

Ingegneria del Software

Appunti di teoria, by Tommaso Pellegrini

1. Introduzione

1.0 What is Software Engineering?

SF è la costruzione di sistemi software grandi e molto complessi. La dimensione e la complessità del sistema richiedono la cooperazione di diverse persone, spesso organizzate in teams che si occupano di questioni diverse. SF è la diretta conseguenza della sempre incrementale complessità dei sistemi software stessi.

Una possibile definizione:

“L'applicazione sistematica di conoscenze tecnologiche e scientifiche, metodi, esperienza al design, implementazione, testing e documentazione del software.”

1.1 Task principali

Software requirement analysis - si tratta della analisi, specifiche, validazione e requisiti del software.

Software design - è la definizione dell'architettura, delle componenti, interfacce e altre caratteristiche di un sistema, moduli o componenti che siano.

Software development - è l'attività principale di costruzione, la combinazione di programmazione, verifica e debugging.

Software testing - è la investigazione empirica che fornisce

agli stakeholder delle informazioni sulla qualità del prodotto che sta venendo testato.

Software maintenance - ovvero la manutenzione, si riferisce alle attività richieste per fornire un supporto che sia effettivo al costo dopo aver completato un progetto.

Software retirement - comprende le attività richieste per fornire un ritiro del prodotto software che sia effettivo al costo.

1.2 Logica proposizionale

un semplice strumento atto a provare le interessanti proprietà che gli oggetti matematici studiati hanno. Nel software engineering, un tool importante per studiare formalmente le caratteristiche di un sistema software.

$$\begin{array}{c} (p \vee q) \wedge (\neg p \wedge \neg q) \\ \downarrow \\ p \vee q, \neg p \wedge \neg q \\ \downarrow \\ p \vee q, \neg p, \neg q \\ \swarrow \quad \searrow \\ p, \neg p, \neg q \qquad q, \neg p, \neg q \\ \times \qquad \times \end{array}$$

1.2.1 Linguaggi logici I-III

Un linguaggio formale è dato in termini di:

- Sintassi: è il set di formule ben formate.
- Semantica: è l'interpretazione delle formule ben formate a un dominio di discorso.

I linguaggi logici possono essere usati per dimostrare dei teoremi partendo da assiomi e regole di inferenza.

Ci sono linguaggi logici classici:

- Logica proposizionale, per ragionare usando proposizioni e connettivi.
- Logica dei predicati, per ragionare usando termini, predicati, connettivi, quantificatori e variabili.

E ci sono **linguaggi logici modali**.

- **Logica epistemologica**, per ragionare su pensieri e conoscenza.
- **Logica temporale**, per ragionare su relazioni nel tempo.

1.2.2 Logica proposizionale: sintassi e semantica

Il linguaggio si basa sui **simboli proposizionali**. Questi vengono conosciuti come “**atomi**” e possono essere veri o falsi. Es: “Alice ama Bob”.

Le **proposizioni** sono ottenute invece combinando simboli proposizionali con dei connettivi quali \neg , \wedge , \vee , \rightarrow , \equiv .

Un **letterale** invece + un simbolo proposizionale o la sua **negazione**.

Dato un set finito di simboli proposizionali $P \neq \emptyset$ il set $\text{Prop}[P]$ di proposizioni su P è costruito come segue:

- Every propositional symbol is a proposition
- \top (*top*) and \perp (*bottom*) are propositions (it is always assumed that $\top \notin P$ and $\perp \notin P$)
- If A is a proposition, then $\neg A$ is a proposition
- If A and B are proposition, then $A \wedge B$, $A \vee B$, $A \rightarrow B$, $A \equiv B$ are proposition
- Nothing else is a proposition

Convenzioni sintattiche:

- Parentheses can be used to delimit propositions
 - Outer parentheses can be omitted
 - Square brackets and curly braces are also usable, but only parentheses are normally used.
- Binary connectives are *left associative* • $p \wedge q \wedge r$ shortens $((p \wedge q) \wedge r)$.

- The precedence of connectives follows the order \neg ,
 $\wedge, \vee, \rightarrow, \equiv$
- $p \rightarrow q \wedge r$ shortens $(p \rightarrow (q \wedge r))$

La **semantica** della logica proposizionale è rivolta al ragionare sulla verità e la falsità delle proposizioni che vengono analizzate. Un simbolo proposizionale deve per forza essere vero o falso di modo che possa essere computato.

-> Per approfondimenti sulla semantica della logica proposizionale vedi slide da 20 in poi.

1.2.3 Tautologie

Una proposizione A è logicamente valida (o è tautologia) se e solo se $I \models A$ per ogni interpretazioni di I. Se A è una tautologia allora $\models A$ e tutte le interpretazioni sono modelli per A.

Per tutte le proposizioni A e B, le seguenti proposizioni sono tautologie:

- $\neg A \rightarrow A$
- $A \rightarrow (B \rightarrow A)$
- $\neg A \rightarrow (A \rightarrow B)$
- $\perp \rightarrow B$
- $A \vee \neg A$
- $\neg(A \wedge \neg A)$
- $(A \rightarrow B) \rightarrow ((A \rightarrow \neg B) \rightarrow \neg A) - ((A \rightarrow B) \rightarrow A) \rightarrow A$

Una proposizione A è una **contraddizione** se e solo se non ci sono interpretazioni che sono modelli per A. Una proposizione A è una tautologia se e solo se $\neg A$ è una contraddizione oppure è una contraddizione se e solo se $\neg A$ è

una tautologia.

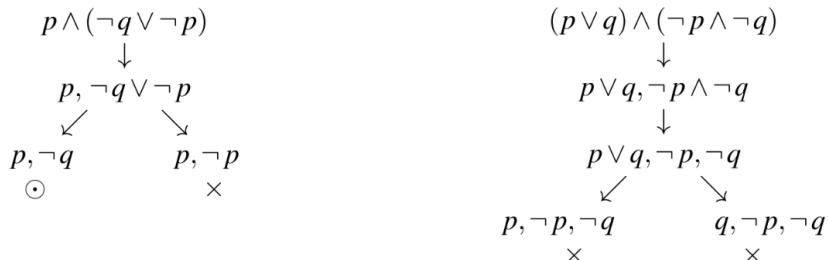
-> per equivalenze logiche, conseguenze logiche, forme negate e forme congiuntive e disgiuntive vedi slide 29->34.

1.3 Tableaux semantico e logica temporale

Un tableau semantico è un albero i cui nodi sono etichettati con set di proposizioni che usano le seguenti regole, finché queste non sono più applicabili:

- An initial set of propositions in negation form labels the root of the tree
- If a leaf of the tree is labeled with $X \cup \{A \wedge B\}$ (for some X, A, and B), add a child node labeled $X \cup \{A, B\}$
- If a leaf of the tree is labeled with $X \cup \{A \vee B\}$ (for some X, A, and B), add two child node labeled $X \cup \{A\}$ and $X \cup \{B\}$
- If a leaf of the tree is labeled with $X \cup \{P, \neg P\}$ or $X \cup \{\perp\}$ (for some X and P), add a child node labeled with \emptyset

Un set di proposizioni S è non soddisfacibile SSE tutte le foglie del tableau sono etichettate con set vuoti.



1.3.1 Logica temporale

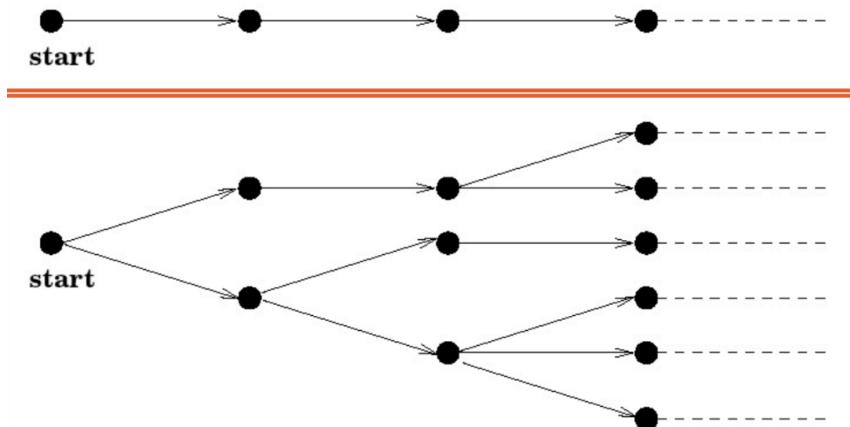
Nella logica classica, le proposizioni sono interpretate statisticamente.

I sistemi software sono caratterizzati da stati che cambiano nel tempo, quindi la logica classica non è sufficiente.

Nella logica classica infatti le proposizioni sono interpretate nello scope di un singolo mondo statico, mentre nella logica temporale le interpretazioni prendono posto nello scope di un set di mondi.

Il set di mondi che caratterizzano la logica temporale corrisponde a momenti nel tempo: il modello particolare del tempo data una logica temporale è catturata da una relazione di accessibilità fra i mondi.

La logica temporale estende quella proposizionale con operatori temporali (modali) che navigano i mondi usando le relazioni di accessibilità della logica scelta.



-> per approfondimenti sulla logica temporale vedi slide 42-51.

-> per modelli ed equivalenze logiche vedi slide 52-54.

1.4 Tableaux Proposizionale e tableaux LTL

1.4.1 Tableaux proposizionale

I tableaux proposizionali sono alberi usati per ragionare su un set di proposizioni:

I nodi del tableau sono etichettati con set di proposizioni in forma negata.

Due regole vengono usate per espandere il tableau aggiungendo figli ai nodi foglia con la (\wedge -rule and \vee -rule).

Una regola (\neg -rule) viene usata per identificare foglie contraddittorie.

Un tableau è completo se non ci sono più regole applicabili.

Le foglie (contraddittorie o non) di un tableau completo sono etichettate con set di letterali perché le regole di espansione rimuovono i connettivi dalle proposizioni.

I tableaux proposizionali sono usati per controllare la soddisfaccibilità.

Un percorso dalla radice di un tableau completo ad una foglia è *chiuso* se la foglia è contraddittoria, altrimenti è *aperto*.

Un tableau è chiuso se tutti i suoi percorsi sono chiusi.

L'etichetta di una foglia di un percorso aperto identifica un set di modelli del sett di proposizioni che etichettano la radice.

1.4.2 LTL Tableaux

Questo tipo di tab è un grafo diretto (non è un albero) e viene usato per controllare la soddisfaccibilità di un set di proposizioni LTL.

Le proposizioni LTL nel set da analizzare sono convertite nella loro forma negata usando le seguenti equivalenze logiche:

- $A \equiv B \leftrightarrow (A \rightarrow B) \wedge (B \rightarrow A)$
- $A \rightarrow B \leftrightarrow \neg A \vee B$
- $\neg \neg A \leftrightarrow A$, $\neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$, $\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$
- $\neg X A \leftrightarrow X \neg A$, $\neg F A \leftrightarrow G \neg A$, $\neg G A \leftrightarrow F \neg A$, $\neg(A \cup B) \leftrightarrow (\neg B \cup (\neg A \wedge \neg B)) \vee G \neg B$

Le seguenti regole proposizionali sono applicate a partire dalla radice del tableau etichettato con il set di formule LTL negate:

- If a leaf of the tableaux is labeled with $S \cup \{ P, \neg P \}$ or $S \cup \{ \perp \}$ (for some S and P), stop expanding the leaf
- If a leaf of the tableaux is labeled with $S \cup \{ A \wedge B \}$ (for some S , A , and B), add a child node labeled $S \cup \{ A, B \}$
- If a leaf of the tableaux is labeled with $S \cup \{ A \vee B \}$ (for some S , A , and B), add two child nodes labeled $S \cup \{ A \}$ and $S \cup \{ B \}$

Le seguenti regole temporali sono applicate a partire dalla radice etichettata con il set di formule LTL nella forma negata:

- If a leaf of the tableaux is labeled with $S \cup \{ G A \}$ (for some S and A), add one child node labeled $S \cup \{ A, X G A \}$
- If a leaf of the tableaux is labeled with $S \cup \{ F A \}$ (for some S and A), add two children labeled $S \cup \{ A \}$ and $S \cup \{ X F A \}$
- If a leaf of the tableaux is labeled with $S \cup \{ A \cup B \}$ (for some S , A , and B), add two children labeled $S \cup \{ B \}$ and $S \cup \{ A, X(A \cup B) \}$

Le seguenti regole di loop sono applicate se nessun'altra regola è applicabile partendo dalla radice del tab etichettata

con il set di formule LTL nella loro forma negata:

- If a leaf node is labeled only with literals and LTL propositions structured as XP , for some P , then
 - If the set S' of the propositions that are captured by the X operator in the label of the node is not a subset of any label in the tableau (*loop checking*), add one child to the leaf node labeled with S'
 - Otherwise, connect the leaf node with the node whose label is a superset of S' .

-> per approfondire vedi slide 61-67

1.5 Asynchronous concurrent reactive system (s75)

La logica temporale è usata per ragionare su sistemi asincroni concorrenti reattivi.

I sistemi reattivi sono sistemi che interagiscono con il loro ambiente e generalmente non dovrebbero terminare. Sono spesso chiamati agenti.

I sistemi concorrenti sono dei set di componenti che eseguono in modo concorrente.

Se sono asincroni solo un componente lavora ad ogni passo.

Se sono sincroni allora tutti i componenti lavorano assieme.

Il *modeling* di un sistema è usato per costruire specifiche (formali) di un sistema partendo da dettagli astratti irrilevanti.

La specificazione viene descritta in termini di:

Stato di un sistema, che è uno snapshot dei valori dei parametri che lo caratterizzano.

Transizioni di un sistema, che descrive come lo stato cambi nel tempo per via di eventi.

Computazione di un sistema, che è la sequenza di stati attivati dalle transizioni.

1.5.1 La struttura di Kripke

Formalmente una struttura di Kripke è una quintupla $K = \langle S, I, R, P, L \rangle$.

$I \subseteq S$ è un set non vuoto di stati.

$R \subseteq S \times S$ è una relazione di accessibilità, un set di transizioni non vuoto tale che R è totale a sinistra.

P è un set contabile di simboli proposizionali usato per costruire $\text{Prop}[P]$.

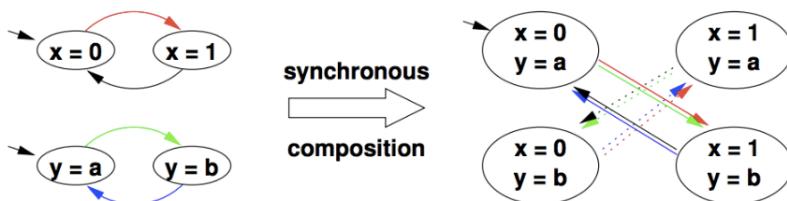
Un percorso PIGRECO in una struttura di Kripke K da uno stato S_0 appartenente a S è uno stato infinito di sequenze.

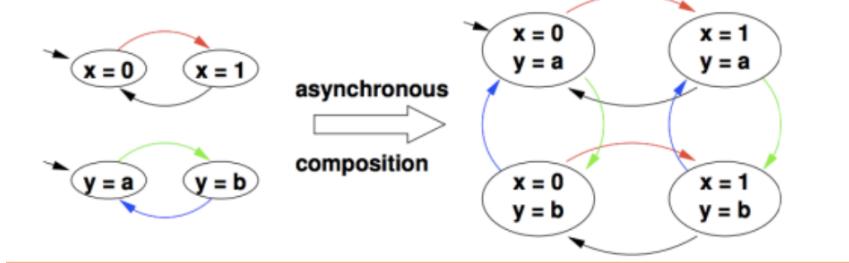
Strutture di Kripke complesse sono normalmente ottenute da una composizione di altre strutture più semplici.

Le strutture di Kripke possono essere combinate usando composizioni sincrone:

Lo stato delle componenti si evolve in sincrono.

Ad ogni momento nel tempo, ogni componente esegue una transizione.





Tipicamente le strutture di Kripke sono descritte usando linguaggi di alto livello chiamati *process (meta) languages*. Questi permettono di descrivere componenti (processi), aggregare componenti usando composizioni sincrone o asincrone e anche di descrivere componenti che contengono set di variabili, values iniziali o procedure.

1.5.2 Computation Tree Logic

CTL -> è una logica temporale che va oltre le limitazioni della logica LTL. È una branching time logic, mentre la LTL è lineare. La CTL introduce quantificatori di percorso, che non sono permessi nella LTL per via della linearità.

Inizialmente date due proposizioni CTL P e Q, si usano i seguenti operatori temporali esistenziali:

- E G P -> esiste un percorso che parte dal mondo corrente in cui P è sempre vero.
- E F P -> esiste un percorso che parte dal mondo corrente in cui P diventerà vero.
- E X P -> “ “ in cui P è vero nel prossimo mondo sul cammino.
- E P U Q -> “ “ in cui P è vero fino a che poi Q diventa vero.

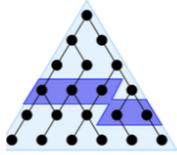
Abbiamo anche operatori universali temporali:

- **A G P** -> in tutti i cammini che iniziano dal mondo corrente, P è sempre vero.
- **A F P** -> "", P diventerà vero.
- **A X P** -> "", P è vero fino al prossimo mondo nel cammino.

A P U Q -> "", P è vero finché Q diventa vero.

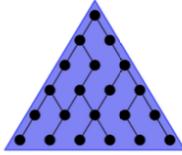


finally P



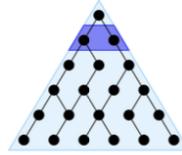
AF P

globally P



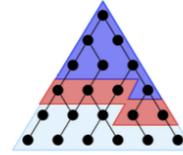
AG P

next P

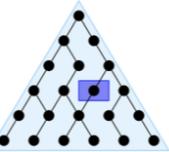


AX P

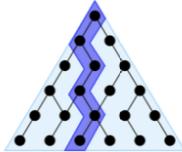
P until q



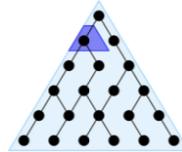
A[P U q]



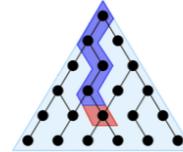
EF P



EG P



EX P



E[P U q]

1.5.3 CTL VS LTL

Diverse proposizioni CTL contengono quantificatori di percorso che non possono essere espressi in logica LTL.
Diverse proposizioni LTL che selezionano un range di cammini usando proprietà non possono essere espresse in logica CTL.

Diverse proposizioni LTL possono essere invece tradotte in CTL e vice versa.

CTL e LTL hanno poteri espressivi che non sono comparabili.
La scelta dipende dalla situazione.

-
- > per approfondire, slide 84-44
 - > per mutua esclusione slide 85-95

2. Sistemi software concorrenti in Java

2.1 Sistemi software concorrenti

Un sistema software è concorrente se la computazione è ottenuta dalla composizione di esecuzioni sequenziali indipendenti, che possono essere eseguite virtualmente in parallelo sotto specifici constraints che impongono la sequenzializzazione di parti delle computazioni.

Le computazioni di un sistema concorrente sono strutturate in termini di un set di execution flows (possibilmente dinamico): Ogni execution flow [→] performa una computazione sequenziale. Gli execution flows sono composti in parallelo sotto constraint di sequenzializzazione.

Gli execution flow possono condividere dati e risorse necessari per la computazione.

La descrizione statica di un sistema software concorrente è data da un set di programmi concorrenti:

Ogni programma concorrente descrive i flow di esecuzione del sistema concorrente.

Ogni programma concorrente descrive alcuni constraints sequenziali necessari per il flow.

Java è un linguaggio di programmazione concorrente:

Fornisce costrutti di linguaggio per esprimere per esprimere constraints di sequenzializzazione. Patterns complessi di sequenzializzazione possono essere espressi combinando diverse strutture.

2.2 Processi e threads

Un processo concorrente è un programma concorrente in esecuzione:

- Un programma è un documento statico che descrive il comportamento di un processo a livello di tempo di esecuzione.
- Un processo è una entità dinamica che esegue un programma usando un particolare set di dati e di risorse.
- Due o più processi possono eseguire lo stesso programma, ognuno usando un rispettivo set di dati e risorse.

Un processo comprende almeno le seguenti componenti.

- Il programma da eseguire
- I dati su cui ogni programma esegue
- Le risorse richieste dal processo a execution time
- Lo status di esecuzione del processo

I processi vengono eseguiti in un ambiente macchina astratta

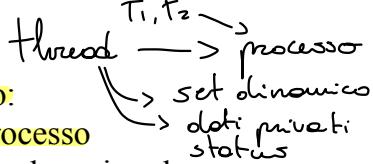
Processo :
 { - programma
 - dati
 - risorse
 - status }

che gestisce la condivisione dei dati e risorse e li isola nel modo corretto in mezzo agli altri programmi. La JVM fornisce la macchina astratta in java, avremo quindi una JVM per ogni processo.

JVM



Un **thread** è l'execution flow di un processo:



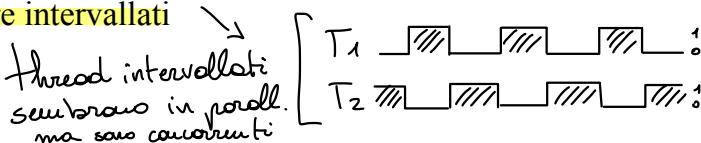
- Ogni **thread** è associato ad un singolo processo
 - Diversi **threads** possono essere associati ad un singolo processo
 - Un **set di threads** associati ad un processo è **dinamico** perché i threads vengono fatti partire e terminati dinamicamente
 - Un **thread** **alloca dinamicamente parte delle risorse** dei processi per i suoi requisiti di computazione
 - Oltre ai dati del processo, un **thread** ha i suoi dati privati e uno status



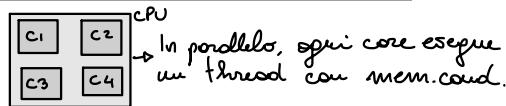
NB: Un processo sequenziale ha un singolo thread ed una memoria condivisa è visibile a tutti i threads di un processo.

I **threads** hanno le seguenti proprietà:

- Un thread inizializza la sua esecuzione in un punto specifico del programma. Per uno dei thread, chiamato il main thread, il punto iniziale è all'inizio del programma. Per ognuno degli altri altri thread, il punto di inizio è specificato da qualche parte nel programma ed è deciso a compile o run time.
 - Un thread esegue in modo ordinato una sequenza predefinita
 - Un thread esegue indipendentemente dagli altri thread
 - I thread appaiono eseguiti in parallelo anche se possono essere intervallati



2.3 Concorrenza e parallelismo

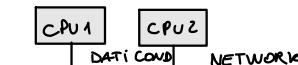


I **sistemi paralleli** sono lanciati su un set di CPUs per eseguire diversi processi e i loro thread in parallelo, ovvero allo stesso tempo.

Una memoria condivisa fra le CPU è normalmente presente.

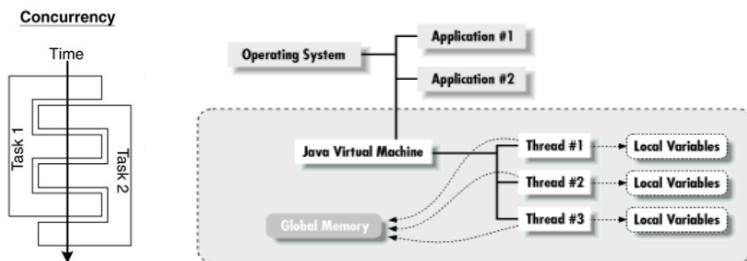
Se non ci dovesse essere una memoria condivisa e le

comunicazioni usano un network, allora il sistema è chiamato **distribuito**.



I sistemi concorrenti possono anche essere distribuiti su una singola CPU, ma sono strutturati come se fossero un sistema parallelo.

A volte, parti di un sistema concorrente sono parallele. Il designer ha il compito di dare per scontato che processi e threads vengano eseguiti in parallelo



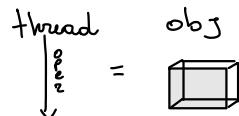
2.3.1 Threads in java

Ci sono due modi per descrivere un thread in Java:

- Estendendo la classe `java.lang.Thread` usando classi top-level o classi interne anonime
- Implementando l'interfaccia `java.lang.Runnable` usando classi top-level, classi interne anonime, lambda expressions o metodi statici referenziati

Ogni thread è associato con un oggetto chiamato *thread object*

```
public class MyThread extends Thread {  
    @Override  
    public void run() {  
        //body of the thread  
    }  
}
```



```
public class MyRunnable implements Runnable {  
    @Override  
    public void run() {  
        //body of the thread  
    }  
}
```

In entrambi i casi il metodo `run()` contiene la sezione del programma che il thread esegue.

L'inizio del metodo `run()` è l'**entry point** del thread.

Il metodo `start()` è usato per far partire il thread e iniziare l'esecuzione di `run()`

Viene chiamato su un oggetto thread, per esempio estendendo thread:

```
new MyThread().start();
```

Oppure implementando runnable: `new Thread(new MyRunnable()).start()`

Un thread termina quando il metodo run () termina. Un thread non può essere terminato forzatamente.

I thread in Java hanno un ciclo vitale strutturato su tre livelli:

Il ciclo di vita dei thread può essere alterato programmaticamente:

- Il metodo sleep (millis) forza il thread allo stato bloccante dopo il tempo specificato
- Il metodo interrupt () forza il thread allo stato running, e se il thread è nello stato bloccante quando interrupt viene invocato, il metodo che forza il thread verso lo stato bloccante lancia una *InterruptedException*
- Il metodo interrupted () controlla che un thread sia stato interrotto precedentemente
- Il metodo yield () trattiene un thread nel suo stato running, ma chiama il thread scheduler per permettere ad altri thread di eseguire
- Il metodo join () forza il thread allo stato bloccante, se necessario, e aspetta gli altri thread su cui è stato invocato per terminare

Su una singola CPU i thread eseguono uno alla volta di modo che sia possibile dare l'illusione di parallelismo - in ogni caso, differenti thread dello stesso processo possono essere eseguiti su diverse CPU per assicurare un vero parallelismo.

La JVM implementa un semplice algoritmo di scheduling per i thread chiamato *fixed priority scheduling* :

- La JVM schedula thread basandosi sulle loro priorità relative agli altri thread che sono nello stato *runnable*
- La JVM sceglie per l'esecuzione uno dei thread *runnable*

con la priorità maggiore

- Se due thread che stanno aspettando sono entrambi runnable e hanno la stessa priorità lo scheduler ne sceglie uno in ordine round-robin

Da notare, le priorità dei thread possono essere decise alla creazione e possono essere cambiate a piacimento del programmatore

2.3.2 Java Memory Model (s.112)

I thread di un programma Java hanno uno stack privato per supportare l'invocazione di metodi. Hanno anche un thread-local storage privato e condividono l'heap dell'oggetto. Il Java Memory Model (JMM) descrive come i thread accedono ai loro tre tipi di memoria oltre che come il contenuto delle memorie è salvato nella gerarchia di memoria.

I thread di un programma Java non sono necessariamente eseguiti tutti dalla stessa CPU perché la JVM di norma è distribuita su diverse CPU.

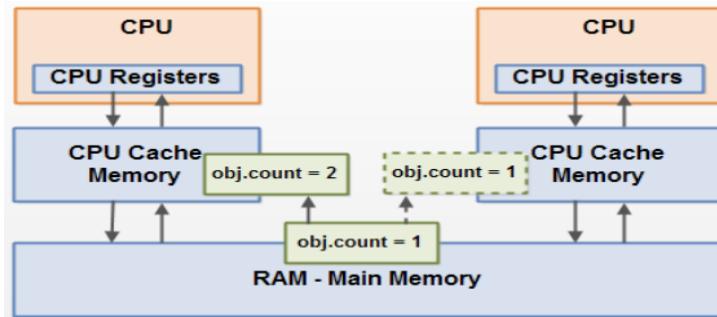


L'esecuzione di thread su multiple CPUs causa rilevanti problemi di *memory coherence*!

Questi problemi vengono risolti se l'accesso alla memoria condivisa è controllato.

Da notare che, tutti i controlli di accesso necessari possono

causare un uso inefficiente del parallelismo e delle risorse.



2.3.3 Mutua esclusione in Java

Il **problema della mutua esclusione** è uno dei più grandi nella programmazione concorrente. Un **programma concorrente** ha questo problema quando deve essere assicurato che, dato un **set M di sezioni del programma a mutua esclusione**:

- Solo un thread alla volta può eseguire una delle sezioni del programma in M
- I thread che non possono eseguire sezioni in M sono bloccati e riprendono il loro funzionamento il prima possibile

Per esempio, se un programma contiene il set $M = \{ P_1, \dots, P_N \}$ di procedure a mutua esclusione, solo un thread alla volta può eseguire una delle procedure in M.

La mutua esclusione in Java si risolve usando le sezioni

critiche

Una sezione critica è una sezione di programma associata a un oggetto MUTual EXclusion device (mutex), che controlla l'accesso alla sezione critica.

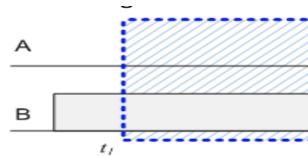
Un mutex può essere acquisito e rilasciato dai thread, ed è posseduto da un thread dopo la sua acquisizione fino al suo rilascio. Se un thread acquisisce un mutex, allora nessun altro thread può acquisire quel mutex a meno che non venga esplicitamente rilasciato.

I thread sono forzati nel loro stato blocked quando non possono acquisire un mutex, e sono forzati al loro running state quando invece possono.

Un thread esegue una sezione critica solamente quando è in possesso del mutex che fa da guardia alla sezione.

Ci sono due tipi di *acquisizione* e *rilascio* del mutex:

- Reentrant -> operazioni rientranti, perché un mutex può essere acquisito e rilasciato in loop annidati diverse volte senza che il thread che possiede il mutex vada in blocco
- Synchronous -> operazioni sincrone al processo perché i loro effetti sono sincronicamente condivisi tra tutti i thread del processo, anche fra diverse CPU.



Una sezione critica viene identificata da:

- I modificatori o metodi synchronized, che etichettano il

- corpo del metodo come sezione critica, e che `this` è
referenza all'oggetto che deve essere usato come mutex
- I blocchi `synchronized`, che etichettano il corpo del blocco
come sezione critica e che le referenze agli oggetti nella
head del blocco sono usati come mutex
-

```
public synchronized void myMethod() {  
    //sezione critica  
}  
  
public void myMethod( Object o ) {  
    synchronized(o) {  
        // sezione critica  
    }  
}
```

La disponibilità di una sezione critica in Java fornisce una ottima soluzione a diversi problemi:

1. *Thread indifference*, ovvero il problema che occorre quando due operazioni che girano su diversi thread agiscono sugli stessi dati e sono in parallelo. Questo causa una **race condition**.
2. *Memory coherence*, ovvero quando thread differenti hanno viste inconsistenti sulle copie degli stessi dati fra varie CPU (esempio per via di caches non allineate)

Da notare che l'uso della SC per salvaguardare contro memory coherence è una scelta di design di Java.

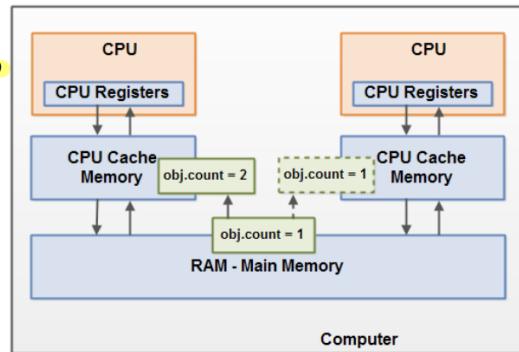
2.3.4 Happens before

Java definisce la relazione **happens-before** sulle operazioni

Happens
before

di memoria come read e write su variabili condivise.

Il risultato di una write da un thread è garantito visibile ad una read di un'altro thread solo se la operazione di write accade prima della operazione di read.



Ogni azione in un thread *happens-before* un'altra azione in quello stesso thread che viene più tardi nel programma.

Lo sblocco di un mutex avviene prima del seguente blocco e acquisizione del mutex stesso da un altro thread.

E' una relazione transitiva.

Una write su un campo *volatile* avviene prima di ogni sua successiva lettura.

Una chiamata *start ()* avviene prima di ogni altra azione nello stesso thread.

Tutte le azione in un thread accadono prima ogni altra tutte le altre azioni di un diverso thread che sta ritornando da un *join ()* su quel thread.

Propagazione del synchronized su campi oggetto
La parola chiave *synchronized* può risolvere diversi problemi di *memory coherence*. Inoltre *synchronized* su un oggetto OBJ si assicura che tutti i cambi allo stato di OBJ siano propagati a tutti i thread interessati prima di qualsiasi altro accesso sincronizzato a OBJ.

Le **operazioni atomiche** in Java sono operazioni *read* e *write*, che non possono essere interrotte per sospendere il thread corrente ed attivarne un'altro.

In dettaglio:

- Le read e write sono atomiche per referenze e variabili primitive
- Le read e write sono atomiche per tutti i campi e variabili dichiarati volatili

Le operazioni atomiche non soffrono di interferenze fra thread. Se però non si usa volatile ci possono essere problemi di consistenza con la memoria.

La usione per indicare operazioni atomiche

2.3.5 Waiting and Notifying

Un thread deve notificare gli eventi:

- 1 - Prima entra nella SC con un OBJ che fa da guardia
- 2 - Poi invoca il metodo notifyAll () sull'oggetto OBJ per notificare tutti gli altri oggetti che stanno aspettando che un evento è accaduto.
- 3 - Infine, lascia la sezione critica.

Un thread deve aspettare per gli eventi:

- 1 - Entra nella SC con un OBJ guardia, che è lo stesso delle notifiche.
- 2 - Poi invoca il metodo wait () sull'oggetto per bloccarlo e iniziare ad aspettare per eventi.
- 3 - Infine dopo essere tornato al running state per via di una notifica di un diverso OBJ, lascia la sezione critica.

Da notare che notifyAll () modifica tutti i thread che sono in attesa, e vengono tutti forzati allo stato running anche se solo uno di quelli poi andrà in SC.

Il metodo notify() notifica solo uno dei waiting thread, che è quello che poi entrerà in SC.

L'uso del metodo notify() è ereditariamente più efficiente ma

più prone ad errori.

Si può aggiungere wait(millis) e può lanciare eccezioni.



2.4 Liveness problems in Java

L'uso superficiale o erroneo del supporto per la programmazione concorrente può risultare in problemi seri di liveness.

Alcuni di questi sono:

- **Deadlock**: due o più thread sono bloccati per sempre; si aspettano l'un l'altro.
 - **Livelock**: due o più thread continuano a reagire a eventi di cui si notificano a vicenda.
 - **Starvation**: un thread non può accedere regolarmente alle risorse condivise e di conseguenza non può progredire come ci si aspetta.
-

2.5 High level abstractions for concurrency

Java fornisce meccanismi per gestire la concorrenza e i suoi problemi: astrazioni di alto livello adatte a migliorare manutenibilità, riusabilità e problem solving.

Alcune di queste astrazioni non sono altro che semplici strutture dati simili ad alcune viste in passato:

2.5.1 Blocking Queues

Una coda non è altro che una sequenza di oggetti che cambiano dinamicamente stando alla politica FIFO.

Operazioni basiche della coda sono: creazione, distruzione,

is_empty test, is_full test, enqueue e dequeue.

Una **coda bloccante** è una coda intesa per l'uso concorrente. Le operazioni sono bloccanti (vanno in blocco) se non possono essere eseguite immediatamente: enqueue può bloccare la coda se questa è piena e dequeue può fare l'opposto.

Da notare che tutte le code bloccanti possono bloccare su dequeue se sono vuote e solamente le code bloccanti a capacità limitata possono bloccare su enqueue se sono piene. Le code bloccanti sono un'astrazione che può essere usata per coordinare attività dentro ad un sistema concorrente.

2.5.2 Locks and conditions

Un **lock** esplicito è una astrazione che può essere usata per assicurarsi la mutua esclusione. Non ha limitazioni dovute a sezioni annidate come nelle sezioni critiche ma è molto più prona ad errori.

Un **lock** può essere:

- Esplicitamente acquisito e rilasciato
- Solo un thread alla volta può possedere un lock
- Un thread può bloccarsi nel tentativo di acquisire un lock che è già stato acquisito da un thread differente
- Un thread che è bloccato mentre tenta di acquisire un lock è restartato appena il lock può essere acquisito

Una **condizione** è una astrazione che serve per fare *waiting* ovvero aspettare un evento interessante e per segnalare eventi interessanti:

- Un thread può segnalare che una condizione è diventata true
- Un thread può bloccare waiting per una condizione finché viene segnalata
- Un lock è sempre necessario per fare da guardia a una condizione

Un lock è normalmente associato a diverse condizioni:

- Waiting e signaling su condizioni è possibile solo per i thread che possiedono il lock delle condizioni
- Quando un thread aspetta che una condizione venga segnalata, rilascia il lock della condizione
- Quando un thread segnala una condizione, deve rilasciare esplicitamente il lock della condizione

Code bloccante → Lock → condizione

2.5.3 Atomic References

Una referenza atomica incapsula una referenza verso un oggetto e ne gestisce la mutua esclusione. Le seguenti operazioni sono di norma fornite:

- Dereferenziare la atomic reference
- Assegnare la atomic reference
- Dereferenziare la atomic reference da R e assegnarla a f(R), ritornando R
- Dereferenziare la atomic reference da R e assegnarla a f(R), ritornando f(R)
- Le operazioni sopra sono tutte atomiche.

2.5.4 Pools of resources and Thread pools

I problemi di concorrenza sono spesso causati da risorse condivise. La corretta gestione degli accessi alle risorse condivise è molto importante.

Un gruppo di risorse identiche è chiamato **pool di risorse**, **thread pool**:

- Quando una pool viene creata o distrutta, tutte le risorse associate sono acquisite o rilasciate
- Le risorse sono assegnate per l'utilizzo dopo una richiesta dalla pool
- L'accesso controllato alle risorse è garantito dalla pool

Pool di risorse -> normalmente sono usati per controllare l'ammontare di risorse necessarie da un sistema concorrente. Per assicurarsi che ci sia un numero adeguato di risorse disponibili e che le risorse dette siano usate efficientemente.

I thread sono la risorsa più importante in un sistema concorrente. La creazione, distruzione e accesso ai thread è controllata usando semplici **thread pools**.

Quando viene creata, una thread pool crea e attiva tutti i thread dentro alla pool:

- per assicurarsi che i thread siano immediatamente pronti quando necessario
- per assicurarsi che il livello di concorrenza sia controllato

2.5.5 Executors and Callbacks

Un **esecutore** o **executor**, è un'astrazione che può essere usata per far girare più task in maniera concorrente. Viene associata ad una thread pool usata per far girare le task.

Fa enqueue delle task che non possono essere fatte partire immediatamente e fornisce metodi per ritornare risultati o eccezioni (se necessari) delle task che sono state fatte girare.

Permette anche di interrompere più task contemporaneamente in modo sexy.

A volte, fornisce un set di scheduling policies.

Un esecutore fornisce tre modi per eseguire le task:

- **One way execution** -> l'esecutore non fornisce modi per sapere se la task è stata effettivamente eseguita né per leggerne il risultato
- **Execution with callback** -> permette ad una task callback di usare il risultato ottenuto dalla task precedente
- **Execution with future** -> permette un futuro (promessa) che gestisce il risultato della task richiesta quando questo arriva

Gli esecutori sono asincroni.



Una **task callback** è una task che viene eseguita quando un esecutore completa una diversa task richiesta precedentemente.

La task callback riceve il risultato della task precedente, oppure l'eccezione generata (se ci sono). Dopo di che la task callback viene eseguita nello stesso thread che ha eseguito la task precedente.

Le task callback sono implementate come oggetti che sono associati ad una richiesta per l'esecuzione di una task.

2.5.7 Futures and futures pools

Un **future** o (promessa) serve per gestire 1. I risultati di task asincrone e 2. le eccezioni causate dalla terminazione di task asincrone. ↗ Può essere bloccante : dip. dal risultato

I future sono implementati come oggetti che sono associati ad una richiesta di esecuzione di una task: sono dinamicamente associati al risultato della task richiesta (se c'è) quando questo diventa disponibile. Allo stesso modo, sono associati

all'eccezione causata dalla terminazione della task, se c'è e quando diventa disponibile.

Un future blocca un thread che cerca di accedere ai suoi valori embedded se questi ultimi non sono ancora pronti. Il thread viene fatto ripartire dal momento in cui i valori diventano disponibili, oppure se al posto dei valori ci sono delle eccezioni.

Da notare che un future non è bloccante se i suoi valori sono già disponibili quando richiesti.

Visto che i futures bloccano il thread che cerca di leggere i valori interni, l'accesso ad un **set di future** da un singolo thread è sequenziale. I **pool di futures** possono essere trattati come un solo future con diversi valori:

- Se una future pool è costituita da n future, il suo valore può essere letto n volte per accedere al valore interno del future
- Il valore del future in una pool di future è accessibile non appena diventa pronto

2.6 Java Reflection

Java e la JVM forniscono un package chiamato `java.lang.reflect` per posporre delle decisioni a livello runtime. Java rimane un linguaggio *statistically typed* (le variabili sono dichiarate esplicitamente, ergo vengono definite a runtime), ma fornisce anche metodi puramente orientati agli oggetti come supporto, tipo:

- Linkin dinamico di classi

Java → vor definite a runtime

- Introspezione dinamica di oggetti
- Creazione dinamica di oggetti
- Accesso dinamico ai campi
- Invocazione dinamica dei metodi

NB

Da notare che i seguenti fatti sono sempre veri in Java:

- Ogni oggetto è associato alla classe che è stato usato per crearla, e viene chiamata *factory class*
- Ogni classe o interfaccia è rappresentata a runtime da un oggetto, e viene chiamato *class object (o descriptor)*
- Ogni oggetto classe è associato al suo *class loader*, che è l'oggetto usato per caricare il bytecode della classe

Gli oggetti classe sono gli entry point della reflection in java.

Seguono diverse maniere di ottenere oggetti classe:

- Dato un oggetto *o*, *o.getClass()* ritorna la classe associata all'oggetto
- Data una stringa *n* che contiene il nome completo della classe, *Class.forName(n)* ritorna la classe corretta dell'oggetto
- Dato un class loader *l* e una stringa *n* contenente il nome completo della classe, *l.loadClass(n)* ritorna la giusta classe dell'oggetto

Da notare che alla classe oggetto si può accedere dal nome anche se non ci sono oggetti della classe disponibili -> *ClassNotFoundException* può essere lanciata in caso

Oltre alle funzionalità relative alla riflessione dinamica in Java, un oggetto *c* di classe *Class<C>* può:

- Performare un tipo di cast di oggetto *o* alla classe rappresentata con *c.cast(o)* che è equivalente a *(C)o*.

- Controllare se un oggetto o è un'istanza della classe rappresentata, con $c.isInstance(o)$ che è equivalente a $o instanceof C$
 - Controllare se una classe o interfaccia referenziata sono uguali o se è una superclasse / superinterfaccia di una classe rappresentata da k , con $c.isAssignableFrom(K)$
- Alcuni altri usi di classe oggetto non sono discussi.

2.7 Introspezione

Dato un oggetto c di classe $Class<C>$, è possibile *inspectare* la struttura di C e per esempio:

- E' possibile fare una lista dei campi descrittori dei campi visibili di C
- E' possibile fare una lista dei costruttori descrittori dei costruttori visibili di C
- E' possibile fare una lista dei metodi descrittori dei metodi visibili di C
- E' possibile ottenere una referenza ad una classe oggetto della classe base (o superclasse) di C
- E' possibile ottenere referenze alla classe oggetto delle interfacce che C implementa

2.8 Dynamic in Java

2.8.1 Dynamic Object Creation

Dato un oggetto C di classe $Class<C>$ è possibile creare oggetti di classe C , per esempio:

- Usando $c.newInstance()$
- Usando c per accedere ad uno dei costruttori descrittori di c ed invocando il costruttore con argomenti corretti

Da notare che se *C* è conosciuto (non è ? o un tipo parametrazione), allora la creazione dell'oggetto dinamico non richiede type casts esplicativi.

Per esempio, la seguente è una stringa vuota: *String s = String.class.newInstance()*

2.8.2 Dynamic Access to Fields

Dato un oggetto *C* di classe *Class<C>*, è possibile accedere ai descrittori di campo dei campi visibili di *C*.

Oltre che descrivere i campi, i descrittori di campo possono essere usati come setter o getter per valori di campo per alcuni oggetti.

Dato un oggetto *C* di classe *Class<C>*, un descrittore di campo *f* ottenuto da *c*, ed un oggetto *o* di classe *C*, è possibile:

- Usare *f.get(o)* per leggere il valore corrente del campo per *o*
- Usare *f.set(o, v)* per settare il valore corrente del campo *v* per *o*

Da notare che la classe *java.lang.reflect.Field* non è generica e quindi non fornisce al compilatore il tipo di campo descritto.

2.8.3 Dynamic Method Invocation

Dato un oggetto *C* di classe *Class<C>* è possibile accedere al metodo descrittore del metodo visibile di *C*.

Oltre che descrivere i metodi, i metodi descrittori possono essere usati per invocare il metodo descritto con gli argomenti migliori.

Dato un oggetto *C* di classe *Class<C>*, un metodo descrittore *m* ottenuto da *c*, un oggetto *o* di classe *C* e un array di oggetti *a*, è possibile usare *m.invoke(o, a)* per invocare il metodo descrittore di *m* sull'oggetto *o* con argomenti *a*.

Da notare che la classe `java.lang.reflect.Method` non è generica e quindi non fornisce il tipo di ritorno del metodo descritto al compilatore.

2.8.4 Dynamic Proxies

Dato un array `a` di classe oggetti associato ad interfacce, un **proxy dinamico** è un oggetto che implementa le interfacce in `a` e invoca codice utente tramite invocazioni di metodo.

Nota bene:

- I proxy sono creati usando
`java.lang.reflect.proxy.Proxy.newInstance`
 - Il codice utente invocato è arbitrario ed è una implementazione di un'interfaccia funzionale
`java.lang.reflect.InvocationHandler`
 - Il codice utente è fornito col descrittore del metodo invocato
 - Il codice utente può ritornare un valore, che è usato come valore di ritorno della invocazione che ha triggerato l'attivazione del codice utente.
-

3.0 Aspect-Oriented programming

3.1 Aspect oriented programming I-III

L'AOP è stato considerato come il passo successivo dopo la programmazione orientata agli oggetti (OOP) dai primi anni del 2000.

In modo molto semplice, l'AOP riguarda l'aggiunta di determinati aspetti alla OOP.

Diversi approcci sono possibili ed alcuni sono estesi anche oltre la OOP per raggiungere altri paradigmi di

programmazione.

Nello scope della quasi pura programmazione orientata agli oggetti a cui Java si appoggia, la AOP può essere drasticamente semplificata:

- Aspetti e features di un oggetto che non sono prontamente fornite dalle loro classi
- Un **aspect provider** è un oggetto che può attaccare o staccare un aspetto ad un oggetto, oppure può creare un oggetto con un aspetto richiesto

In questa vista peculiare della AOP, l'interesse è volto a fornire implementazioni di aspetto che sono:

- Quasi completamente indipendenti dalle caratteristiche degli oggetti a cui sono attaccate
- Quasi completamente liberamente composable, di conseguenza un oggetto può essere attaccato a diversi aspetti, e di conseguenza può acquisire diverse features
- Quasi completamente ortogonali, quindi la composizione di aspetti fornisce la somma delle features indipendenti oggetto $\xrightarrow{\text{attach}}$ aspett (feature)

Questa particolare incarnazione della AOP può essere ottenuta in Java usando proxy dinamici. Dato un oggetto o un aspect provider attacca l'aspetto a o come un proxy dinamico che intercetta tutte le invocazioni dei metodi pubblici su o.

Seguono per esempio alcuni aspetti general-purpose che sono normalmente considerati:

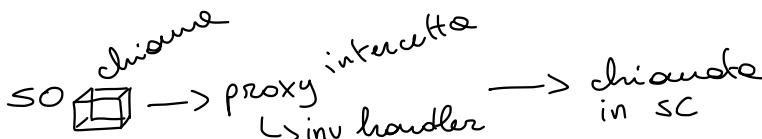
- **Shared:** un oggetto shared è un oggetto che deve assicurare mutua esclusione per l'esecuzione dei suoi metodi
- **Logging:** un oggetto logging è un oggetto che traccia le

invocazioni ai suoi metodi in un messaggio log

- **Persistent:** un oggetto persistente è un oggetto che sopravvive allo shutdown del sistema in cui è stato creato o modificato
- **Active:** un oggetto attivo è un oggetto che può eseguire i suoi metodi in un thread pool dedicato
- **Remote:** un oggetto remoto è un oggetto che accetta invocazione di metodi da client remoti e fornisce valori di ritorno ed eccezioni ai suoi clients

3.2 Aspects

3.2.1 Shared aspect



Un oggetto shared è un oggetto che deve assicurare la mutua esclusione per l'esecuzione dei suoi metodi. Solamente i metodi implementati dalle interfacce sono interessanti perché sono i metodi esportati.

Un proxy dinamico è sufficiente per intercettare tutte le invocazioni ai metodi che ci interessano:

- L'invocation handler fornisce un oggetto usato come lock per la sincronizzazione
- L'invocation handler entra in sezione critica a cui un lock fa da guardia prima di invocare il metodo bersaglio, ed esce dalla sezione critica subito dopo
- L'invocation handler esce dalla sezione critica anche in caso di eccezioni

3.2.2 Logging aspect

Un logging object è un oggetto che tiene traccia delle invocazioni ai suoi metodi tramite messaggi di log. Solamente i metodi che vengono implementati dalle interfacce ci interessano perché sono metodi esportati.

Anche in questo caso è sufficiente un proxy dinamico per intercettare le invocazioni:

- L'invocation handler crea logs prima e dopo l'invocazione del metodo
- L'invocation handler crea logs anche in caso di eccezione

3.2.3 Persistent aspect

Un oggetto persistente è un oggetto che sopravvive allo shutdown dei sistemi in cui è stato creato o modificato. Un proxy dinamico non è necessario in questo caso perché l'utente deve esplicitamente fare commit dei cambi allo store persistente e i rollback.



Semplici persistenze possono essere ottenute caricando / salvando oggetti serializzabili nei files:

- Oggetti che implementano java.io.Serializable possono essere serializzati in maniera semplice usando java.io.ObjectOutputStream e deserializzati usando java.io.ObjectInputStream
- Gli oggetti serializzabili forniscono un campo serialVersionUID privato per eliminare disambiguazioni tra versioni differenti della loro classe

3.2.4 Active aspect

Un oggetto attivo è un oggetto che deve eseguire i suoi metodi dentro a thread pool dedicate. Solamente i metodi implementati da interfacce ci interessano, perché sono esportati.

Di nuovo, basta un proxy dinamico per intercettare tutte le invocazioni ai metodi che ci interessano:

- Però, se il target object implementa l'interfaccia T, un'altra diversa interfaccia A (chiamata interfaccia attiva) è necessaria
- L'interfaccia attiva fornisce metodi firmati simili alle firme dei metodi in T, ma i risultati sono ritornati usando futures e callback

Normalmente, l'interfaccia attiva A di una interfaccia T deve estendere Active<T> e se R m (T1, T2, ..., Tn) è un metodo di T, allora

Future<R> m(T1, T2, ..., Tn)
void m(T1, T2, ..., Callback<R>) sono metodi di A

3.2.5 Remote aspect

Un oggetto remoto è un oggetto che accetta invocazioni da client remoti e fornisce valori di ritorno ed eccezioni ai client che hanno invocato i metodi.

Normalmente:

- Gli oggetti remoti aspettano per richieste remote da client su un canale di comunicazione
- Lo stesso canale di comunicazione che viene usato per ricevere richieste è anche usato per mandare le risposte
- Oggetti spediti su questi canali di comunicazione sono serializzabili
- Le richieste contengono l'identificatore del metodo che si vuole invocare e i suoi argomenti
- Le risposte contengono i valori di ritorno o le eccezioni

Il canale di comunicazione più semplice per oggetti remoti sono socket TCP/IP associati a singole interazioni.

Un server (oggetto) crea un socket per accettare tutte le richieste di connessione entranti usando

java.net.ServerSocket. La porta TCP è passata al costruttore ed il metodo *accept()* è usato per aspettare e ricevere un oggetto di classe *java.net.Socket* da cui leggere e scrivere al peer connesso. Normalmente un server processa connessioni in thread separati, uno per ogni connessione.

Un client (oggetto) crea un socket per richiedere una connessione ad un server usando *java.net.Socket* il cui host name e la porta del server vengono passate al costruttore. Il socket può essere usato per leggere e scrivere al peer connesso

Il remote aspect provider può essere usato per: registrare un oggetto su un server su una porta specifica o fornire un proxy per mandare richieste ad un server e ricevere risposte corrispondenti.

Se il server implementa l'interfaccia T, allora il proxy dinamico usato per accedere in remoto al server implementa l'interfaccia T.

Nota che, un proxy fornisce metodi chiamata sincroni al server. Il client aspetta finché la risposta ad una richiesta è disponibile. Metodi chiamata asincroni possono essere facilmente ottenuti attaccando gli aspect attivi al proxy remoto.

Usiamo il protocollo **TCP** per le connessioni. Arriva tutto, corretto, nell'ordine in cui è stato spedito. Non ci poniamo diversi problemi che il protocollo TCP ci permette di bypassare senza problemi. Forniamo quindi all'oggetto server la comunicazione, processiamo la risposta, la diamo, e

chiudiamo il canale.

3.3 Annotations in Java

Tra le altre, una funzione che si distingue in Java sono le **annotazioni**: possono essere introdotte nel codice sorgente usando la “@”. Per esempio `@override`. Possono anche avere argomenti a compile time tipo `@SuppressWarnings ("unchecked")`.

Le annotazioni sono usate di norma in un sorgente per suggerire al compilatore delle proprietà relative ad alcune parti del codice. Sono metadati sintattici. Sono spesso usate per controllare warning e errori, ma anche per controllare il comportamento del compilatore. Inoltre possono essere ritenute a runtime e ci si può accedere via reflection.

Le annotazioni possono essere attaccate a definizioni di:

- classi, interfacce, enums
- costruttori, metodi, parametri
- campi e var locali
- packages
- annotazioni etc

Il compilatore Java fornisce un po' di general purpose annotations spesso usate.

NewAnnotation

```
public @interface NewAnnotation {  
    String value() default ""; String name();  
    int age();  
    String[] newNames();
```

} Questa è una annotazione dichiarata da un utente, contenuta in un normale source file. Una volta importata nel source code, una annotation definita dall'utente può esser usata per annotare parti del sorgente.

```
@NewAnnotation(  
    name="Mario",  
    age=37,  
    newNames={"Giovanni", "Matteo"}  
)  
public class AClass { }
```

Se una annotazione A dichiara solo un campo chiamato *value* allora la notazione @A(v) può essere usata al posto di @A(value=v).

La definizione di un'annotazione può usare le seguenti annotations per descrivere alcune proprietà delle annotazioni definite dall'utente.

- **@Target (t)** si usa per enumerare parti del sorgente che possono essere annotate con user defined annotations.
- **@Inherited** può essere usato come segnale che una Java annotation usata in una classe deve essere ereditata dalla sua sottoclasse.
- **@Retention (p)** può essere usato per dichiarare la *retention policy* della annotazione, che è la politica che il compilatore dovrebbe usare per decidere quando smettere di propagare le annotazioni dal compile al run time.

3.3.1 Runtime annotations

Le annotazioni user defined sono usate per fornire metadati a

runtime via reflection.

L'uso di annotazioni a runtime è una estensione della introspezione.

Una annotazione può essere resa pronta a runtime definendo la sua retention policy a runtime.

3.3.2 Annotation processors

Le annotazioni definite dall'utente sono spesso usate per istruire il compilatore a eseguire codice fornito dall'utente quando processa del sorgente che contiene annotations.

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

Un annotation processor definito dall'utente può essere usato includendo un JAR file che classifica il processor nel buildpath.

Tutti gli annotation processor disponibili nel buildpath sono inclusi nel processo di compilazione e ripetutamente usati per processare il codice.

Un annotation processor può:

- aggiungere all'output del processo di compilazioni un set di errori, warning o log
- aggiungere all'input di un processo di compilazione un set nuovo di file source
- abortire dei processi di compilazione segnalando errore

Comunemente sono usati per controllare convenzioni che non sono forzate dal compilatore e generare sorgente che può essere facilmente scritto partendo dal contenuto di altri sorgenti disponibili dalla compilazione.

3.4 The Java Language Model I-III

Gli annotation processor ricevono in input contenuti del

sorgente rappresentato in termini di oggetti Java, non file di testo. Il compilatore fa parsing del sorgente per produrre un AST abstract syntax tree per ogni codice sorgente.

Un **AST** è composto da oggetti connessi che descrivono tutte le parti rilevanti di un sorgente. È disponibile solo per sorgente corretto sintatticamente. È passato all'annotation processor per ispezione e supporto alla costruzione di nuovi sorgenti. I nodi di un AST sono chiamati *elementi* del sorgente e sono oggetti che implementa.

Il grosso del lavoro di un annotation processor è basato sul processare elementi che vengono dati al AST.

Non sempre però i data types sono rappresentati come elementi, perché non sono sempre definiti usando elementi del sorgente.

Il Java language model fornisce **type mirrors** per poter usare data types a compile time.

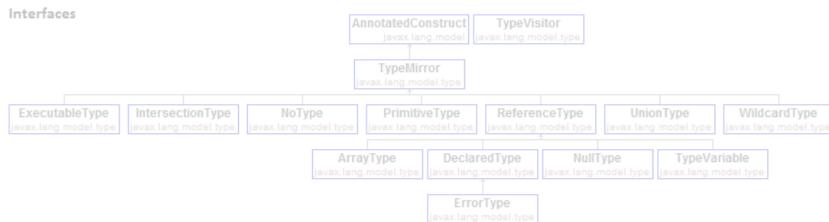
Il **TIPO** di un type mirror è una label che può essere usata per controllare che il type mirror descriva un tipo di dato primitivo o dichiarato. Performa anche operazioni comuni sui dati.

Da notare che il JLM non è ristretto alle dichiarazioni di tipi ma descrive anche le parti procedurali di un sorgente Java.

Interfaces

Packag
java.lang

An
java.x



3.5 Test Driven Development

Il **costo correlato al software** è spesso la parte predominante del costo di un sistema.

La parte più significativa del costo correlata al software è associata con la **manutenzione più che lo sviluppo**. Uno dei goal principali del software engineering è **ridurre quanto più possibile i costi di manutenzione** delle strutture e aumentare al massimo il riuso.

3.5.1 Costi

Normalmente i costi dei sistemi software sono divisi in:

- costi diretti che sono direttamente associati con le attive fatte per il sistema, quindi costo di sviluppo e di tool, librerie esterne ecc
- costi indiretti che sono associati alle attività necessarie per supportare le attività del sistema, quindi costo di consultazione amministrativa, legale ecc

3.5.2 Evolution and maintenance

I sistemi software devono evolversi perché: o i requisiti iniziali non sono stati calcolati correttamente o perché i requisiti sono cambiati durante il ciclo di vita del sistema. L'evoluzione di un sistema software è inevitabile anche se all'inizio tutto è stato fatto bene. La manutenzione è un'attività che viene preformata per far evolvere un sistema software concorrentemente al suo utilizzo -> rimuovere anomalie, migliorare la qualità ecc.

La manutenzione costa spesso più della metà dei costi relativi ai sistemi software nel loro ciclo di vita. Spesso si avvicinano al 75%. La manutenzione può costare dal 20% al 60%. Studi ben fatti sui costi di manutenzione in sistemi software già impiegati forniscono una prova empirica che la maggior parte delle anomalie possono essere trovate revisionando gli *artefatti* di progetto.

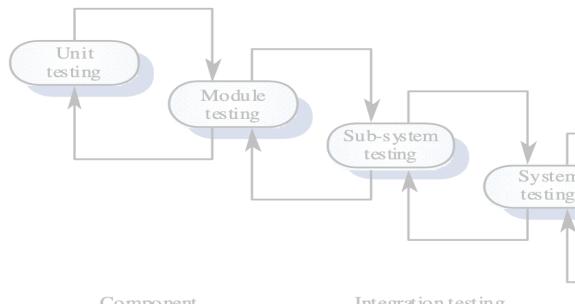
3.5.3 Testing

- Se il comportamento di una parte del sistema viene testato in una quantità di casi sufficientemente grande, allora il

comportamento di quella parte può essere considerato accettabile anche nei casi rimanenti.

- Il testing può trovare anomalie ma non può provare che parte di un sistema sia corretto.

Di norma il testing è diviso in due parti: Testing in the small (dove le parti singole sono testate) e Testing in the large (dove l'intero sistema viene testato).



Abbiamo diversi tipi di testing:

- Unit testing -> testing delle singole unità di riuso
- Module testing -> testing del sub-system che sono i sistemi piccoli richiesti dal sistema
- System testing -> testing della integrazione dei sub-system
- Acceptance testing -> testing performato insieme al cliente per assicurare che il sistema consegnato esibisca tutte le qualità tecniche richieste

Il testing in the large tratta il sistema come un black box ed il testing serve a controllare che il comportamento del sistema sia come ci si aspetta.

Il testing in the small tratta il sistema come una white box e ne esamina parti sufficientemente piccole ispezionando il codice consegnato.

Lo statement testing è anche detto coverage testing perché

basato sul fatto che nessuna parte del codice può essere considerata testata se non è stata eseguita.

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

Il **branch testing** è spesso chiamato *path coverage testing* perché ha come bersaglio l'analisi di tutte le esecuzioni dei path del codice in considerazione. Un set di test cases T può essere usato per performare branch testing di un codice C se, dopo aver performato tutti i test cases in T tutti gli execution paths in C sono stati eseguiti almeno una volta.

Il **branch and condition testing** viene spesso chiamato *condition coverage testing* perché ha come bersaglio l'analisi delle cause che generano path di esecuzione differenti nel codice considerato. Un set di test cases T può essere usato per performare il branch and condition testing di un codice C se, dopo aver performato tutti i test cases in T, tutte le condizioni in C hanno preso in considerazione tutte le cause per i loro valori.

3.5.5 JUnit

Il JUnit è un tool di supporto strutturato e accurato per testing del codice Java. JUnit funziona su Test cases e Test suites.

Eclipse ci fornisce un supporto diretto per JUnit per promuovere il test-driven development.

In generale con la JUnit andremo sempre e comunque a testare parti del programma che sono sincrone. Tutto ciò che è concorrente lo andremo a testare con pathfinder o sul foglio di carta.

3.5.6 Design patterns

Un design pattern è una soluzione generale e riusabile ad un problema comune dato un contesto nel software design. Non è un design finito che può essere trasformato direttamente in codice sorgente, è una descrizione o un template per come risolvere un problema che può essere usato in diverse situazioni.

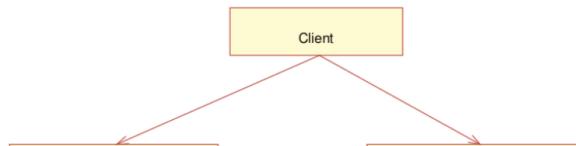
I design pattern sono formalizzati come **best practices** che il designer può usare per risolvere problemi comuni quando realizza un sistema.

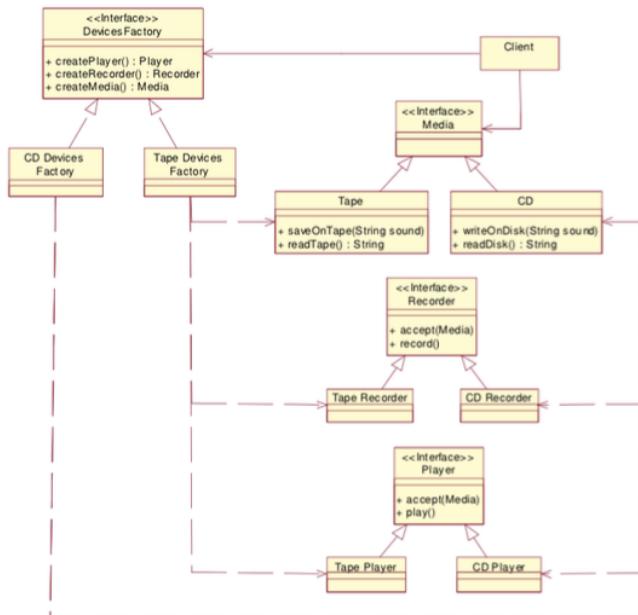
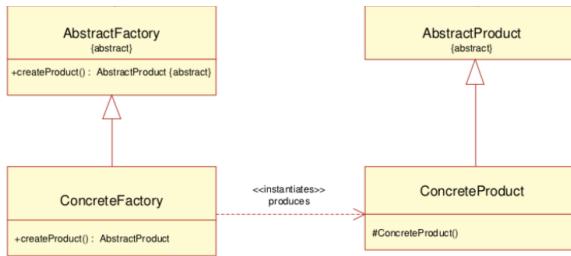
Le design pattern orientate agli oggetti, tipicamente mostrano relazioni e interazioni fra classi e oggetti ma senza specificarne l'applicazione finale di classi o oggetti che ne fanno parte.

Ci sono anche le **creational patterns**:

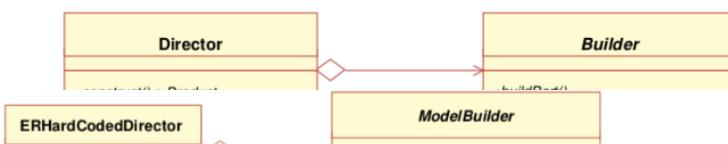
- **Abstract factory** -> gruppa oggetti factories che hanno un tema comune
- **Builder** -> costruisce oggetti complessi separando costruzione e rappresentazione
- **Factory method** -> crea oggetti senza specificare la classe esatta da creare.
- **Prototype** -> crea oggetti clonando oggetti già esistenti
- **Singleton** -> restringe la creazione di oggetti per una classe ad una sola istanza

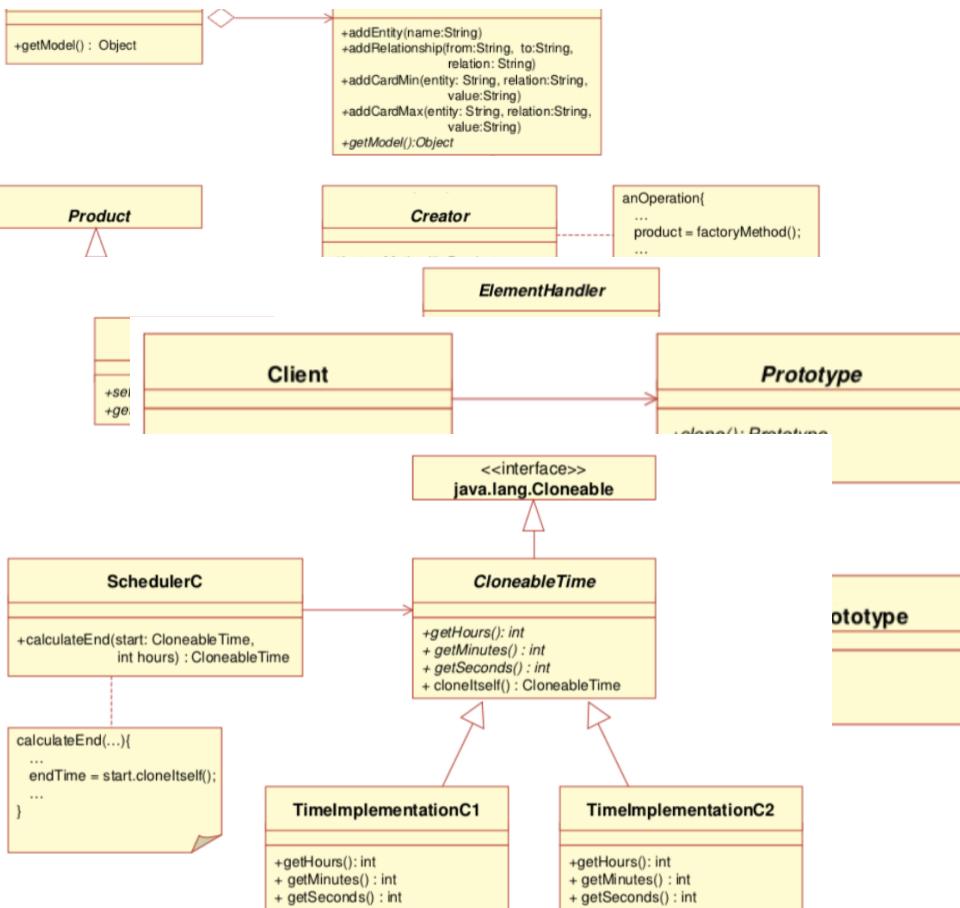
Abstract F:





Bui

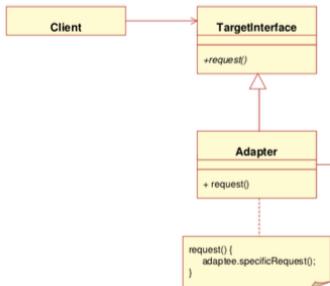




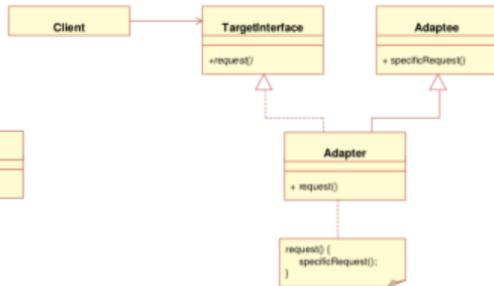
3.6 Structural Patterns

Adapter -> permette alle classi con interfacce incompatibili di lavorare insieme facendo wrap della propria interfaccia attorno a quella di una classe già esistente.

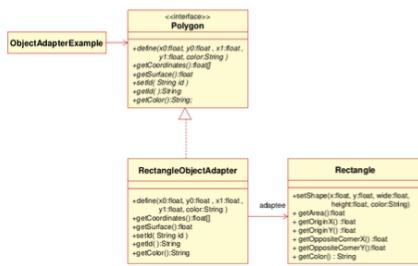
Object adapter



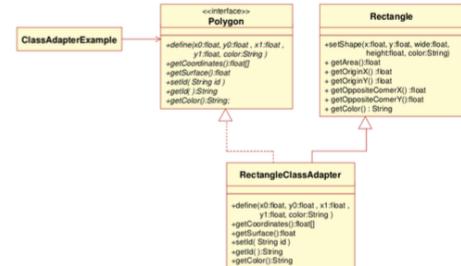
Class adapter



Object adapter

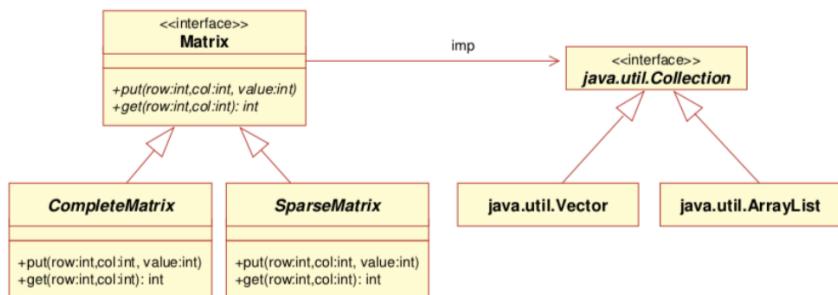
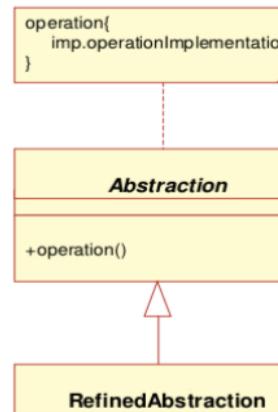


Class adapter

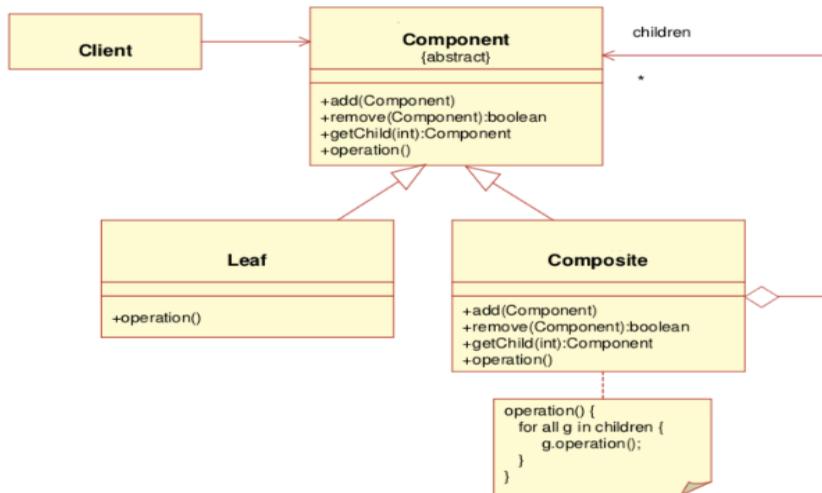


Bridge -> separa una astrazione dalla sua implementazione

così che le due possano variare indipendentemente.

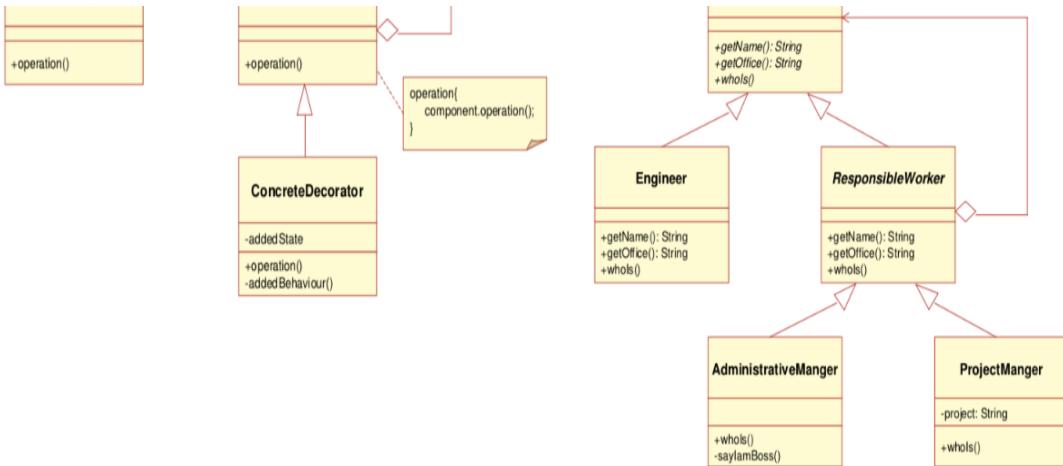


Composite -> compone zero o più oggetti simili così che possano essere manipolati come se fossero un solo oggetto.

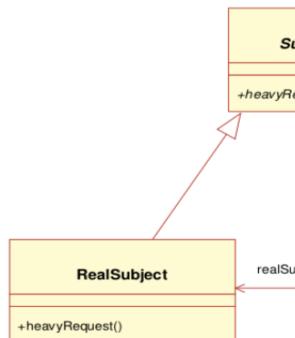


Decorator -> aggiunge o fa override dinamicamente del comportamento in un metodo esistente di un oggetto.

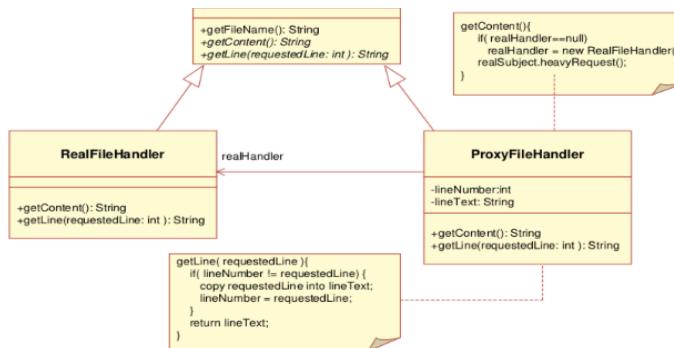




Proxy -> fornisce un placeholder per un altro oggetto per controllare l'accesso e ne riduce la complessità.

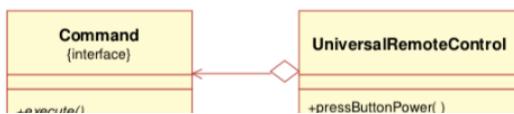
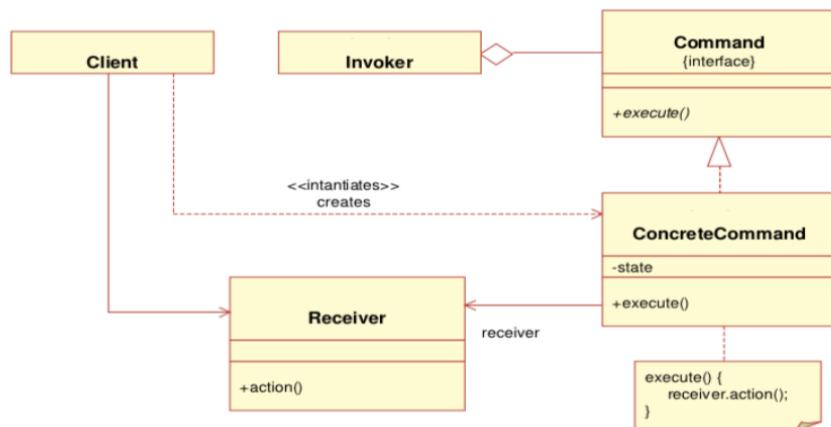


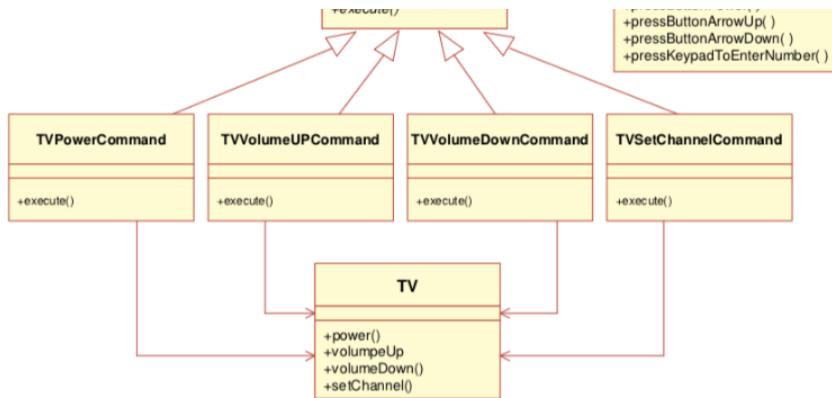
FileHandler



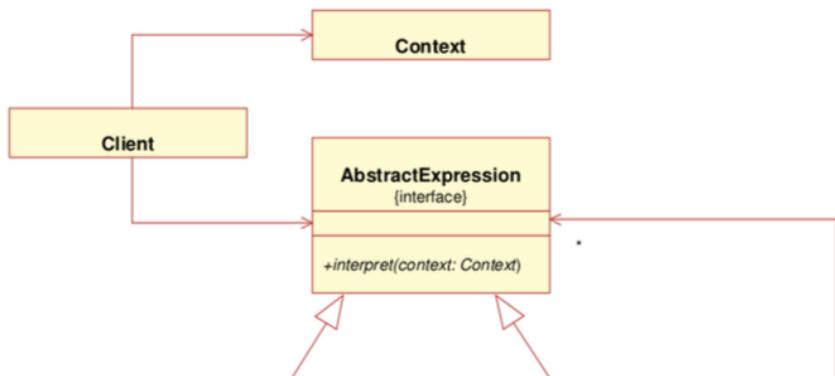
3.7 Behavioral patterns

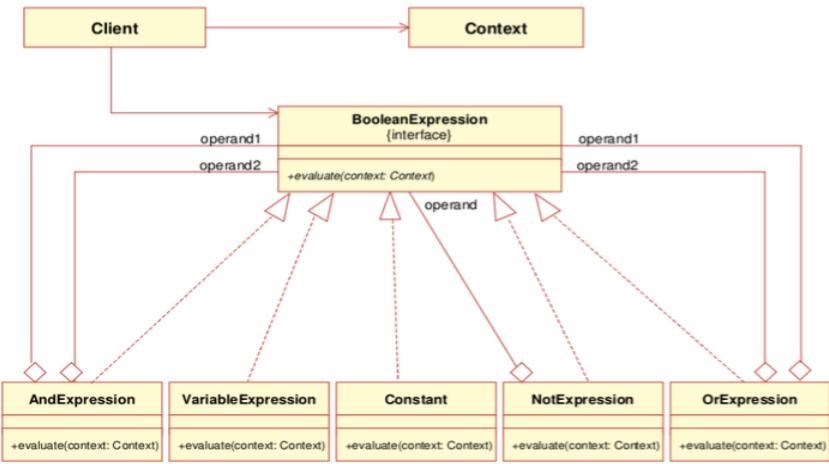
Command -> crea oggetti che encapsulano azioni e parametri.





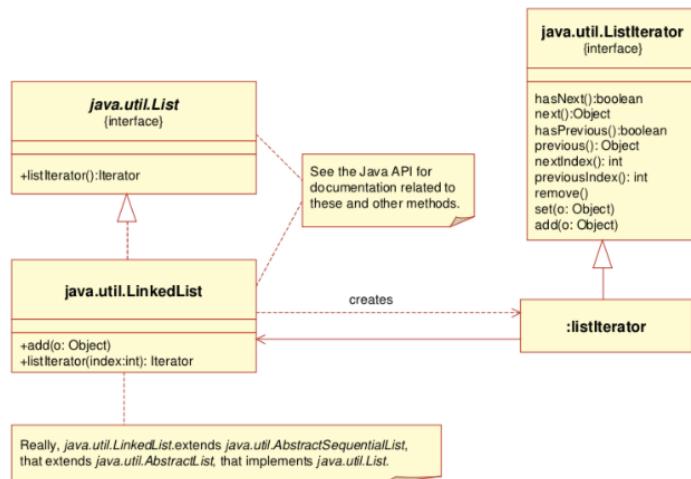
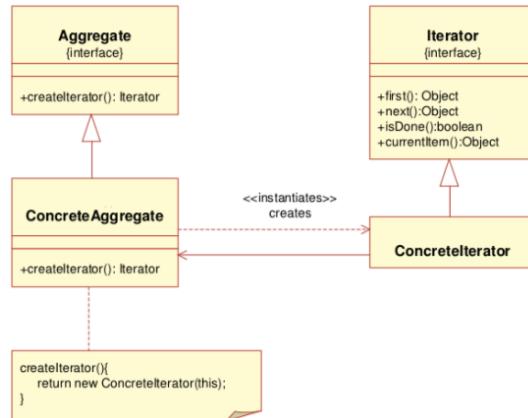
Interpreter -> implementa un interprete per un linguaggio specializzato, spesso un DSL (domain specific language).





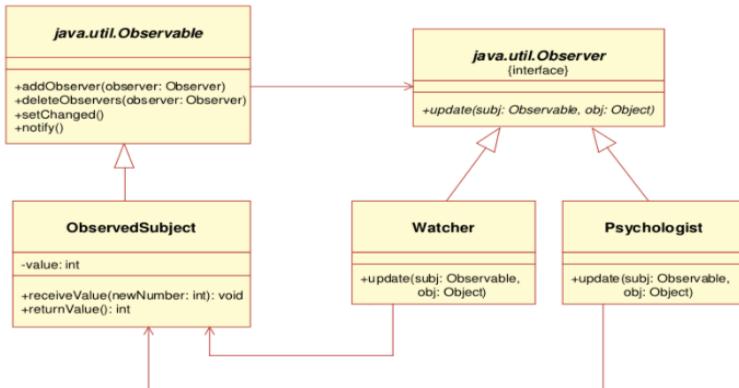
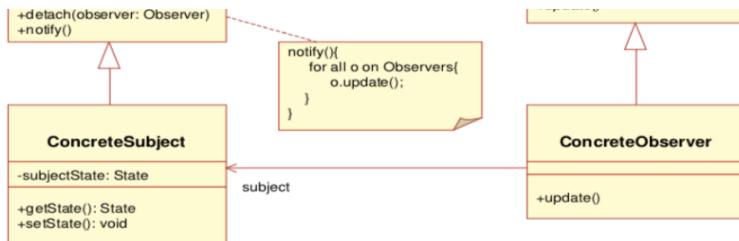
Iterator -> accede agli elementi di un oggetto

sequenzialmente senza esporre la loro rappresentazione sottostante.



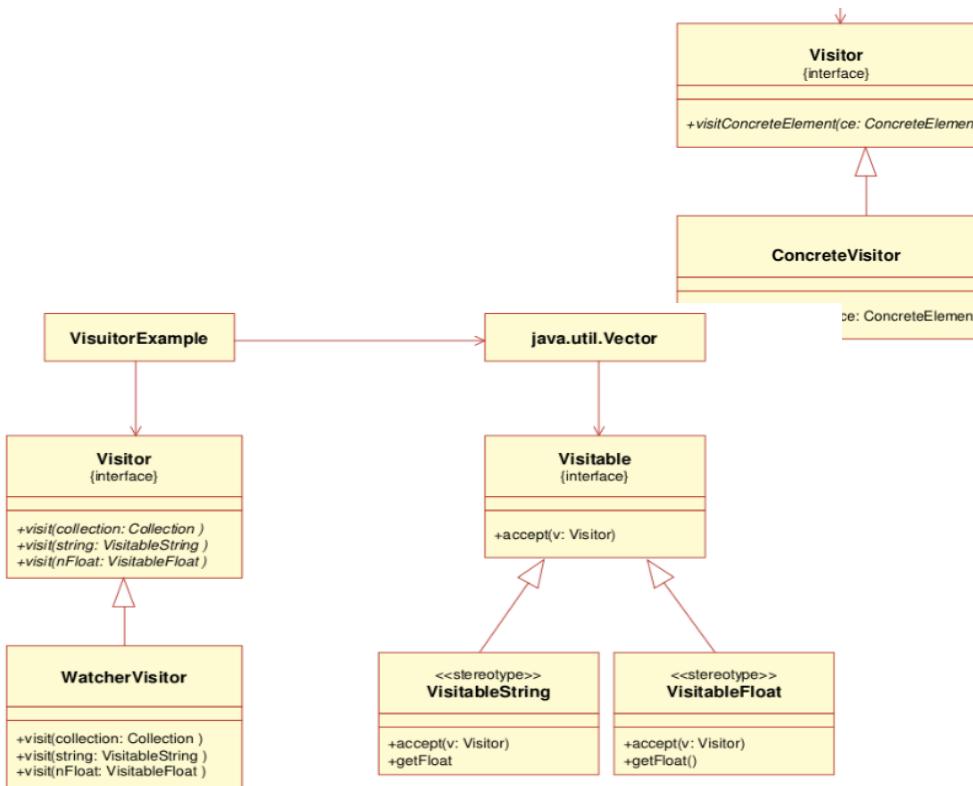
Observer -> è un pattern publish/subscribe, che permette ad un numero di osservatori oggetto di vedere un evento.





Visitor -> separa un algoritmo da un oggetto structure, muovendone la gerarchia di metodi in un oggetto.





L'idea del JLM è sfruttare la possibilità di inserire del codice nostro all'interno del compilatore java per generare del codice che ci serve per semplificare alcune operazioni che sarebbero considerate di routine.

Per utilizzare l'aspetto attivo dobbiamo costruire una interfaccia attiva, quindi non possiamo usare l'ereditarietà perché deriva da una interfaccia simile ma che non è uguale ed i metodi sono diversi.

Il JLM è una descrizione astratta di quello che può contenere un codice sorgente java che viene messa a disposizione all'annotation processor per analizzare i sorgenti che il compilatore sta compilando senza necessariamente dover fare dei file di testo.

4.0 Eclipse examples

4.1 Concurrency - classes and interfaces

4.1.1 Blocking Queues

La blocking queue è una coda che utilizziamo per i nostri scopi. E' un'interfaccia che andiamo a definire. La coda bloccante è una interfaccia nel package concurrent generica con parametro generico T. Ha 4 metodi: enqueue, dequeue,

isEmpty e isFull.

nota: i costruttori non vanno nell'interfaccia

Le code possono essere distrutte, ma in Java vuol dire solo renderlo invisibile.

Ad ogni modo anche il distruttore non andrebbe nell'interfaccia.

I metodi Enqueue e Dequeue sono due metodi bloccanti, quindi devono poter lanciare la InterruptedException. L'unica differenza da una coda normale è che la nostra è bloccante. Contenendo i synchronized, si acquisiscono e rilasciano i mutex quindi c'è concorrenza e blocca certe operazioni.

Possiamo fare una prima implementazione su una lista concatenata (non ha limiti di lunghezza). Implementiamo quindi questa coda bloccante per prima cosa su una lista concatenata, che la rende una coda bloccante concatenata, o LinkedBlockingQueue.

4.1.2 Linked Blocking Queue

E' una classe final ovvero non può essere estesa. Serve per dire all'utente di non fare le sottoclassi. Di conseguenza tutti gli attributi sono privati (non protetti, perché si possono vedere all'interno del package).

La prima riga è ovvia, la nuova classe implementa l'altra. Vogliamo che dentro ci sia una public LinkedBlockingQueue.

Poi un costruttore vuoto (riga 13). *This*. fa riferimento al nostro oggetto, lo trattiamo come attributo.

Comincia ora l'override dei metodi con *@*. Serve per dire che il metodo definito sotto è un override di un metodo della classe base o implementazione dell'interfaccia.

Lo facciamo di isFull() ma sincronizzarlo è inutile perché ritornerà sempre false.

NB: Hereditary anomaly -> anomalie sul riuso dato dall'utilizzo di blocchi synchronized che non servono a nulla.

In isEmpty() invece la sync serve perché le chiediamo la dimensione, e quindi accediamo alla coda. Usiamo lo stesso mutex per enqueue e dequeue. Prendiamo la sezione critica con synchronized (queue) e aggiungiamo poi facciamo la notifica ai consumatori. Per dequeue facciamo wait se la coda è vuota. I produttori in questo caso non devono bloccarsi ma i consumatori sì.

Il notifyAll() sveglia tutti i consumatori in attesa sul mutex. notify() invece ne sveglia uno.

Il consumatore svegliato (uno) controlla se la coda sia vuota o meno. Se non lo è ne sveglia uno. Altrimenti non sveglia nessuno. Questo evita di svegliare 50 e passa consumatori senza nulla da consumare. Le code bloccanti sono comode per rallentare la produzione

4.1.3 Array Blocking Queue (JPF)

Al posto di una coda concatenata usa un array di oggetti. Abbiamo una coda di oggetti che rappresenta la sequenza. In sarà dove mettiamo il prossimo. Out è da dove togliamo. Count è il numero di elementi attualmente nella coda. Size è la dimensione dell'array.

Nel momento in cui sono in fondo all'array mi servirà.

Se size == count la coda è piena e allora mi metto in wait. Può succedere solamente ad un produttore però. C'è un attraversamento circolare dell'array se -> count < size, perché si sblocca. Mette in l'oggetto, incrementa count e sposta in di

una posizione.

Questo succede quando in == size e si torna a 0.

Produttori e consumatori possono fermarsi, le due notify sbloccano di modo che possano lavorare dopo una dequeue o una enqueue. Mettiamo null in posizione out perché altrimenti il garbage collector avrebbe dei problemi per via dell'array di riferimenti.

Sappiamo così che l'unico riferimento a result è result e non ci sono riferimenti all'interno della coda.

4.1.4 Lock e ReentrantLock

Non lo usiamo perché siamo stufi di usare synchronized, ma lo usiamo perché vogliamo utilizzare le condizioni. Java non ci permette di usare condizioni con synchronized, quindi è per quello che li usiamo.

Una volta che un thread viene lockato dobbiamo tenerne traccia, per sapere anche quanti unlock dobbiamo fare.

Abbiamo anche bisogno di un oggetto mutex, per fermare il thread lo mettiamo in wait() su un mutex. Possiamo usare *this* come mutex ma non usiamo quello perché vogliamo essere gli unici all'interno della classe a decidere quando blocchiamo e sblocchiamo.

La classe ha 3 attributi: un riferimento al proprietario del lock. Un contatore = 0 che dice quante volte viene acquisito, incrementando e decrementando. Reentrant lock ovviamente implementa lock. Se vogliamo prendere il lock e siamo già owner, aumentiamo il counter. Se non siamo owner perché ce l'ha qualcun altro ci mettiamo in attesa.

Se owner è nullo, divento io owner. Siamo all'interno del blocco synchronized. La wait() può interrompersi -> se lo fa

la lock va avanti.

Anche la unlock lavora su un blocco synchronized. Sono tutti metodi gestiti con una guardia che tiene l'intero metodo.

4.1.5 Condition

L'interfaccia condition dirà cosa possiamo fare sulle condizioni: abbiamo await() e signal().

Await è mettersi in attesa di qualcuno che faccia signal. Signal sveglia tutti quelli che si sono messi in await() su quella condizione.

Possiamo mettere condition come inner class di reentrant lock. Facendo la inner class automaticamente abbiamo riferimento al lock.

NB: finally nel try catch: è come un try catch ma **finally** viene lanciato in 3 situazioni-> viene catturata una eccezione, NON viene catturata nessuna eccezione, si continua.

Il lock anche nella Array Blocking Queue garantisce la mutua esclusione, per questo motivo lo usiamo.

Se anziché usare un lock comune usiamo un oggetto esterno su cui facciamo sync, abbiamo la dequeue che si mette in attesa sull'oggetto condizione. Ma se facciamo wait() su quell'oggetto liberiamo il mutex. Otteniamo quindi un deadlock.

Nella wait facciamo unlock dell'oggetto lock e POI ci mettiamo in attesa. Una volta sbloccati lo riprendiamo quando serve. Se lo facessimo con un oggetto esterno non riusciremmo a farlo.

4.1.6 Atomic Reference

La realizziamo con una classe generica e ha tipo T.
Memorizziamo in value quello che sarà il nostro riferimento.
Per ottenere il riferimento dell'oggetto a cui puntiamo usiamo la get. Se vogliamo renderlo modificabile aggiungiamo anche la set, che ha tipo T, blocca il lock con sync e sovrascrive value. Perdiamo ovviamente il riferimento all'oggetto di prima e ne abbiamo uno a quello nuovo.

Sarebbe una ***shared reference*** se ci fossero solo queste due.
Può essere letta e scritta in mutua esclusione. Volendo si potrebbe usare volatile e togliere il lock.

Volendo garantire le letture e scritture in senso atomico dobbiamo usare il lock per costruire il blocco synchronized.

Passiamo noi come argomento il valore passato per la sovrascrittura. Queste operazioni di lettura e modifica in modo atomico stanno alla base dei meccanismi di mutua esclusione di basso livello. E' garantito che il valore restituito sia quello che abbiamo sovrascritto.

NB-> ***interfacce funzionali***: hanno un unico metodo. Si può implementare dicendo al compilatore di stabilire il nome del metodo solo guardando l'interfaccia. La sua implementazione richiede solo che quel metodo abbia un corpo.

4.1.7 Thread Pool

Usiamo un runnable. Creiamo prima l'astrazione -> *interface ThreadPool*. I runnable si mettono in coda aspettando che i thread arrivino, una volta che arrivano il runnable prende il thread e scoda, lo esegue e fa catch di eccezioni. Dopo di che si rimette in coda. I thread aspettano che ci siano dei runnable

per eseguirli. Siccome i thread stanno in un ciclo infinito usiamo un metodo stop () per fermarli.

Nel costruttore quando costruiamo i thread stiamo costruendo delle implementazioni dell'interfaccia thread. Quindi sono inner class. Poi costruiamo la coda dei runnable con una linked blocking queue (così non abbiamo limiti superiori) e accodiamo i runnable che verranno scodati dai thread nel pool e poi una var bool stop che dice se è fermo o no.

Al posto della coda FIFO potremmo usare una coda con priorità: ma non è questo il caso visto che è un simple thread pool.

La classe interna è una classe interna privata (possiamo stare più tranquilli sui vincoli di verifica dello stato). La classe inner thread è interna non statica (ha bisogno di accedere allo stato dell'oggetto contenitore tipo il flag).

4.1.8 Download Manager

Stampa quanti dati sono stati scaricati. Contiene un pool di thread ed il numero di connessioni che abilitiamo per il DM lo passiamo nel costruttore. Il numero di connessioni viene usato per costruire il thread pool. Rifiutiamo l'esecuzione dei runnable se non abbiamo chiamato start. Ci sarà una guardia che se ne occupa.

C'è il try catch con risorse, ma anche quello normale può farlo.

OpenStream lancia la richiesta al server, si collega col socket e manda la get. A quel punto input stream ci collega con i dati -> ci può sempre essere una eccezione a cui serve ovviamente un catch.

4.1.9 Executor + Simple Executor

Astrazione -> definiamo l'interfaccia. Col metodo start facciamo partire l'esecuzione dei thread collegati. Diventa un esecutore concorrente (lo fa su più thread) e si mette in attesa di eseguire. C'è anche il metodo stop per fermare l'esecuzione. C'è un metodo execute runnable, lo esegue ma non offre nessun modo per informare il chiamante del completamento del task né del risultato. E' quindi un *one way executor*.

Più avanti c'è un metodo execute con callback e un execute con future.

SimpleExecutor è una public final class che implementa executor.

Ha un metodo privato ThreadPool threadPool;

Ha anche una SimpleExecutor(int size) che crea una nuova simple thread pool.

Abbiamo i metodi start e stop dell'executor e ed execute.

4.1.10 Callback

Descritta su un parametro di tipo generico T, quindi se ci arriva una task che produce un numero intero allora faremo una callback con un parametro int. Se abbiamo una callback che produce array di byte, allora produrremo quello.

Il tipo T è il prodotto del parametro del completamento.

Abbiamo poi onSuccess che permette a chi implementa la callback prende il tipo di ritorno di una task per utilizzarla successivamente.

4.1.11 Task

Non estende né implementa niente di diverso.

La execute non prenderà un runnable, ma un task.

Questa interfaccia ha due scopi: uno è permettere di dire quale sia il tipo di ritorno della run, cosa che con runnable non si può fare. Poi permettere alla run di lanciare un qualsiasi tipo di eccezione.

Se vogliamo permettere alla nostra task di lanciare una eccezione di quelle da dichiarare allora dobbiamo permettere alla run di lanciarla, verrà passata la callback tramite onfailure.

4.1.12 Future

E' un'interfaccia con un unico metodo get che ha un unico valore di ritorno di tipo T, che andiamo a definire con il future e quindi ce lo portiamo a dietro.

Se qualcosa dovesse andare storto, la get deve poter lanciare un throwable.

4.1.13 Bean + csv

Java può caricare i file CSV ma a modo suo. Ovviamente bisogna caricarli come tali e utilizziamo un'interfaccia Bean.java per caricarle. I CSV contengono solamente dei dati e Bean è un'interfaccia vuota. Viene detta interfaccia **marcatore** perché se c'è c'è se non c'è non c'è, non avendo metodi. Però viene comodo averla perché legge bean dal file e lavora su oggetti che la contengono.

La lettura del file CSV ci dovrà quindi generare una lista di file Bean, uno per riga.

Il bean conterrà i valori delle colonne del file CSV. Se non ci fosse l'interfaccia bean otterremo una lista di objects, che non è sbagliato, ma è non preciso.

Ref student.java

4.1.14 BeanLoader

Per isolarla nel modo corretto dovremmo creare un altro package *beans* e renderlo *it.unipr.informatica.beans*. Per ora terremo negli esempi bean e beanloader.

4.1.15 Shared Aspect

Vogliamo ricevere un oggetto di una certa classe e costruire un suo proxy della stessa classe che implementa le stesse interfacce e garantire che questo proxy faccia sì che i metodi dell'oggetto vengano evocati in mutua esclusione.

Passiamo quindi allo shared aspect l'oggetto a cui vogliamo dare la mutua esclusione -> target è oggetto passivo. Ne costruiamo una variante che ha accessi in mutua esclusione.
Usa esempio file manager

4.1.16 Logging aspect

File di log vengono prodotti in continuazione e contengono tantissimi messaggi. Vengono archiviati per poi essere consultati in caso di necessità.

Di solito si definiscono almeno 3 livelli di tipi di log: livello informativo (stampiamo solo il fatto che la chiamata al metodo ha dato un risultato), livello di errori (stampiamo anche l'eccezione che abbiamo catturato) e livello debug dove non solo stampiamo le cause di errore ma anche lo stacktrace con tutta la sequenza delle chiamate.

4.1.17 Persistent Aspect + Persistent Handler

Aspect

Mettiamo un metodo statico attach a cui non forniamo un target ma un oggetto iniziale e un nome file da usare come oggetto di persistenza. Se il file esiste, leggi i dati, deserializzali e ritorna l'oggetto contenuto nel file. Se non esiste, ritornami l'oggetto che ti passo come default che diventa la versione vuota da salvare dell'oggetto persistente. Per memorizzare un elenco di libri prendiamo l'aspect chiamiamo attach col nome del file dove vogliamo salvare ed un array list vuota se non esiste. Non mi viene ritornato solo quello, (sennò non saprei dove chiamare rollback o commit) mi ritorna quindi un handler, che gestisce l'aspect. Mi permette di arrivare all'oggetto persistente e di eseguire rollback e commit.

Per serializzare un oggetto target la nostra save costruisce un file output stream sul file che abbiamo specificato

Handler

NON estende serializable quindi non è un oggetto persistente, ma contiene nella interfaccia un oggetto T che sarà quello persistente.

Commit e rollback fanno quello che ovviamente farebbero in SQL o qualsiasi altra cosa su DB. Noi vogliamo che solo il persistent aspect possa costruire il persistent handler. Quindi ne mettiamo una implementazione privata (dell handler) dentro all'aspect. Quindi l'utente vedrà una interfaccia pubblica.

Quello lo vogliamo fare sempre: definiamo le interfacce pubbliche ma le implementiamo privatamente.

E' l'interfaccia base delle interfacce attive. Tutte le interfacce attive estendono active. Avrà un attach che ritorna un active di T. Facendo così leghiamo in qualche modo l'oggetto attivo con le sue chiamate asincrone con l'oggetto passivo che non ha chiamate asincrone.

4.1.19 Active Aspect

Abbiamo due attach, il terzo argomento è la dimensione del pool di thread che diamo a disposizione dell'oggetto attivo. Fino a 10 chiamate vengono eseguite in modo concorrente, dopo di che saranno in attesa. La prima chiamata è un ponte verso la seconda -> che ci dice che serve un primo argomento (descrittore di classe) che descrive una classe A. Il secondo argomento è un target di classe T. Vogliamo che A estenda active di T. Facendo così riusciamo a dire che il parametro T del target deve contenere /estendere delle interfacce. Tra queste interfacce ce ne vuole una che estenda active di T. I parametri generici sono 2. T potrà essere qualsiasi. L'idea è che A estenda active di T.

4.1.20 SimpleBook.java + book.java

Book è un'interfaccia che descrive l'interfaccia e la relazione. SimpleBook implementa il tutto. Di solito ci si appoggia su un DBMS ma noi memorizziamo in memoria.

Presa l'interfaccia book, è una interfaccia semplice che avrà una sua implementazione in oggetti valore di tipo simple book. Book descrive le caratteristiche dei libri mentre simple book realizza una classe che permette di istanziare dei libri. Book estende cloneable e mette pubblico clone ed estende serializable -> ci avviciniamo alla definizione di oggetto valore -> un oggetto che implementa tostring equals ecc, è cloneable ed è serializzabile. È un oggetto che trattiamo come

dato primitivo, come un gruppo di dati, non un oggetto che offre servizi. Un oggetto valore spesso è immutabile, non in questo caso, ma va specificato.

Il codice di test va affiancato al codice da testare.

Automatizziamo tutti i metodi di simple book usando simple book test. Di solito si fa un progetto parallelo che avrà gli stessi package e costruirà le classi test nel progetto di test.

Le test suite di solito si fanno una per package. Bisogna abituarsi a farlo ogni qualvolta si raggiunge un punto dove siamo soddisfatti e vogliamo testare le implementazioni delle interfacce.

4.1.21 RemoteException + remote handler + remote request + remote response Exception che usiamo per il remote aspect e per esempio 15. Ovviamente una chiamata remota può causare una eccezione e questa può essere una IOException. Spesso vogliamo evitare di lanciarla subito per vedere se poi c'è bisogno. Sono delle runtime exception e possiamo lanciarle senza dichiararle, ma contengono una IO exception che sono la vera causa.
Server irraggiungibile -> il socket ritorna una IO exception -> non possiamo lanciarla perché dovremmo dichiararla -> sol: la inglobiamo passandola nel costruttore dentro una runtime exception e poi la lanciamo. Tutte le eccezioni sono serializzabili.

→ *to handler*

Quando costruiamo un oggetto e lo rendiamo remoto tramite l'aspetto otteniamo un remote handler che permette di attivare (fare bind su porta e aspettare chiamate) o interrompere

l’oggetto remoto disattivandolo (si sgancia dalla porta non riceve chiamate e ritorna un oggetto qualunque). Lo usiamo per start and stop semplicemente.

→ *to request + response*

Dobbiamo cercare di evitare di mandare dati che magari non sono utilizzabili dall’altra parte. Usiamo oggetti di classe method. Quando siamo sul server usando get method, otteniamo il metodo sul server (se inviamo solo stringhe). Se l’eccezione è nulla lanciala, altrimenti c’è un risultato. Ovviamente mancano tutte le implementazioni.

4.1.22 Processor.java

Gli si fa estendere la classe AbstractProcessor, di modo che abbiamo già qualcosa.

Viene chiamata poi la init quando viene costruito. E’ il metodo di inizializzazione (che non facciamo nel costruttore ma nel metodo init). Gli viene passato il processor environment, l’oggetto che descrive le info che il compilatore si sta portando a dietro.

Il builder genera codice Java. Passiamo il builder e l’elemento su cui lavoriamo alla classe che genera metodi attivi, che va a generare nel builder i metodi attivi. Fatto questo andiamo a generare il file.

4.2 Examples

4.2.1 Example 01

Non ci sono import.

Class Example01 + private boolean stop = false.

void go() -> contiene un nuovo Thread looper e uno waker.
Quando chiamata, invoca start() su looper e su waker. looper
è THIS loop e waker è THIS wake

void loop() -> printa che il looper ha iniziato. fa un try catch
con synchronized THIS e while stop = false va in wait.

void wake() -> Wake starta, sleep per 2 sec poi stoppa e
synchronized mette stop in true e manda una notifica. C'è un
catch alla fine del blocco per interrupted

main() -> chiama la funzione go()

Looper si attiva, va in wait() finché stop è su false e poi
continua e stampa stopped.

Wake si mette in attesa per 2 secondi poi imposta stop a true e
poi scrive a video stopping. Un thread va in attesa e aspetta,
l'altro fa cose. Coordinazione fra due thread. Quando uno
termina e l'altro gli dice di terminare termina. C'è del busy
waiting.

L'idea è che il thread looper, non va in stop prima di
terminare, ma sta aspettando l'altro thread che finisce quello
che deve fare, POI continua e finisce.

La loop verifica stop e rimane lì finché lo stop è false, la wake
parte dopo (2s) così siamo sicuri che non metta subito in stop.

4.2.2 Example 02 + producer + consumer

usa 4.1.2 LinkedBlockingList

Example 02

```
l13. BlockedQueue<String> queue = new  
LinkedBlockingQueue
```

Costruiamo 5 consumatori e gli diamo un ID e prendono la coda. La coda è l'oggetto di coordinazione. Il produttore implementa runnable e lo usiamo per attivare i thread. I produttori partono e si accodano aspettando dati nella coda. 5 produttori scrivono nella coda e scrivono FIFO e 5 consumatori leggono.

E' da linked blocking queue che vediamo come un produttore sveglia 1 consumatore se la queue è == 1. Sono poi i consumatori a svegliarsi l'un l'altro se dopo aver rimosso l'oggetto la coda è > 0.

Producer

Crea il produttore e verifica che gli identificativi abbiano senso, altrimenti lanciano una eccezione. Se queue è nulla lanciamo un eccezione, non possiamo produrre dati senza una coda.

Implementiamo run() che produce 5 messaggi. Stampa il messaggio con i dati e poi accoda il messaggio.

Richiedono solo la mutua esclusione ma non si bloccano mai.

Consumer

Ricevono gli stessi argomenti, un oggetto coda, i check e poi parte la run() che scoda 5 elementi. 5 p x 5 c, tutte le run terminano sicuramente.

I consumatori si bloccano solo quando la coda è vuota.

4.2.3 Example 03 + producer + consumer

Praticamente è l'esempio 2 ma usiamo una Array Blocking Queue invece della Linked List così possiamo utilizzare il reentrant lock e la condition. Lo usiamo principalmente per capire se ci fossero dei deadlock.

4.2.4 Example 04

Ci serve per fare un esempio di atomic reference.

Costruiamo una atomic reference ad una stringa (dentro al go). Abbiamo quindi S new AtomicReference<String> con valore iniziale "Start".

Vogliamo quindi modificare la stringa contenuta nell'atomic reference. Per fare una modifica in senso atomico possiamo fare una Set ma la facciamo usando una GetAndSet.

Tutte le volte che abbiamo un Integer possiamo convertirlo a intero.

Cambiamo la stringa e mettiamo un integer pari a 0. Dopo di chè possiamo aggiornare il suo valore usando un do while che finché *old* è minore di 10 aggiorna la atomic reference fino a raggiungere il valore richiesto.

In Java non esistono riferimenti a metodi ma solamente riferimenti ad oggetto. Quindi quando facciamo riferimento ad un metodo stiamo in realtà creando un oggetto e stiamo facendo boxing creando una inner class.

NB: l'unary operator non ha codice dentro.

4.2.5 Example 05

Con download manager

Non è altro che una classe che crea un nuovo download manager, ha la funzione go che chiama la start del download e dice di scaricare diverse pagine tra cui google, youtube, amazon ecc. Il file è molto semplice, il 99% del codice sta in DownloadManager.java.

I dati non arriveranno in ordine, arriveranno i primi i cui pacchetti sono completi. Dipende dall'ordine in cui i pacchetti arrivano dalla rete.

4.2.6 Example 06

Uguale a example 05, contiene il download manager della versione precedente. Unica differenza: questa volta fornisce i risultati. Al posto di essere void adesso la funzione.

Quando un future viene ritornata da una execute possiamo fare una get. Due effetti: se è tutto nullo si blocca (mi metto in attesa del risultato). Quando arriva il risultato si sbloccano e viene ritornato il risultato (quindi get sarà metodo T) .

Quando viene messe l'eccezione sul future il get si blocca e lancia l'eccezione.

4.2.7 Example 07 + 08

Simile alla 06 e 05, questa è una download che scrive una sotto l'altra le dl. Non ha parametro aggiuntivo ma ritorna un future per ogni download, relativo al valore di quella chiamata li. Facendo le chiamate tutti gli scaricamenti partono in concorrenza. Per accedere al risultato dello scaricamento faccio get sul future correlato.

La versione 08 utilizza la fu ture pool.

4.2.8 Example 09

In questo esempio, chiedo all’utente da tastiera il nome di una classe disponibile nella VM e l’utente potrà metterci una delle classi std della libstd java.lang.object, una delle interfacce standard della library di java java.lang.runnable, oppure potrà mettere le classi della nostra applicazione (che ha una classe sola ed è classe09).

A questo punto vogliamo che vengano stampate alcune informazioni riguardanti la classe.

Come si chiamano e che tipi hanno gli attributi, i metodi ecc?
Andiamo quindi a fare una “introspezione” della classe (sbagliato perché si fa sugli oggetti) ma comunque lo facciamo su una classe. In C++ questa cosa non si può fare.

Abbiamo la go() come al solito e la classe scanner da istanziare di java util viene usata per passare lo stream di dati.
Scanner scanner = new Scanner(system.in). Lo dobbiamo mettere dentro ad un try catch perché è un oggetto che poi va chiuso.

Chiamiamo poi show() su quello che abbiamo inserito che non fa altro che fare forename e carica la classe che vogliamo in memoria. Se la otteniamo carica un oggetto di classe class con anche un parametro generico che però noi non sappiamo. Possiamo mettere il “?” al posto del tipo, perché giustamente non lo sappiamo.

Quando eseguiamo il programma non possiamo vedere i metodi privati. Il compilatore può decidere di fattorizzarli.

4.2.9 Student.java + Book.java etc (for CSV) + SimpleStudent.java

Questa interfaccia serve come complemento a Bean.java per

implementare i file CSV dentro a Java. Vedendo questa interfaccia vediamo che abbiamo a che fare con un Java Bean (abbiamo che l'interfaccia estende Bean).

Abbiamo anche property int e string per ID e due getter per Name e Surname.

Queste due interfacce sono semplicemente contenitori di dati, che però ora non contengono nulla, perché sono interfacce.

Partiamo sempre dalle interfacce per modellizzare i dati.

La classe Simple invece non è altro che una implementazione delle classi sopracitate.

SimpleStudent implementerà Student e Bean. SimpleBook -> Book e Bean.

Queste interfacce non sono altro che dei metodi **setter** e **getter** per i vari campi dei dati che vogliamo integrare.

Abbiamo un costruttore senza parametri, che in teoria ne avrebbe bisogno ma ci serve comunque -> il nostro Bean parte da uno stato invalido finché non inseriamo i dati.

Altrimenti senza costruttore senza parametri non potremmo usare la reflection.

ToString invece serve ridefinirlo perché nella convenzione dei JavaBean è richiesto che venga ridefinito, visto che tanto i dati li dobbiamo convertire in stringhe.

4.2.10 Example 10

Questo esempio ha la Load dei file -> si aspetta una classe come primo argomento e poi un nome di un file. Verifica che la classe implementi Bean (per nostra convenzione le classi dei nostri JavaBean devono implementare Bean). Se questo

non è possibile lanciamo una eccezione. A questo punto leggiamo un file di testo usando lo scanner con System.in (input stream collegato allo standar input). Nessuno ci vieta però di fare un file input stream -> lo costruiamo, lo passiamo allo scanner e questo va a leggere le parole fino al terminatore.

Nella private String[] splitLine possiamo specificare una espressione regolare (in questo caso usiamo \t che è tab) che può agire da separatore per le varie righe per prenderle come parole righe colonne separate.

Chiamiamo poi trim che è comoda per togliere spazi, capital, tab, nella lettera iniziale e finale della stringa. Essendo la stringa immutabile in Java creiamo una copia di quella stringa senza gli “errori” che ci sono nella str originale

Abbiamo quindi ottenuto l’elenco delle property. Chiediamo quindi alla classe i loro tipi (id è intero cognome e nome sono stringhe). Senza questi dati non riusciremmo a fare le set. Object alla fine ci offre pochi metodi: toString, clone, equals, hashCode.

ToString offre una visione testuale dei dati dell’oggetto. Se l’oggetto è di tipo valore quando due oggetti contengono gli stessi dati il toString che ne esce deve essere identico.

Equals controlla che (convenzione) due oggetti siano uguali. L’implementazione standard ne verifica gli indirizzi (ma non qualifica quelli di tipo valore) e equals viene ridefinita.

Essendo SimpleBook final- no problem. Quando si fanno i bean si mettono sempre le classi final per evitare errori.

4.2.11 Example 11 (proxy)

Abbiamo l’interfaccia studente e ha 3 metodi getter. Abbiamo la StudentData.java. Non vogliamo costruire l’implementazione. Vogliamo che l’utente possa avere degli oggetti che implementano questa interfaccia. In C++ questa cosa non ha senso, non si può fare. Come faccio ad avere una interfaccia senza metodi con oggetti che la implementano a runtime? Lo chiediamo al dynamic proxy a tempo di esecuzione.

Costruiamo un oggetto studente che implementa l’interfaccia studente senza avere la classe. Chiamiamo metodo statico new proxy instance e passiamo 3 parametri il cui secondo è un array di descrittori di classi che contiene student.class.

Quel secondo parametro è l’insieme delle interfacce che vogliamo vengano implementate dall’oggetto student che abbiamo creato.

Il proxy è un oggetto vicino che ci permette di parlare con un oggetto lontano. Ha questo nome proprio perché dovrebbe inglobare un oggetto lontano e permetterci di parlarci aggiungendo delle funzionalità.

Costruisci un oggetto di classe studentData e attaccarci un logger (dice quanto si entra ed esce dal metodo) poi viene chiamato getId. Il proxy permette di intercettare delle chiamate ad oggetti di classi che abbiamo già costruito ma in cui non vogliamo mettere delle funzionalità (che sarebbero aggiuntive). Aggiungere per esempio logging dentro a student data non ha senso. Ci permettono di aggiungere funzionalità ad oggetti costruiti con classi che non dispongono di quelle capacità.

4.2.12 Example 12 File manager + simple

Costruisce un oggetto file manager -> ha due metodi entrambi messi pubblici nella interfaccia. Uno dà la lista dei nomi dei file contenuti in una certa cartella da specificare (con eccezione in caso) e l'altro va a leggere i byte dei dati. Andiamo quindi a leggere i dati direttamente dal disco. L'interfaccia descrive la superficie di scambio di dati dei risultati delle eccezioni.

Faremo anche l'implementazione poi.

La implementazione è SimpleFileManager.java

Alla fine quello che fa è sostituire la lock and unlock dello shared aspect con una print di ingresso e una print di uscita. Ovviamente possiamo attaccare il logging aspect al file manager originale ma possiamo anche attaccarlo al file manager esteso con le proprietà di essere condiviso.

4.1.13 Example 13

Collegato a Handler e aspect di persistent.

I file .dat contengono solamente dei file dati, non è un formato particolare.

Sono dati non testuali però comunque di tipo “dato”.

Questo programma costruisce un file che si chiama books.dat a cui attacchiamo all’oggetto il persistent aspect (se esiste possiamo leggerlo). Se il file non dovesse esistere ci verrà ritornata nell’handler la array list e potremmo iniziare a lavorarci sopra.

Se books è vuoto allora salta una parte, altrimenti stampa l’elenco dei libri. In entrambi i casi aggiungi dei nuovi libri (casuali) e poi fai commit.

4.1.14 Example 14 + ActiveDownloadManager

Estende Active, ma è uguale alla download manager. Avrà start e stop e potremo collegare il tipo del target col tipo dell’interfaccia attiva. A questo punto aggiungiamo i metodi asincroni con tanto di eccezioni.

Fa poi partire la download asincrona che ritorna un future e quindi poi facendo la get riusciamo ad avere il risultato. Da lì in avanti è uguale ai vecchi download manager.

4.1.15 Example 15 Server + Client

Se dovessimo lanciare il client prima del server, ovviamente otterremo una connection refused perché non c’è un server attivo in attesa.

Lanciando prima il server poi il client, il server si mette in attesa e poi il client fa quello che deve fare. Basta conoscere l’IP e che la nostra rete ci faccia arrivare - non scontato - e se i messaggi arrivano, chiediamo alla macchina che file ha a disposizione e ce li facciamo inviare. Non abbiamo costruito messaggi, abbiamo chiamato dei metodi che inviano dei messaggi.

Example 16 - Annotations

Bisogna andare nelle proprietà del progetto, su annotation processing e abilitarlo. Facendo così tutte le annotazioni che creeremo verranno utilizzate dal compilatore. Finché non facciamo i test case JUnit (finché l’annotation processor non genera file) la cartella non viene costruita. In Factory Path bisogna scegliere i JAR dell’annotation processor.

