency

* With hyperthreading a physical core is splitted in two different virtual cores

NB: simple and short tasks shouldn't be run in parallel.

(11) Image processing → sequential
 → multi-threaded } latency and
 } computing

Image processing app in sequential and multi thread ways.
 Speed up changes when images are heavy. Performances are really improved only when the problem can be broken up in smaller problems.

(12) throughput → completed tasks in a given period = $\frac{\text{tasks}}{\text{time}}$

task 1
 task 2
 task 3 / system] throughput

1 task = T in latency
 ↓

subtask \rightarrow T/N time

- thread pooling => creating threads once and reusing them later
 ↳ maximize throughput

→ by serving each task on a different thread, in parallel, we can improve throughput by $N \rightarrow N = \# \text{ threads} = \# \text{ cores}$

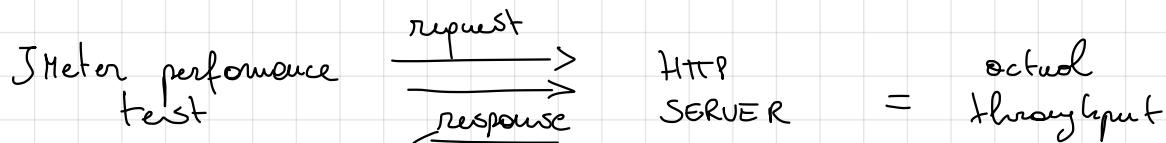
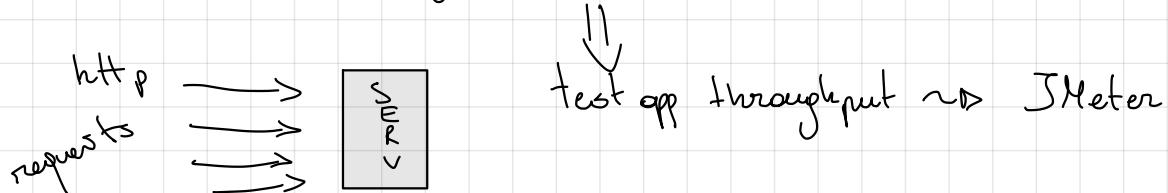
→ by using a thread pool, we maintain a constant number of threads and eliminate the need to re-create them.



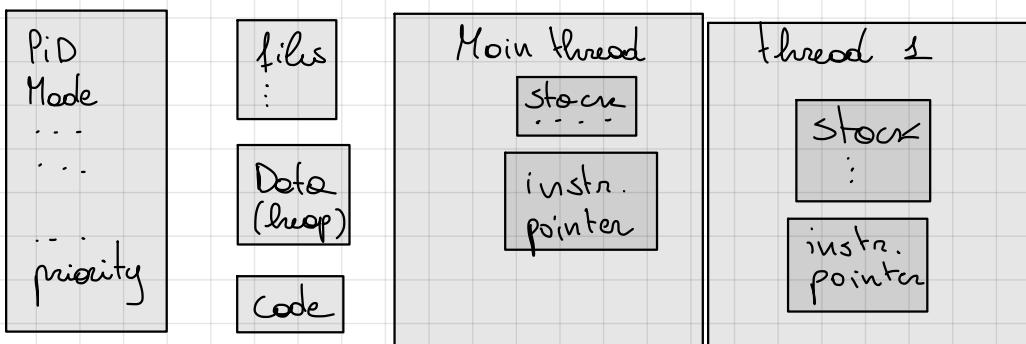
(13) 12 + optimization

(14) Http server → sends flow of http requests as input
 ↳ loads a big book from the disk

app → search engine → search how many times it appears in the novel



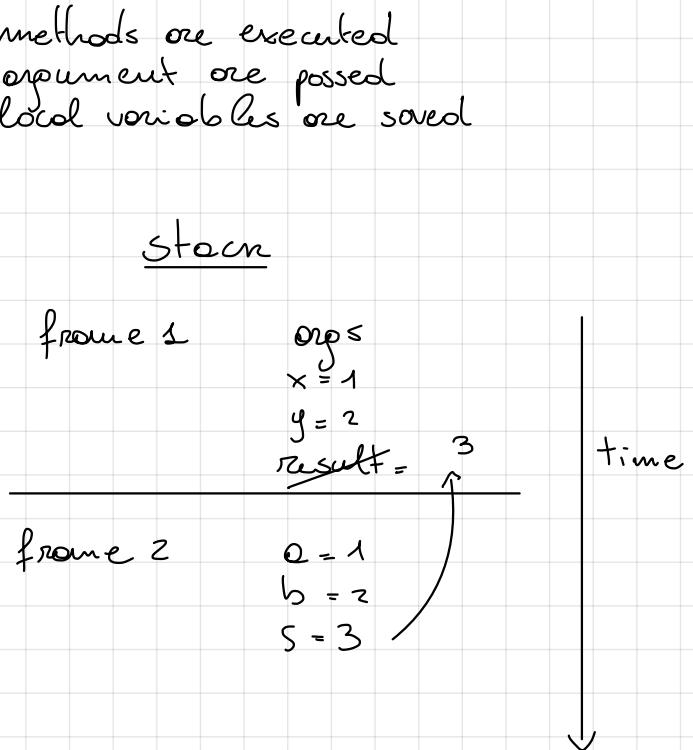
(15) Stack memory region / heap memory region



- Stack: memory region where
 - methods are executed
 - argument are passed
 - local variables are saved
- example:

```
void main(String[] args) {
    int x = 1
    int y = 2
    int res = sum(x, y)
}
```

```
int sum(int x, y) {
    int s = x + y;
    return s;
}
```



◦ Stack properties

- All vars belong to the thread executing on that stack
- statically allocated when the thread is created
- stack's size is fixed, and a bit small
- If calling hierarchy is too deep → stack overflow (risk with recursion)

- Heap:

Objects: strings, obj, collection
 stored in heap

- member of classes
- static variables

NB: a reference is NOT an object!

↓
 Obj refVar1 = new Obj();
 Obj refVar2 = refVar1;

refVar1 → Obj
 refVar2 → Obj



Design pattern with code

1. Creational :

- 1 - abstract factory
- 2 - factory method
- 3 - builder
- 4 - prototype
- 5 - singleton

2. Structural :

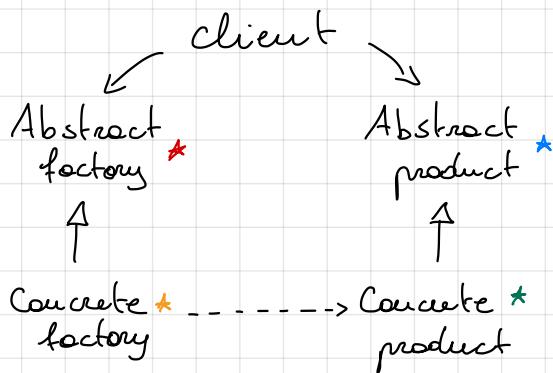
- 1 - adapter
- 2 - bridge
- 3 - composite
- 4 - proxy
- 5 - decorator

3. Behavioural

- 1 - command
- 2 - iterator
- 3 - observer
- 4 - visitor
- 5 - interpreter

1.1 Abstract factory :

roponete factories di oggetti che hanno un tema comune.



* + public abstract class AbstractFactory {
 pub obs AbstractProduct createProduct();
}

* public obs class AbstractProduct {
 pub obs String getName();
}

```
public class Client {  
    private static void stampaNome (prod A prod A) {  
        sysprint (product.getName());  
    }  
    public static void main (String [] args) {  
        Ab factory ob factory = new Concrete factory ();  
        Ab prod product = ob factory . createproduct ();  
        printNames (product);  
    }  
}
```

* pub class Concrete factory extends Abstractfactory {
 public Abstractproduct createproduct () {
 return new Concrete product ();
 }
}

* pub class Concreteproduct extends Abstractproduct {
 public String getName () { return "A"; }
}

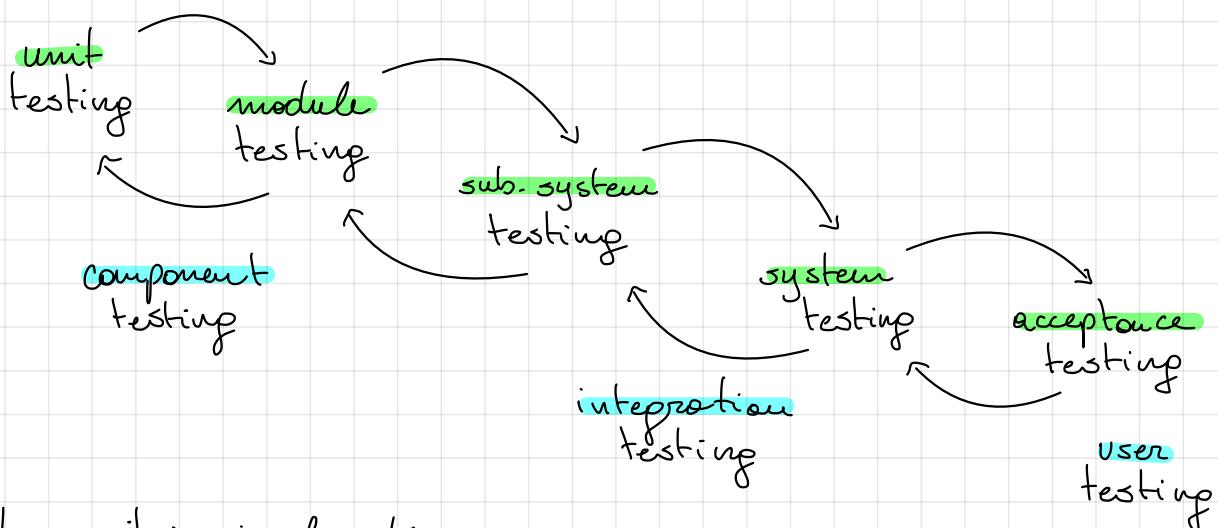
* Continuare nel loro pdf dedicato sulle dp.

Java testing and J-Unit

Fatti sempre veri sul testing:

- Se il comportamento di una parte di un sistema è testato in un numero sufficientemente grande di casi, allora il comportamento di quella parte può essere considerato accettabile anche nei pochi casi rimanenti.
- Il testing può trovare anomalie, ma non può provare che una parte del sistema sia corretta.

testing
in the small → singole parti testate
in the large → sistema intero testato



- **Unit**: unità singole di riuso.
- **Component**: moduli, gruppi di unità inter-dipendenti.
- **Sub-system**: sistemi più piccoli requisiti del sistema.
- **System**: integrazione dei sotto sistemi.
- **Acceptance**: fatto col cliente per assicurarsi che il tutto rispetti le qualità tecniche richieste.

- In the large : **black box** → il test è volto a controllare che il **funzionamento dell'intero sistema sia corretto e si comporti come ci si aspetta**.

Il set di casi di testing è selezionato usando gli artefatti prodotti per specificare i requisiti del sistema.

- In the small : white box non si esaminano parti sufficientemente piccole del sistema ispezionando il codice conseguente.

Il set di casi di testing è selezionato usando il codice considerato. Il set viene normalmente diviso in:

- { - statement testing
- branch testing
- branch and condition testing

- Statement testing: anche detto coverage testing perché basato sul fatto che nessuna parte del codice può essere considerata testata se non viene prima eseguita → una anomalia può essere trovata solo se le parti del codice che la producono vengono eseguite.

Un set di casi T può essere usato per performare statement testing di un codice C se, dopo aver performato tutti i casi di test T, tutti gli statement C vengono eseguiti almeno una volta.

- Branch testing: viene spesso chiamato path coverage perché ha come obiettivo l'analisi di tutti i percorsi di esecuzione del codice considerato. Le anomalie sono spesso causate da percorsi di esecuzione complessi che non possono essere ridotti a percorsi più semplici.

Un set di casi T può essere usato per condurre branch testing di un codice C se dopo aver performato tutti i casi di testing T, tutti i percorsi di esecuzione in C vengono eseguiti almeno una volta.

- Branch and condition testing: spesso chiamato condition coverage testing perché ha come obiettivo l'analisi delle cause che generano percorsi di esecuzione differenti nel codice considerato. Le anomalie spesso vengono generate da percorsi di esecuzione complicati causati da diverse condizioni inter-relazionate.

Un set di casi T può essere usato per fare il testing di un codice C se, dopo tutti i casi T, tutte le condizioni in C sono state considerate e tutte le cause per i loro valori di verità.

ESERCizi ESAMI :

① Si considerino le interface:

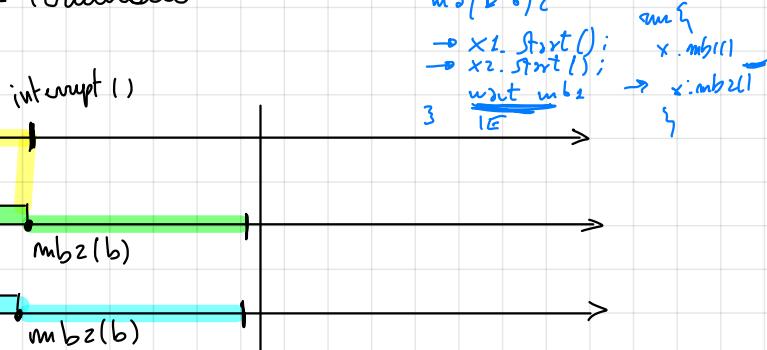
```
public interface B {  
    public void m1();  
    public void m2();  
}
```

A UML sequence diagram illustrating thread interactions:

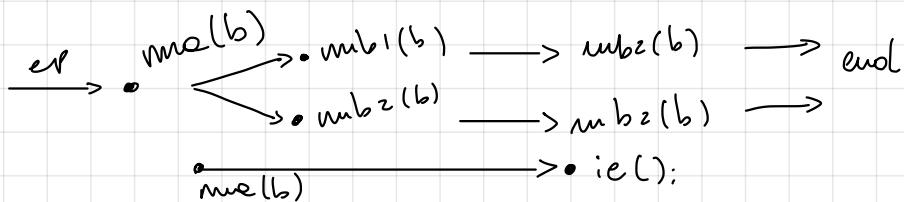
- Participants:** `public class B`, `new T()`, and `wait()`.
- Initial Message:** `new T()` creates a new object `T` at time $t_{1,2}$.
- Parallel Regions:** Two parallel regions are shown, each starting at t_1 and ending at t_2 .
 - The first parallel region contains the message `mb1(b)`.
 - The second parallel region contains the message `mb1(b)`.
- Final Message:** Both parallel regions send a `wait()` message at time t_2 .

Il comportamento della classe `me(B)` è:

- Il metodo inizia creando e attivando due new thread
 - Il metodo mette in attesa che entrano i threads
obiettivo completo sub1() e poi terminare.
 - I due threads si comportano così.
|
 - chiamano sub1() su oggetto di tipo (B)
 - chiamano sub2() su oggetto di tipo (B)
 - terminano



$\mu_{\text{so}}(B)$ può essere chiamato in concorrente \rightarrow moto parallellismo



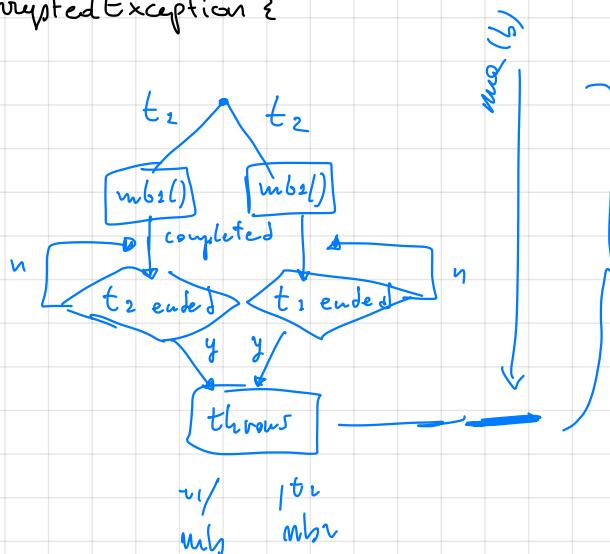
Import java.util.*

public class A1 implements A {

```
public void me(B b) throws InterruptedException {
```

```
thread t1 = new Thread();
Thread t2 = new Thread();
print("Starting")
{
    t1.run();
    t2.run();
    print("...")

    t1.join();
    t2.join();
}
```



Elencare le quattro fasi principali del processo di analisi e specifica dei requisiti indicando una breve descrizione delle fasi stesse e l'insieme degli artefatti prodotti.

Il processo di analisi e specifica fa parte del processo di sviluppo. In questa fase non viene scritto codice e si cerca di comprendere cosa il sistema deve fare, tutti i vincoli dello sviluppo, alla manutenzione ed evoluzione.

Le quattro fasi principali sono:

1. **Studio di fattibilità** → produce un documento di fattibilità, ancora prima di stabilire costi e budget. Si vede se la richiesta è ragionevole. → **report**
2. **Estrazione** (elicitation) ed **analisi dei requisiti**. Questa è la analisi vera e propria, estrae i requisiti interrogando il committente, poi li si analizza, si ottengono i primi modelli del sistema → artefatti da **rifinire**. → **system models**
3. **Specifico dei requisiti** → descrizione dettagliata dei requisiti in modo esauritivo e chiaro, attraverso strutture adeguate. → **User & system requirements**
4. **Validazione dei requisiti**: quando il sistema è stato creato si va a verificare che tutti i requisiti siano stati rispettati. → **requirements document**

Definizione di semantica e sintassi della logica LTL

- **Sintassi**: La sintassi della LTL è definita dai suoi operatori:

X - O	next
F - ◊	finally
G - □	globally
U - U	until
P - P	precedes
W - W	unless

La sintassi minima per LTL è:
 $\varphi = T \mid p \mid \neg \varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \times \varphi$

- **Semantica**: S insieme finito di stati, So stato iniziale, R relazione di raggiungibilità, P propostionali, V: $P \times S$ funz. valutaz.

In Kripke, la semantica è una quintupla $\langle S, So, R, P, V \rangle$

S → l'insieme finito di stati;

So → lo stato iniziale

R → le relazioni di raggiungibilità

P → le proposizioni atomiche

V → le funz. di valutazione $V: P \times S$

Il testing nelle sue fasi:

Analisi e specifico dei requisiti:

Studio di fattibilità → estrazione dei requisiti → specifico dei requisiti → validazione dei requisiti

estrazione → specifico → validazione

Progettazione e realizzazione:

Modularizzazione



Modello di controllo



Decomposizione dei moduli



Diversi livelli di dettaglio

Verifica:

Testing e debug - si controlla che il sistema sia esattamente quello chiesto dal cliente a livello comportamentale e di features.

Mantenzione ed evoluzione:

JUNIT EXAMPLES:

FindMax

```
public class Calc {  
    public static int findMax (int arr[]) {  
        int max = 0;  
        for (int i=1; i<arr.length; i++) {  
            if max < arr[i];  
            { max = arr[i];  
        }  
        return max; } }
```

@Test

```
public void Test () {  
    assertEquals(4, Calc.findMax (new int [] {1, 2, 3, 4}));  
    assertEquals(-1, Calc.findMax (new int [] {-1, -2, -3, -4}));  
}
```

L'idea è usare le assertzioni per controllare che i risultati siano validi e non vengano sostituiti valori sbagliati.

x_1 x_2 risultato
↑ ↑

assertEquals (x , funzione ())

+ Estendiamo findMax {}

```
public static int cube (int n) {  
    return n*n*n;  
}
```

```
public static String reverse (String str) {  
    StringBuilder result = new StringBuilder();  
    StringTokenizer token = new StringTokenizer(str, " ");
```

```
    while (token.hasMoreTokens ())  
        StringBuilder sb = new StringBuilder,  
        sb.append (token.nextToken());  
        sb.reverse ();  
        result.append (sb);  
    } result.append (" ");
```

```
} return result.toString();  
}
```

@Test

```
public void testCube () {  
    assertEquals (27, Calculation.cube (3));  
}
```

@Test

```
public void testReverse () {  
    assertEquals ("ynot", Calc.reverse ("tong"));  
}
```

```
public class Aimpl implements A {
    private Object mutex;
    private boolean stop;
    public Aimpl {
        this.mutex = mutex;
        this.stop = false;
    }
    @Override
    public void run() throws InterruptedException {
        Thread t1 = new Thread();
        Thread t2 = new Thread();

        t1.start();
        t2.start();

        synchronized(mutex) {
            while (stop == true)
                mutex.wait();
        }
    }
}
```

```
public class A extends Thread {
```

```
@Override
public void run() {
    B. sub1();
    B. sub2();
    synchronized(mutex) {
        stop = true;
        mutex.notifyAll();
    }
}
```

Esercizio decoupled:

Un oggetto decoupled garantisce che tutte le chiamate ai suoi metodi pubblici vengono eseguite in thread dedicati. Scrivere il sorponte della classe che usa attach (Object) per dare l'aspetto.

```
import java.lang.reflect { InvocationHandler  
                           InvocationTargetException  
                           Method  
                           Proxy }
```

```
public class GenericAspect {
```

```
    public static <T> T attach (T target) {  
        if (target == null)  
            throw new IllegalArgumentException ("no");  
        //descrittore di classe  
        Class <?> targetClass = target.getClass ();  
        //descrittore interfaccia  
        Class <?> [] targetInterfaces = targetClass.getInterfaces ();  
        //creo proxy  
        Object proxy = Proxy.newProxyInstance (targetClass.getClassLoader (),  
                                             targetInterfaces, new InnerInvocationHandler (target));  
  
        @SuppressWarnings ("unchecked")  
        T result = (T) proxy;  
        return result;  
    }
```

```
public static class InnerInvocationHandler implements InvocationHandler {  
    private Object target;  
    private void invocation (Object target) {  
        this.target = target  
    }
```

```
@Override
```

```
public Object invoke (Object proxy, Method method, Object args [])  
throws Throwable {
```

```
    String methodName = method.getName ();  
    try {  
        //corpo aspect
```

```
} catch
```

Esercizio metodi :

Si considerino le due seguenti interfacce nel package:

Public interface A {
} public void m1() throws InterruptedException;

Public interface B {
} public Object m1();
} public void m2();

Il metodo m1() della classe B ha il seguente comportamento:

1. Il metodo inizia creando un thread in cui viene eseguito il metodo mb2()
2. Il metodo termina immediatamente ritornando un oggetto che può essere usato per mettersi in attesa della terminazione della chiamata al metodo mb2() appena effettuata.

Il metodo m2() della classe B ha il seguente comportamento:

1. Il metodo si mette in attesa per cinque secondi usando thread.sleep(5000).
2. Il metodo termina segnalando la propria terminazione mediante l'oggetto che è stato ritornato dal metodo m1() che ha fatto la chiamata corrente a mb2()

Il metodo m1() di classe A:

1. Il metodo inizia creando tre oggetti di classe Bimpl, che implementa l'interfaccia B, e chiamando il metodo mb1() su ognuno dei tre oggetti.
2. Il metodo prosegue mettendosi in attesa che tutte e tre le chiamate ai metodi mb2() terminino usando i rispettivi valori di ritorno.
3. Il metodo termina.

Public class Bimpl implements B {

 private Object mutex;

 public Bimpl() {
 this.mutex = new Object();
 }

 @Override

 public Object m1() {
 Thread t1 = new Thread(new InnerThread()); start();
 return mutex;
 }

 private class InnerThread implements Runnable {

 @Override

 public void run() {
 m2();
 }

@Override

```
public void sub2() {
    try {
        Thread.sleep(5000);
        synchronized (mutex) {
            mutex.notify();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

public class Aimpl implements A {

@Override

```
public void ma() throws InterruptedException {
    B b1 = new Bimpl();
    B b2 = "";
    B b3 = "";
}
```

```
Object o1 = b1.sub1();
Object o2 = b2.sub1();
Object o3 = b3.sub1();
```

synchronized (o1)

o1.wait();

"

"

}

3 domande ↗ misto
metodi - sempre
logici

Last ↗ aspect
metodi
LTZ formule

Winter ↗ Command DP
JUNIT classe
kerippe
bridge
metodi
LTZ

Formato ↗ Aspect or design pattern
metodi & Junit
logici

LAST 2 ↗ JUNIT
simil-metodi
tableaux

Cosa è stato chiesto nei vari appelli?

5/07 Appello:

③ Descrivere cosa è, a cosa serve e come può essere utilizzato un Java annotation processor → per generare classi di testing JUnit.

Un annotation processor è una feature fornita da Java che apre sul compilatore. Veniamo normalmente usati per controllare le presenze di annotazioni nel codice sorgente e nel caso:

- generare un set di file sorgente
- mettere il codice sorgente corrente
- analizzare il sorgente e generare messaggi di diagnostiche.

Nel caso del JUnit l'annotation processor andrà a leggere le nostre annotations (sia custom che no) per generare dei test case in JUnit per testare l'app.

POST - SUMMER UPDATE

Raccolta domande di teoria (4 appelli)

T₁) Rispondere alle domande seguenti in modo chiaro e ordinato:

- Definire formalmente l'insieme delle prop. $LTL[P]$.
- Definire formalmente la prop. G_I usata per interpretare le prop. LTL .
- Definire formalmente i modelli di una proposizione LTL usando G_I .

T₂) Dimostrare mediante la costruzione di un opportuno tableau che il seguente insieme ben formato delle logiche delle proposizioni, con un insieme di simboli di proposizione $P = \{a, b, c\}$ è insoddisfacibile.
 $\{a \wedge c, a \Rightarrow b, a \Leftrightarrow c, b \Rightarrow \neg c\}$

T₃) Scrivere brevemente cosa è, a cosa serve e come può essere utilizzato un annotation manager del linguaggio Java? Si faccia riferimento nella descrizione dell'uso di un annotation processor per generare classi di test odi classi opportunamente annotate.

T₄) Scrivere brevemente che cosa si intende per invocazione dinamica di metodi in Java mediante il meccanismo delle reflection.
1. Indicare classi e metodi principali coinvolti nell'invocazione dinamica.
2. Scrivere il codice Java per invocare dinamicamente `m1()` su un oggetto che implementa l'interface `A` dell'es 2 passando come argomenti un nuovo oggetto di classe `BImpl` e il valore `s`.

R₁) - L'insieme delle prop. $LTL[P]$ è costruito in tale modo:

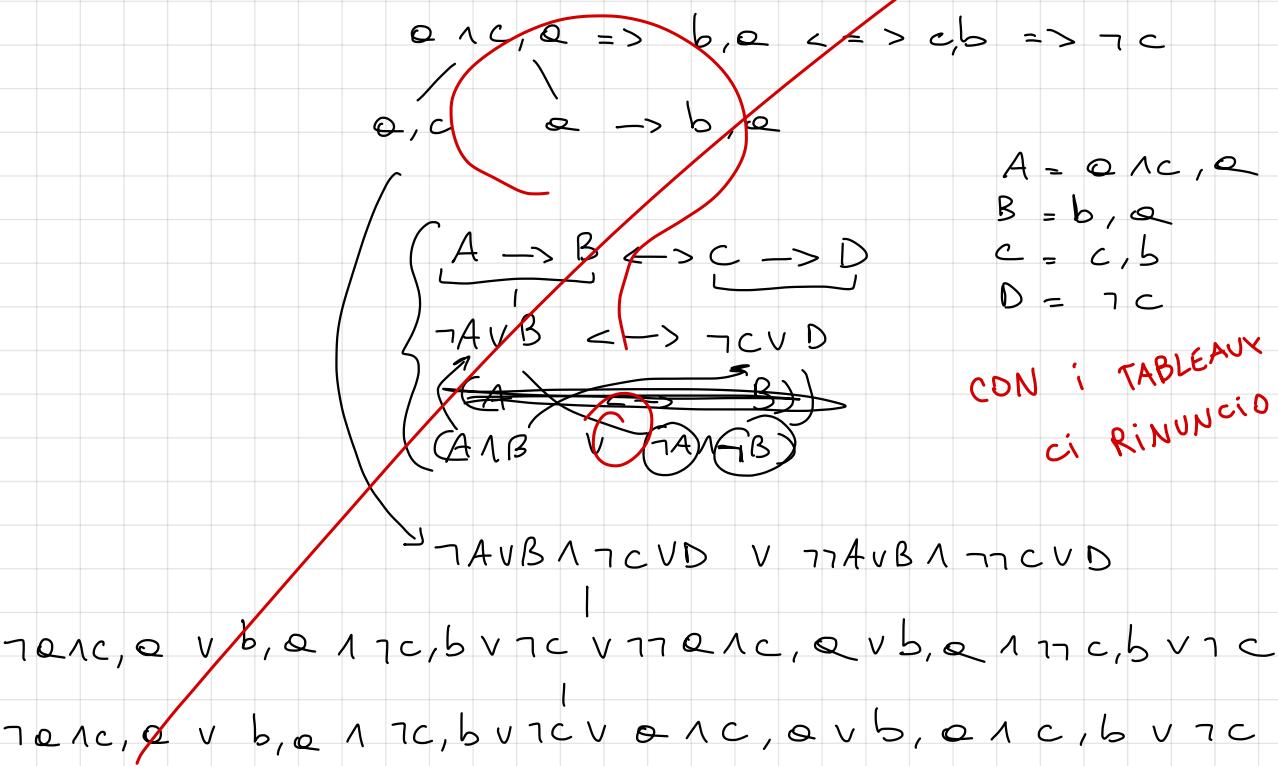
- ogni simbolo prop. è una prop. LTL .
- top T e bottom F sono simboli prop.
- Se A è una prop., $\neg A$ è una prop. LTL .
- Se A e B sono prop., $A \vee B$, $A \wedge B$, $A \rightarrow B$ e $A \equiv B$ sono prop. LTL .
- Se A e B sono prop., $\exists A$, $\forall A$, F_A , $A \cup B$ sono prop. LTL .
- Nient'altro è una prop. LTL .

- La funzione di interpretazione è $I: P \times \mathbb{N} \rightarrow B$ dove $B = \{F, T\}$ mappa ogni simbolo proposizionale a B per ogni momento nel tempo.
Data una interpretazione I su P la funzione G_I è:
$$G_I: LTL[P] \times \mathbb{N} \rightarrow B$$

- Una interpretazione M è modello per P se $\exists! i \in \mathbb{N}$ tc $\langle M, i \rangle \models A$

(R2)

$$P = \{a, b, c\} \quad \{a \wedge c, a \Rightarrow b, a \Leftrightarrow c, b \Rightarrow \neg c\}$$



(R3)

Le annotazioni definite dall'intente sono spesso usate per istruire il compilatore ad eseguire codice fornito dall'intente quando viene processato sorgente con annotazioni.

→ Un annotation processor è un oggetto Java usato dal compilatore per processare codice sorgente sintatticamente corretto.

- Sono definiti dall'intente e:
- facilmente inseribili nel compilatore std.
 - trasparenti al programmatore.
 - + validazione semantic-sintattica.
 - generare altro sorgente da includere.

Si include tramite JAR file → incluso e usato ripetutamente.

Può anche:

- aggiungere output con errori, log, warnings
- aggiungere sorgente all'input della compilazione
- abortire e segnalare errori

Comunemente usati per:

- controllare convenzioni non rinforzate dal compilatore
- generare sorgente aggiuntivo a partire da quello presente

R4 La Java reflection è una funzionalità fornita da un package che permette di postporre alcune decisioni a run-time.

Per invocazione dinamica dei metodi in Java intendiamo la possibilità di poter usare i descrittori di metodo (e cui possono accedere) per invocare il metodo stesso con argomenti corretti. Dato un oggetto e di classe `Class<c>`, un descrittore di metodo m ottenuto da c e un array di oggetti `o`, possiamo usare `m.invoke(o, o)` per invocare il metodo m su o con argomenti o.

Parliamo infatti di polimorfismo quando facciamo sovrivalere di una funz., che sposterà l'invocazione del metodo corretto (di quale versione) a run-time (postponendo la decisione come ci dice la reflection).

Sunto:

1. La logica LTL
2. tableau
3. annotation manager
4. Invocazione dinamica (reflection)
5. Strutt. kripke

Esame 9 Giugno Lab - Condition Set

Sistema software $S = \langle\!\langle \text{conditionObj} \implies \text{ConditionSet} \rangle\!\rangle$

Worker Manager

↓
Singleton che crea oggetti
worker

inner → worker
condSetImpl

$\langle\!\langle \text{ConditionSetImpl} \implies \text{MainCondition}$
 $\langle\!\langle \text{MainCondition} \implies \text{WorkerManager}$
 $\langle\!\langle \text{WorkerManager} \implies \text{Worker}$

interfaccia
base fornita
+ class

Worker ← $\overset{\text{creato}}{\leftarrow} \text{wManager} \rightarrow \overset{\text{associato}}{\rightarrow} \text{condObj}$ (con s.o.w)

wM creata → condsetObj { 1x1 } dentro al CondSetImpl → poi lo usa
 " " worker × mettersi in attesa ciclicamente. Quando
 svegliato ritorna il ref. dell obj notificato
 ogniwo esegue in un thread dedicato
 → 1. attesa 20-40 ms 2. notifica il suo obj e sveglia il WM

Un oggetto condition set ha le caratteristiche:

- Memorizza i riferimenti agli oggetti che gli vengono passati con la add()
- con await() si mette in attesa che 1 degli obj nella coll venga segnalato con notify() o notifyAll()
- quando await() si sblocca ritorna un riferimento all'opp segnalato dalla notify()

condition set < i >
 $\langle\!\langle \text{ConditionSetObject} = \langle\!\langle \text{so workers} = \langle\!\langle \text{so ConditionSetImpl} + \text{Main} \rangle\!\rangle \rangle\!\rangle$
 spauriti da worker manager

Main → crea workermanager.getInstance(50)

- crea ConditionSet = new ConditionSetImpl();
- crea ConditionObject (50) in ciclo
- oppiunge il condObj al condSet

implementa le sue interfacce e ha le funzioni aggiuntive per set/get + stampa

condSet
 ↓
 condSetImpl { (Worker M)
 worker
 condObj + }

ha get/set/print + toString
 e può bloccarsi con un val booleano / sbloccarsi

Atomic References :

Una AR incapsula una referenza ad un oggetto e ne controlla la mutua esclusione.

Sono normalmente previste le seguenti operazioni:

- assegnazione → leggi la referenza embedded
- assegnazione → salvi la referenza
- leggi e sostituisci
- leggi, assegna e ritorna

Executors:

Esegue delle task (usando una pool di resources, opzionale) in concorrenza e permette di associare una pool alle task. Può mettere in coda più task, catturare eccezioni e fornisce 5 modi di esecuzione:

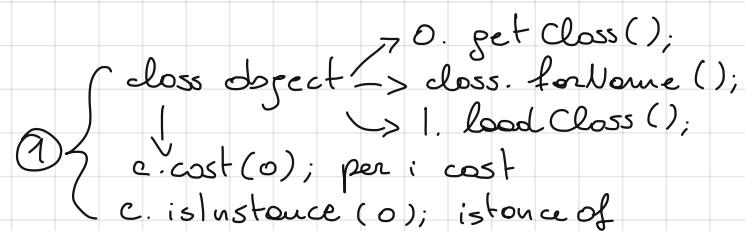
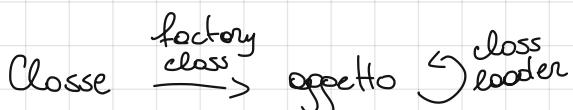
- one way: esegue e ritorna il risultato se c'è
- exec w/ callback: esegue e permette di usare il risultato ritornato in una callback task che ci farà qualcosa.
- exec w/ future: garantisce che il risultato ritornato verrà gestito in qualche modo se presente.

Reflection:

La reflection permette di posticipare le decisioni al runtime:

- Dynamic → linking di classi
→ introspezione
→ creazione di oggetti
→ accesso ai campi
→ invocazione di metodi

- ①
②
③
④
⑤



② Dato c di classe C si può fare introspezione su:

- campi
- costruttori (descrittori)
- metodi (descrittori)
- ottenere referenze al class object della classe
" " " " " della interfaccia

- ⑤ Creare oggetti con `c.newInstance()`; //ref. singleton
Usare `c` per accedere a descrittori del costruttore e invocarli
- ④ Con `f` descrittore di campo per obj c di classe C:
- `f.get(o)` per leggere il valore corrente.
- `f.set(o, v)` per impostare il valore corrente.
- ⑤ Si può accedere ai descrittori di metodi di `c` e usare `m.invoke(o, e)` per invocare il metodo definito da `m` su `o` con argo `e`. (ref. proxy)

Ricordare x Laboratorio:

Singleton

```
public class WorkerManager {  
    private static WorkerManager instance;  
    private WorkerManager () {  
        print ("New Singleton returned");  
    }  
  
    public static WorkerManager getInstance () {  
        if (instance == null) {  
            synchronized (WorkerManager. getClass ()) {  
                if (instance == null) {  
                    instance = new WorkerManager ();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

Inner Thread

```
mythread = new InnerThread (↓)  
mythread.start ();
```

Worker worker

go()

```
public class InnerThread extends Thread {  
    public Worker worker;  
    public InnerThread (Worker worker) {  
        this.worker = worker;  
    }  
}
```

@ Override

```
public void run () {  
    worker.execute ();  
    print ("Thread. currentThread. getID ()");  
}  
}
```