Indice

Torna all'indice

Tautologie

> *Una tautologia, in logica, è un'affermazione vera per definizione, quindi fondamentalmente priva di valore informativo.*

Una proposizione A e' una tautologia se e soltanto se $I \models A$ per ogni possibile interpretazione I. Tutte le interpretazioni sono modelli per A.

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

Contraddizioni

Una proposizione A e' una contraddizione (proposizione insoddisfabile) se e soltanto se nessuna interpretazione e' modello per A. Esempio: $\neg \;(\;p\to (\;p \vee q \;))$

Una proposizione A e'

- una tautologia se e soltanto se $\neg A$ e' una contraddizione
- una contraddizione se e soltanto se $\neg A$ e' una tautologia

Torna all'indice

Equivalenze logiche

Le proposizioni A e B sono equivalenze logiche ($A \iff B$) se e soltanto se $\models (A \equiv B)$. Per ogni interpretazione I, $I \models A$ se e soltanto se $I \models B$.

Conseguenze logiche

Un'interpretazione I e' un modello per un insieme di proposizioni S se e soltano se e' un modello per ogni proposizione in S. Un insieme di proposizioni che non ha modelli si dice insoddisfacibile. La proposizione A e' una conseguenza logica di un insieme di proposizioni $S \;(S \models A)$ se e soltanto se tutti i modelli I per S sono anche modelli per A.

> *Nota bene: $S \models A$ non implica che tutti i modelli di A sono anche modelli di S.*

Torna all'indice

Trableaux proposizionale

Un tableaux proposizionale (semantico) e' un albero in cui i nodi sono etichettati con un insieme di proposizioni che usano le seguenti regole finche' esse siano applicabili:

> *Nota bene: un tableau proposizionale e' sempre finito. *

I tableau proposizionali sono usati per verificare la satisfiability.

- Un percorso dalla radiche di un tableau completo verso una foglia e' detto chiuso se la foglia e' contadditoria, altrimenti e' detto aperto.
- Un tableau e' detto chiuso se tutti i suoi percorsi sono chiusi.
- L'insieme di proposizioni che identifica la radice di un completo tableau è 0. unsatisfiable se e solo se il tableau è chiuso
- L'etichetta della foglia di un percorso aperto identifica un insieme di modelli dell'insieme di proposizioni che etichetta la radice

Un insieme di proposizioni S e' unsatisfiable se e soltanto se tutte le foglie del tableau sono marcate come contradditorie.

Dato un insieme di proposizioni S e una proposizione A, per verificare che $S \models A$, e' sufficiente considerare $S' = S \cup \{\neg A\}$ e provare, costruendo un tableau, che S' sia unsatisfiable.

Torna all'indice

Forma negata

Una proposizione e' nella forma negata se:

- Solo congiunzioni, disgiunzioni e negazioni sono usate nella proposizione
- Le negazioni si verificano solo nei letterali (ricordiamo che un letterale e' un simbolo proposizionale o la sua negazione)

Qualsiasi proposizione puo' essere trasformata in una equivalente proposizione in forma negata usando le equivalenze logiche.

Forme congiuntive e disgiuntive

Una proposizione e':

- In forma congiuntiva normale (CNF) se e' in forma negata ed e' strutturata come una congiunzione di disgiunzioni di letterali (ci sono solo AND).
- In forma disgiuntiva normale (DNF) se e' in forma negata ed e' strutturata come una disgiunzione di congiunzioni di letterali (ci sono solo OR).

Qualsiasi proposizione puo' essere trasformata in una equivalente proposizione in forma congiuntiva o disgiuntiva usando le equivalenze logiche.

Torna all'indice

Logiche temporali

Nelle logiche classiche, le proposizioni sono interpretate staticamente e sono interpretate con lo scopo di essere in un unico mondo statico. Le interpretazioni assegnano staticamente i valori alle proposizioni e non possono cambiare nel tempo.

Nelle logiche temporali, invece, l'interpretazione avviene nell'ambito di un insieme di mondi.

> *Trasformiamo il tempo in una variabile discreta.*

L'insieme dei mondi che caratterizzano le logiche temporali corrispondono ai momenti nel tempo. Il particolare modello di tempo per una data logica temporale è catturato da una relazione di accessibilità (temporale) tra i mondi.

Le logiche temporali estendono la logica proposizionale con operatori temporali (modali) per navigare nei mondi utilizzando la relazione di accessibilità della logica scelta.

Esistono due modelli per esprimere il tempo:

Torna all'indice

Logica temporale lineare (LTL)

La logica temporale lineare ha una relazione di accessibilità che descrive un modello lineare e discreto del tempo isomorfo ai numeri naturali.

Dato un software che ci assicura che:

- G(requested \(\to\) F received)
- G(received \(\to\) X processed)
- G(processed \(\to\) FG done) Se abbiamo fatto bene queste 3 cose, allora la seguente affermazione sara' sempre falsa: G requested \(\wedge\) G \(\neg\) done.

Torna all'indice

Sintassi e Semantica

La sintassi della LTL e' una semplice estensione della sintassi delle logiche proposizionali. I connettivi binari sono associativi a sinistra. La precedenza dei connettivi e operatori segue il seguente ordine: \(\neg, \;G, \;F, \;X, \;U, \;\wedge, \;\vee, \;\to, \;\equiv\)

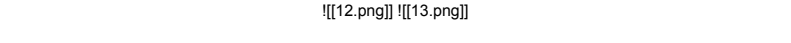
La semantica delle proposizioni LTL e' basata su:

- Ogni momento del tempo e' rappresentato da un numero naturale.
- Per ogni momento del tempo, un mondo e' rappresentato da un insieme di rappresentazioni LTL che sono vere.

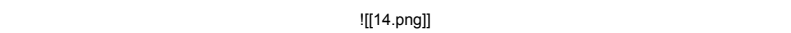
La funzione di interpretazione $I : P \times \mathbb{N} \to B$ dove $B = \{ F, T\}$, mappa ogni simbolo proposizionale a un valore di B per ogni momento nel tempo.

> *Nel caso in cui $\mathbb{N}$ sia 0, indichiamo il momento "adesso". O e' vera o e' falsa, non esiste il "forse".*

Data una interpretazione I su P, l'interpretazione $G_I : LTL[P] \times \mathbb{N} \to B$ di un'arbitraria proposizione LTL in $LTL[P]$ puo' essere calcolata come segue:



Semantica degli operatori temporali



Torna all'indice

Soddisfacibilità e Tautologie

Data una interpretazione M, un momento nel tempo $i \in \mathbb{N}$, e una proposizione LTL A:

- $<M,i> \models A$ (M soddisfa A in i) se e solo se $G_M(A,i)=T$
- $<M,i> \nvDash A$ (M non soddisfa A in i) se e solo se $G_M(A,i)=F$

Un'interpretazione M e' un modello per una proposizione A se e solo se esiste qualche $i \in \mathbb{N}$ tale che $<M,i> \models A$.

Una proposizione LTL A e':

- Soddisfacibile se e solo se esiste un modello per A.
- Una tautologia ($\models A$) se e solo se per ogni interpretazione M e ogni momento nel tempo $i \in \mathbb{N}$, $<M,i> \models A$.

Modelli

Data una interpretazione M definita su un insieme di simboli proposizionali P, le seguenti regole possono essere usate per verificare se $<M,i> \models A$ (M soddisfa A in i):

Torna all'indice

Equivalenze logiche

Due proposizioni LTL A e B sono equivalenti logiche ($A \Leftrightarrow B$) se e solo se $\models(A \equiv B).$

> *Per ogni interpretazione M e in ogni momento del tempo $i \in \mathbb{N}$, $<M,i> \models A$ se e solo se $<M,i> \models B$.*

Altre equivalenze logiche:

Torna all'indice

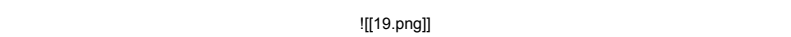
Tableaux LTL

I tableaux LTL sono grafi diretti (non alberi) usati per verificare la soddisfacibilità di un insieme di proposizioni LTL. Un tableaux LTL e' un grafo etichettato, il quale non contiene nodi con la stessa etichetta. Un nuovo nodo non viene aggiunto al grafo se la sua etichetta appare in un altro nodo.

> *$\{A,B\} = \{A,B,A,A,B,B\}$ stessi insiemi.*

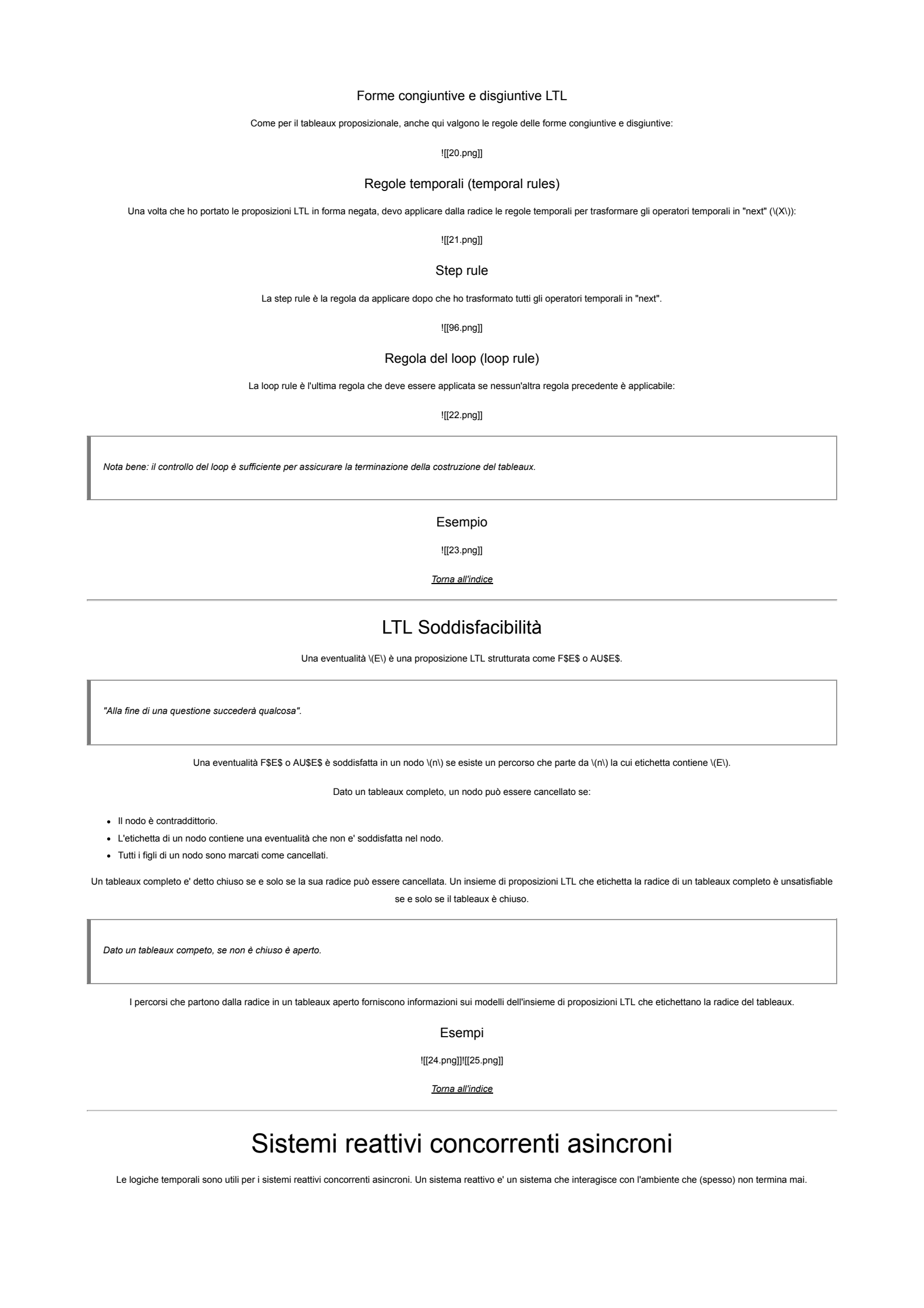
Forma negata LTL

Come prima cosa dobbiamo trasformare le proposizioni LTL in forma negata usando le seguenti equivalenze logiche:



Forme congiuntive e disgiuntive LTL

Come per il tableaux proposizionale, anche qui valgono le regole delle forme congiuntive e disgiuntive:



Regole temporali (temporal rules)

Una volta che ho portato le proposizioni LTL in forma negata, devo applicare dalla radice le regole temporali per trasformare gli operatori temporali in "next" (\(X\)):

Step rule

La step rule è la regola da applicare dopo che ho trasformato tutti gli operatori temporali in "next".

Regola del loop (loop rule)

La loop rule è l'ultima regola che deve essere applicata se nessun'altra regola precedente è applicabile:

> *Nota bene: il controllo del loop è sufficiente per assicurare la terminazione della costruzione del tableaux.*

Esempio

Torna all'indice

LTL Soddisfacibilità

Una eventualità \(E\) è una proposizione LTL strutturata come FE o AUE.

> *"Alla fine di una questione succederà qualcosa".*

Una eventualità FE o AUE è soddisfatta in un nodo \(n\) se esiste un percorso che parte da \(n\) la cui etichetta contiene \(E\).

Dato un tableaux completo, un nodo può essere cancellato se:

- Il nodo è contraddittorio.
- L'etichetta di un nodo contiene una eventualità che non e' soddisfatta nel nodo.
- Tutti i figli di un nodo sono marcati come cancellati.

Un tableaux completo e' detto chiuso se e solo se la sua radice può essere cancellata. Un insieme di proposizioni LTL che etichetta la radice di un tableaux completo è unsatisfiable se e solo se il tableaux è chiuso.

> *Dato un tableaux competo, se non è chiuso è aperto.*

I percorsi che partono dalla radice in un tableaux aperto forniscono informazioni sui modelli dell'insieme di proposizioni LTL che etichettano la radice del tableaux.

Esempi

Torna all'indice

Sistemi reattivi concorrenti asincroni

Le logiche temporali sono utili per i sistemi reattivi concorrenti asincroni. Un sistema reattivo e' un sistema che interagisce con l'ambiente che (spesso) non termina mai.

> *Questi sistemi sono anche chiamati agenti.*

Un sistema concorrente consiste in un insieme di parti che vengono eseguiti concorrentemente.

La modellazione di un sistema ha lo scopo di costruire una specifica (possibilmente formale) del sistema per eliminare dettagli irrilevanti. La specifica e' descritta in termini di:

- Stato di un sistema, il quale è uno snapshot dei valori dei parametri che caratterizzano il sistema.
- Transizioni del sistema, che descrivono come lo stato di un sistema cambi nel tempo a seguito di azioni ed eventi.
- Computazione del sistema, che è la sequenza (possibilmente infinita) degli stati attivati dalle transizioni.

Strutture Kripke

Sono diagrammi di transizione che descrivono l'ambiente dinamico di un sistema reattivo. Una struttura Kripke consiste in:

- Un insieme non vuoto di stati.
- Un insieme non vuoto di transizioni attraverso gli stati.
- Un insieme non vuoto di proposizioni che etichettano gli stati.

Un percorso in una struttura Kripke rappresenta una possibile computazione del sistema descritto (una struttura descrive un processo).

> *Più strutture rappresentano un programma.*

Formalmente, una struttura Kripke è una 5-upla $K=\;<S,I,R,P,L>\;$, in cui

- S è un insieme non vuoto di stati,
- $I\subseteq S$ e' un insieme non vuoto di stati iniziali,
- $R \subseteq S\times S$ e' una relazione di accessibilita', un insieme non vuoto di transizioni tale che R sia left-total $\forall \; s \subseteq S, \exists \; s' \subseteq S : (s,s') \subseteq R$
- P sia un insieme numerabile di simboli proposizionali usati per costruire $Prop[\;P\;]$,
- $L : S \to 2^{Prop[P]}$ è una funzione di etichettatura ($2^{Prop[P]}$ rappresenta l'insieme delle proposizioni).

Un percorso π in una struttura Kripke K da uno stato $s_0\subseteq S$ e' una sequenza infinita di stati $\pi = s_0 \; s_1 \; s_2 ...$ tale che $(s,s') \subseteq R$ per tutti $i \subseteq \mathbb{N}$ (perche' R e' left-total).

Torna all'indice

Strutture Kripke complesse

Le strutture Kripke complesse le otteniamo da una composizione di strutture Kripke semplici. Le strutture Kripke possono essere combinate usando composizioni:

- Sincrone:

  ![[26.png]]

- Asincrone:

  ![[27.png]]

Torna all'indice

Safety, Liveness, Fairness

Gran parte della popolarità della logica temporale è stata raggiunta perché diversi concetti relativi ai sistemi software concorrenti possono essere formalmente espressi e studiati. Di solito, le proprietà dei software concorrenti che vengono studiate usano logiche temporali e sono suddivise in 3 categorie:

- Proprieta' safety (affidabilita').
- Proprieta' liveness.
- Proprieta' fairness.

Le proprieta' safety sono richieste per assicurare che qualcosa di brutto non avvenga mai. $G\; \neg \;(\text{ temperature} > 100 \;)\;$ Le proprieta' liveness sono richieste per assicurare che qualcosa di buono succedera'. $G\;(\text{started} \to F\; \text{terminated})\;$ Le proprieta' fairness (forti) sono richieste per assicurare che se qualcosa viene richiesto infinitamente spesso, allora verrà servito infinitamente spesso. $GF\text{ ready} \to GF\text{ execute}\;$

Esempi proprieta' con la mutua esclusione

1. **Safety **

   ![[28.png]]

2. **Liveness**

   ![[29.png]] ![[30.png]]

3. **Fairness**

   ![[31.png]]

4. **Fairness forte (strong) **

   ![[32.png]]

Torna all'indice

Indice

Sistemi concorrenti

Un software si dice concorrente se la sua computazione è ottenuta dalla composizione di sotto-programmi eseguiti in modo sequenziale indipendenti l'uno dagli altri, i quali possono essere eseguiti in parallelo sotto regole specifiche che impongono una sequenzializzazione delle parti che li compongono. La computazione di un sistema concorrente è strutturato nei termini di un insieme di flussi eseguiti indipendentemente.

> *Ogni flusso di esecuzione effettua una computazione sequenziale.*

La descrizione statica di un software concorrente è descritta nei termini di un insieme di programmi concorrenti:

- Ogni programma concorrente descrive alcuni dei flussi di esecuzione del sistema concorrente.
- Ogni programma concorrente descrive alcuni dei vincoli di sequenzializzazione dovuti durante i flussi di esecuzione.

La descrizione delle computazioni sequenziali associate con i flussi di esecuzione sono presenti in un programma che usa un linguaggio di programmazione. Se il linguaggio di programmazione adottato permette la descrizione esplicita oltre ai vincoli di sequenzializzazione dovuti, allora questo linguaggio è un linguaggio di programmazione concorrente.

Torna all'indice

Sistemi concorrenti in Java

Java è un linguaggio di programmazione concorrente. Offre costrutti del linguaggio per esprimere i vincoli di sequenzializzazione basici. Pattern complessi di sequenzializzazione possono essere espressi come la combinazione di diversi costrutti.

Processi

Un processo (concorrente) è un programma (concorrente) in esecuzione:

- Un programma è un documento statico che descrive il comportamento in fase di esecuzione di un processo.
- Un processo è una entità dinamica che esegue un programma usando un insieme particolare di dati e risorse.
- Due o più processi possono eseguire lo stesso programma, ognuno dei quali usando le proprie risorse e i propri dati.

Un processo comprende sempre i seguenti componenti:

1. Un programma da essere eseguito.
2. I dati grazie ai quali il programma verra' eseguito.
3. Le risorse richieste da un processo a tempo di esecuzione.
4. Lo stato di un processo in esecuzione.

Un processo (in Java) viene eseguito in un ambiente astratto che gestisce i dati condivisi e isolati e le risorse attraverso la "community" dei processi. La Java Virtual Machine (JVM) offre questo ambiente astratto per i processi.

> *Una JVM per ogni processo.*

![[33.png]]

Torna all'indice

Threads

Un thread è il flusso di esecuzione di un processo:

- Ogni thread è associato con un singolo processo.
- Molti thread possono essere associati ad un singolo processo.
- Un thread esegue parti di un programma associati con il suo processo e condivise dati e risorse con quest'ultimo.
- Un thread alloca dinamicamente parti delle risorse di un processo per il suo fabbisogno computazionale.
- Ogni thread ha la sua area privata per salvare dati e strati.

Un processo sequenziale ha un singolo thread. Una memoria condivisa è visibile a tutti i threads di un processo.

> *Condividono tutto tranne alcuni dati privati (es. PID).*

I thread sono spesso chiamati processi leggeri perché il carico di lavoro associato alla loro computazione e' ridotto rispetto a quello dei processi.

![[34.png]]

I thread hanno le seguenti proprietà:

1. Un thread inizia la sua esecuzione in uno specifico punto del programma.
2. Un thread esegue una sequenza di istruzioni ordinata e predefinita.
3. Un thread viene eseguito indipendentemente rispetto agli altri thread.
4. I thread sono nati per essere eseguiti in parallelo (se la macchina che li esegue lo permette).

![[35.png]]

Torna all'indice

Parallelismo e concorrenza

I sistemi paralleli sono deployati su un insieme di CPU per eseguire in parallelo processi e thread nello stesso momento. I sistemi concorrenti sono deployati su una singola CPU, ma sono strutturati come se fossero in un sistema parallelo.

![[36.png]]

Torna all'indice

Thread in Java

Ci sono due modi per descrivere i thread in Java:

1. Estendendo la classe `java.lang.Thread` usando top-level classes o (anonime) inner class.

```
public class ThreadA extends Thread {
        @Override
        public void run() {
                // Corpo del thread
        }
}
```

2. Implementando l'interfaccia `java.lang.Runnable` usando top-level classes, (anonime) inner class, lambda expression.

```
public class ThreadB implements Runnable {
        @Override
        public void run() {
                // Corpo del thread
        }
}
```

In entrambi i casi, il metodo `run()` contiene la sezione del programma che il thread deve eseguire. Il metodo `start()` è usato per "startare" un thread e avviare l'esecuzione del metodo `run()`.

```
// Estendendo la classe Thread
Thread a = new ThreadA();
a.start();

// Implementando l'interfaccia Runnable
Thread b = new Thread(new ThreadB());
b.start();
```

Un thread termina quando il suo metodo `run()` termina: non puo' essere terminato "con la forza".

Torna all'indice

Stati di un thread

Gli stati del ciclo di vita di un thread sono strutturati in tre livelli:

![[37.png]]

Gli stati di un thread possono essere alterati:

- `sleep(millis)`: forza il thread ad andare nello stato blocked per un numero specifico di millisecondi.
- `interrupt()`: forza il thread ad andare nello stato running.
- `interrupted()`: verifica se il thread è stato precedentemente bloccato.
- `yield()`: mantiene il thread nello stato running, ma avvisa il thread scheduler di dare il permesso agli altri thread per essere eseguiti.
- `join()`: forza il thread ad andare nello stato blocked e aspetta gli altri thread sul quale e' stato invocato di terminare.

In una singola CPU, i thread vengono eseguiti uno alla volta dando l'illusione di essere in parallelo. La JVM implementa un algoritmo di scheduling veramente semplice per i thread chiamato *fixed priority scheduling*.

Torna all'indice

Java Memory Model

I thread di un programma Java:

- Hanno uno stack privato che usano come supporto per le invocazioni dei metodi.
- Hanno un provato thread-local storage.
- Condividono l'oggetto heap.

La Java Memory Model descrive come i thread accedono ai loro tre tipi di memoria, e descrive come il contenuto delle memorie è effettivamente salvato nella gerarchia di memoria del sistema.

> *JVM decide come eseguire i thread sui vari core. Tutti gli oggetti risiedono nell'heap. Solo i tipi primitivi stanno nello stack.*

![[38.png]]

La JVM è un "sistema operativo". I thread in Java non sono necessariamente eseguiti sulla stessa CPU, questo perchè la JVM normalmente è distribuita su più CPU.

![[39.png]]

L'esecuzione di thread su multiple CPU può causare problemi di coerenza di memoria. Un esempio è la variabile `done` che abbiamo visto nell'esercizio 1: viene copiata nelle cache dei vari core e può assumere valori diversi in base ai vari thread. Questo provoca inconsistenza.

Questi problemi vengono risolti se l'accesso ai dati condivisi è controllato.

> *Un controllo dell'accesso alla memoria condivisa può causare un uso inefficiente del parallelismo e delle risorse stesse.*

> *La memoria in Java è di 64MB. Quando si sfora questa cifra entra in azione il garbage collector.*

![[40.png]]

Torna all'indice

Mutua esclusione in Java

Il problema della mutua esclusione è uno dei piu' classici problemi della programmazione concorrente. Un programma concorrente incontra un problema di mutua esclusione quando si deve assicurare che se, dato un insieme \(M\) di sezioni mutualmente esclusive di un programma,

- solo un thread alla volta può eseguire una delle sezioni del programma in \(M\);
- i thread che non possono eseguire le sezioni mutualmente esclusive in \(M\) sono bloccati e verranno ripresi il prima possibile.

> *L'assegnamento degli oggetti in Java non esiste: viene copiata la reference. L'assegnamento avviene solo per i tipi primitivi.*

La mutua esclusione in Java è risolta utilizzando le sezioni critiche. Una sezione critica è una sezione di un programma associata ad un oggetto usato come un mutex (Mutual Exclusion device) per controllare l'accesso nella sezione critica.

- Un mutex puo' essere acquisito e rilasciato dai thread: è posseduto da un thread dopo la sua acquisizione e prima del rilascio.
- Se un thread acquisisce un mutex, nessuno degli altri thread puo' acquisire quel mutex finche' non viene esplicitamente rilasciato.
- I thread sono forzati ad andare nello stato blocked quando non possono acquisire un mutex, e successivamente sono forzati a passare allo stato running quando possono finalmente acquisirlo.
- Un thread esegue la sua sezione critica solo quando possiede il mutex usato come guardia per quella sezione.
- Un mutex è un oggetto qualsiasi dell'heap.
- Posso avere sezioni critiche diverse concorrenti che usano mutex diversi.

> *L'acquisizione e il rilascio del mutex devono essere vicini per evitare molti bug.*

L'acquisizione e il rilascio sono:

- Operazioni rientranti perché un mutex può essere acquisito e rilasciato più volte in cicli annidati senza bloccare il thread che possiede il mutex.
- Operazioni sincrone all'interno del processo perché i loro effetti sono condivisi in modo sincrono tra tutti i thread del processo, possibilmente tra più CPU.

> *Java garantisce la mutua esclusione anche su CPU diverse.*

![[41.png]]

Una sezione critica e' identificata da:

- Il modificatore `synchronized` dei metodi, per affermare che il corpo del metodo è la sezione critica e che questa fa riferimento all'oggetto da utilizzare come mutex. In questo modo il mutex e' l'oggetto `this`.

```
public synchronized void myMethod() {
        // the critial section
}
```

- Il blocco `synchronized`, per affermare che il corpo del blocco è la sezione critica e che l'oggetto a cui si fa riferimento nella testa del blocco viene utilizzato come mutex. Con questo metodo identifico la guardia / mutex con l'oggetto `o`.

```
public void myMethod(Object o) {
        synchronized(o) {
                // the critial section
        }
}
```

La disponibilità delle sezioni critiche in Java provvedono una effettiva soluzione per diversi problemi, tra cui:

- Interferenza tra thread: questo problema accade quando due operazioni, che vengono eseguite su diversi thread, agiscono sullo stesso dato causando una race condition.
- Coerenza della memoria: questo problema accade quando thread diversi hanno inconsistenti accessi a copie dello stesso dato attraverso diverse CPU (cache non "allineate").

Torna all'indice

Happens-Before

Java definisce la relazione happens-before sulle operazioni in memoria come la lettura e la scrittura delle variabili condivise. Il risultato di una scrittura di un thread è garantita dall'essere visibile per una lettura di un altro thread se e solo se l'operazione di scrittura happen-before l'operazione di lettura.

Ogni azione in un thread avviene-prima di ogni altra azione di un altro thread che viene dopo in un programma. Lo sblocco di un mutex avviene prima di ogni successivo blocco dello stesso mutex (vincolo multiprocessore).

> *La relazione happens-before e' transitiva.*

Una scrittura su un campo `volatile` avviene prima di ogni lettura successiva dello stesso campo.

> *Si noti che il modificatore `volatile` non comporta la mutua esclusione.*

Una chiamata a `start()` su un thread avviene prima di qualsiasi azione nel thread avviato. Tutte le azioni in un thread si verificano prima che qualsiasi altro thread ritorni con successo da un `join()` su quel thread.

La keyword `synchronized` può essere usata effettivamente per risolvere i problemi di coerenza della memoria. L'utilizzo di `synchronized` su un oggetto `obj` assicura che tutte le modifiche dello stato di `obj` verranno propagate a tutti i thread interessati prima di un qualsiasi successivo accesso sincronizzatoi a `obj`.

> *I campi `final`, che non possono essere modificati dopo che l'oggetto e' stato costruito, possono essere letti tranquillamente e in modo sicuro una volta che l'oggetto e' stato costruito.*

Le operazioni atomiche in Java sono quelle operazioni di lettura e scrittura che non possono essere interrotte per sospendere l'attuale thread e attivarne un altro.

Torna all'indice

Attesa e notifica di eventi

Il problema dell'attesa e della notifica degli eventi è un altro dei classici problemi della programmazione concorrente.

Un thread affronta il problema dell'attesa degli eventi se ha bisogno di attendere gli eventi senza verificare attivamente il verificarsi degli eventi (busy waiting). Un thread affronta il problema di notificare eventi se ha bisogno di comunicare il verificarsi di eventi a un altro thread, che probabilmente è in attesa di notifiche.

Java abbina il problema dell'attesa e della notifica degli eventi alle sezioni critiche perché le sezioni critiche forniscono già un mezzo per bloccare i thread (senza busy waiting).

Un thread che deve notificare eventi invoca il metodo `notifyAll()` su un oggetto `obj` per notificare a tutti gli oggetti che stavano aspettando su `obj` che un evento si è verificato.

Un thread che deve aspettare eventi invoca il metodo `wait()` su `obj` per bloccarsi e aspettare gli eventi.

> *Questi metodi vanno chiamati sugli oggetti guardia altrimenti si verifica una eccezione.*

Nota che:

- Il metodo `notifyAll()` notifica tutti i thread in attesa, che vengono tutti forzati allo stato di esecuzione anche se solo uno di loro entrerà nella sezione critica.
- Il metodo `notify()` notifica solo uno dei thread in attesa, ovvero il thread che eventualmente procederebbe nella sezione critica.
- L'uso del metodo `notify()` è intrinsecamente più efficiente, ma è anche soggetto a errori.
- Il metodo `wait(millis)` può essere usato per aspettare non più di `millis` millisecondi (attesa massima).

Torna all'indice

Problemi di liveness

L'utilizzo superficiale o errato del supporto per la programmazione concorrente fornito da Java può comportare gravi problemi di liveness. Alcuni di questi problemi lo sono:

- Deadlock, che è una situazione in cui due o più thread sono bloccati per sempre, in attesa l'uno dell'altro.
- Livelock, che è una situazione in cui due thread reagiscono continuamente agli eventi che ciascuno notifica all'altro.
- Starvation, ovvero una situazione in cui un thread non è in grado di ottenere un accesso regolare a una risorsa condivisa e, quindi, non è in grado di realizzare i progressi attesi.

Astrazione ad alto livello per la concorrenza

Java offre meccanismi basici per la gestione della concorrenza e dei relativi problemi. L'astrazione ad alto livello per la gestione dei problemi relativi alla concorrenza è necessaria per:

- Migliorare la manutentibilità dei sistemi concorrenti.
- Migliorare la riusabilità delle soluzioni dei problemi concorrenti.
- Migliorare il livello di comprensione delle caratteristiche non funzionali delle soluzioni (es: livello di parallelismo, tipi di controllo, problemi di liveness).

Torna all'indice

Blocking Queues

Una queue (coda) è una sequenza di elementi che cambiano dinamicamente e seguono una politica FIFO (First In First Out). Le azioni basiche delle code sono: creazione, distruzione, is empty test, is full test, enqueue (add item), dequeue (remove item).

Una blocking queue (coda bloccante) è una coda utilizzata per l'uso concorrente. Le operazioni vengono bloccate se non possono essere eseguite immediatamente:

- Enqueue puo' bloccare se la coda è piena.
- Dequeue puo' bloccare se la coda è vuota.

Le code bloccanti sono un'astrazione che possono essere usate per coordinare le attivita' senza un sistema concorrente.

> *Nota bene:*
>
> 1. *Tutte le code bloccanti possono possono bloccarsi sulla dequeue se sono vuote.*
> 2. *Solo le code bloccanti con capacita' limitata possono bloccarsi sulla enqueue se sono piene.*

Torna all'indice

Locks and Conditions

Un lock (esplicito) è un'astrazione che può essere usata per assicurare la mutua esclusione.

> *Svincoliamo l'idea della sezione critica.*

Caratteristiche dei lock:

- Può essere esplicitamente locked (acquisito) o unlocked (rilasciato).
- Solo un thread alla volta puo' possedere il lock.
- Un thread può bloccarsi nel tentativo di acquisire un lock se e' già posseduto da un altro thread.
- Un thread bloccato nel tentativo di acquisire un lock verrà risvegliato quando il lock potrà essere di nuovo riacquisito.

> *Corrispondono ai lock della PTHREAD Library.*

Una condizione è una astrazione che può essere usata per attendere e segnalare eventi:

- Un thread può segnalare che una condizione è diventata vera.
- Un thread si può bloccare e aspettare che una condizione venga segnalata.
- Un lock è sempre necessario come guardia di una condizione.

Un lock è normalmente associato con importanti condizioni:

- L'attesa e la segnalazione di condizioni è possibile solo per i thread che possiedono i lock delle condizioni.
- Quando un thread aspetta che una condizione venga segnalata, rilascia il lock della condizione.
- Quando un thread segnala una condizione, deve esplicitamente rilasciare il lock della condizione.

Le condizioni forniscono un significato ad un thread per la sospensione dell'esecuzione (`wait`) finché non viene notificato da un altro thread che la condizione è diventata vera. Dato che l'accesso a questo stato condiviso avviene all'interno di thread differenti deve essere protetto, per questo motivo un lock di qualunque forma è associato ad una condizione. La proprietà chiave che l'attesa per una condizione fornisce è che il rilascio del lock associato e l'attesa del thread avvengono in modo atomico. Java docs (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Condition.html)

> *Lock rientrante: consente di acquisire più volte lo stesso lock \(\to\) è consentito acquisirlo di nuovo senza che si verifichi un deadlock. Un esempio di lock rientrante:*

```
public class ReentrantLockExample {
    private Lock lock = new ReentrantLock();
    private int lockCount = 0;

    public void performTask() {
        lock.lock(); // Acquisizione del lock
        ++lockCount;

        try {
            // Blocco critico - codice che richiede accesso sincronizzato
            System.out.println("Task in corso...");

            // Chiamata ricorsiva
            performSubTask();
        } finally {
            --lockCount;
            if (lockCount == 0) {
                lock.unlock();
                // Rilascio del lock solo se è stato acquisito una sola volta
            }
        }
    }

    public void performSubTask() {
        lock.lock();
        // Acquisizione del lock anche se è già stato acquisito nel chiamante

        ++lockCount;

        try {
            // Blocco critico della sotto-operazione
            System.out.println("Sotto-task in corso...");
        } finally {
            --lockCount;
            if (lockCount == 0) {
                lock.unlock();
                // Rilascio del lock solo se è stato acquisito una sola volta
            }
        }
    }

    public static void main(String[] args) {
        ReentrantLockExample example = new ReentrantLockExample();
        example.performTask();
    }
}
```

Atomic references

Un riferimento atomico incapsula un riferimento a un oggetto e lo gestisce in mutua esclusione.

Esempio:

```
UnaryOperator<Integer> operator = new UnaryOperator<Integer>() {
        @Override
        public Integer apply(Integer value) {
                return value + 1;
        }
};
```

Viene creato un oggetto `UnaryOperator<Integer>` utilizzando una classe anonima. `UnaryOperator` è una funzione che accetta un argomento dello stesso tipo e restituisce un risultato dello stesso tipo. L'oggetto `operator` è una funzione che, quando applicata a un numero intero, restituisce il numero successivo incrementato di 1. Questo è un esempio di programmazione funzionale in Java, in cui le funzioni sono trattate come oggetti e possono essere passate come argomenti o assegnate a variabili.

Torna all'indice

Pools di Risorse

I problemi di concorrenza sono spesso causati dalle risorse condivise.

- La corretta gestione degli accessi alle risorse condivise e' uno dei piu' classici problemi di concorrenza.

Un gruppo di risorse identiche è chiamato pool (piscina) di risorse.

- Quando una pool è creata/distrutta, tutte le risorse sono anche acquisite/rilasciate.
- Le risorse vengono assegnate per l'utilizzo su richiesta al pool.
- L'accesso controllato delle risorse è garantito dalla pool.

Le pools di risorse sono normalmente usate per controllare l'ammontare delle risorse usate da un sistema concorrente:

- Per assicurare che un numero sufficiente di risorse sia disponibile.
- Per assicurare che le risorse siano usate in modo efficiente.

Thread Pools

I thread sono virtualmente le risorse più rilevanti in un sistema concorrente. La costruzione, distruzione, e l'accesso ai thread sono spesso controllati usando i thread pools.

Quando viene creata, una thread pool crea e attiva tutti i thread nella pool. Questo per assicurarsi che i thread siano immediatamente disponibili quando si avrà la necessita di usarli e per assicurarsi che il livello della concorrenza sia controllato.

Torna all'indice

Executors

Un (semplice) executor è un'astrazione che può essere usato per gestire tasks concorrenti:

- E' associato con una thread pool usata per eseguire tasks.
- Accoda le tasks che non possono essere avviate immediatamente.
- Fornisce modi per restituire i risultati delle tasks (se presenti).
- Fornisce modi per restituire le eccezioni che hanno causato l'errore della cessazione delle tasks (se presenti).
- Permette di interrompere le tasks e di terminare tutti i thread nella pool.
- (A volte) Fornisce una serie di politiche di scheduling.

Un executor offre tre possibili opzioni per eseguire le tasks:

1. *One way execution*: per il quale l'esecutore non fornisce un mezzo per sapere se l'attività è stata effettivamente eseguita e per leggere il risultato dell'attività (se presente).
2. *Execution with callback*: per il quale l'esecutore consente a un'attività di callback di utilizzare il risultato dell'attività (se presente), spesso nel thread che ha eseguito l'attività richiesta.
3. *Execution with future*: per il quale l'esecutore fornisce un futuro (una promessa) che gestisce il risultato dell'attività richiesta (se presente) quando diventa disponibile.

> *Gli esecutori prevedono attività asincrone.*

Callbacks

Una callback task è una task che viene eseguita quando un executor conclude una task richiesta.

- La callback task riceve il risultato dell'attività richiesta (se presente).
- La callback task riceve l'eccezione che ha causato la terminazione dell'attività richiesta (se presente).
- (A volte) la callback task viene eseguita nello stesso thread che ha eseguito l'attività richiesta.

Le callback task vengono implementate come oggetti associati a una richiesta di esecuzione di una task.

Futures

Un future (o promessa) è un mezzo per gestire:

- Il risultato di un'attività asincrona (se presente).
- L'eccezione che ha causato la conclusione dell'attività asincrona (se presente).

I futures sono implementati come oggetti associati a una richiesta di esecuzione di un'attività:

- Sono associati dinamicamente al risultato dell'attività richiesta (se presente) quando diventa disponibile.
- Sono associati dinamicamente all'eccezione che ha causato la terminazione dell'attività richiesta (se presente) quando diventa disponibile.

Un future blocca il thread che tenta di accedere al suo valore incorporato (un risultato o un'eccezione) se il valore non è ancora disponibile:

- Il thread viene ripreso quando il risultato diventa disponibile (se presente) e il risultato viene restituito.
- Il thread viene ripreso quando diventa disponibile un'eccezione (se presente) e l'eccezione viene generata.

> *Nota bene: un future non si blocca se il suo valore è già disponibile al momento della richiesta.*

Torna all'indice

Indice

Java Reflection

Java e Java Virtual Machine (JVM) forniscono un mezzo, il pacchetto `java.lang.reflect`, per rinviare alcune decisioni in fase di esecuzione. Java rimane un linguaggio tipizzato staticamente, ma fornisce un mezzo puramente orientato agli oggetti per supportare:

- Collegamento dinamico delle classi
- Introspezione (dinamica) (degli oggetti)
- Creazione dinamica di oggetti
- Accesso dinamico ai campi
- Invocazione dinamica di metodi

Si noti che i seguenti fatti sono veri in Java:

- Ogni oggetto è associato alla classe che è stata utilizzata per crearlo, la cosiddetta classe factory.
- Ogni classe/interfaccia è rappresentata in fase di runtime da un oggetto, il cosiddetto oggetto classe (o descrittore di classe).

Data la classe/interfaccia `C`, la JVM fornisce un oggetto della classe `java.lang.reflect.Class<C>`. Data la classe/interfaccia `C`, l'oggetto che rappresenta la classe/interfaccia può essere referenziato da `C.class`.

Ogni oggetto di classe è associato a un caricatore di classe, che è l'oggetto utilizzato per caricare il bytecode della classe. Gli oggetti di classe sono il punto di ingresso della Java Reflection.

Esempio riflessione

La sintassi `Class<?>` in Java fa parte del sistema di riflessione ed è utilizzata per rappresentare l'oggetto `Class` di una classe sconosciuta o non specificata.

- `Class`: `Class` è una classe in Java che rappresenta metadati sulla classe di un oggetto, inclusi i dettagli sulla sua struttura, campi, metodi e altro ancora. È parte del sistema di riflessione di Java.
- `<T>`: È un parametro generico che indica un tipo. Quando si utilizza `<T>` dopo `Class`, si sta dicendo a Java di lavorare con un oggetto `Class` che rappresenta il tipo specifico di classe. Ad esempio, `Class<String>` rappresenta l'oggetto `Class` di una classe di tipo `String`.
- `<?>`: Questa è una wildcard (`?`) che indica "qualsiasi tipo". Quando combinata con `Class`, `Class<?>` significa un oggetto `Class` di una classe di tipo sconosciuto. Può essere qualsiasi classe.

In altre parole, `Class<?>` è un modo di indicare che stiamo lavorando con un oggetto `Class` di una classe, ma non stiamo specificando il tipo esatto di quella classe. Questo è utile in situazioni in cui il tipo della classe non è noto a compile time o è un parametro generico.

Ad esempio, se stai scrivendo una funzione che accetta un oggetto `Class` come argomento, ma non sai quale sarà la classe specifica, puoi dichiarare il parametro come `Class<?>`.

```
public void esempio(Class<?> classe) {
    // ...
}
```

Questa dichiarazione consente di passare qualsiasi tipo di classe all'interno della funzione. L'utilizzo di `Class<?>` è comune quando si lavora con il sistema di riflessione e si desidera scrivere codice che possa gestire classi di tipo sconosciuto in modo generico.

> *Guarda Esempio08 in Java.*

Torna all'indice

Class Objects

Di seguito sono riportati alcuni modi per ottenere un oggetto di classe:

- Dato un oggetto `o`, `o.getClass()` restituisce l'oggetto classe associato a `o`.
- Data una stringa `n` contenente il nome completo di una classe, `Class.forName(n)` restituisce l'oggetto classe corretto.
- Dato un caricatore di classi `l` e una stringa `n` contenente il nome completo di una classe, `l.loadClass(n)` restituisce l'oggetto classe corretto.

> *Nota che è possibile accedere agli oggetti della classe per nome anche se non sono disponibili oggetti della classe. Potrebbe essere lanciata un'eccezione `ClassNotFoundException`.*

Oltre alle funzionalità legate alla riflessione Java dinamica, un oggetto `c` della classe `Class<C>` può:

- Eseguire un cast di tipo dell'oggetto `o` sulla classe rappresentata, con `c.cast(o)`, che è equivalente a `(C)o`.
- Controlla se l'oggetto `o` è un'istanza della classe rappresentata, con `c.isInstance(o)`, che è equivalente a `o` istanza di `C`.
- Controlla se la classe/interfaccia di riferimento è la stessa o è una superclasse/superinterfaccia di una classe rappresentata da `k`, con `c.isAssignableFrom(k)`.

Un "class descriptor" è una rappresentazione testuale di una classe che viene utilizzata in vari contesti, come il caricamento delle classi, la riflessione e la firma delle classi.

> *La classe `java.lang.String` ha il class descriptor `Ljava/lang/String;`.*

Torna all'indice

Introspection

Dato un oggetto c della classe `Class<C>`, è possibile ispezionare la struttura di `C` e, ad esempio:

- È possibile elencare i descrittori di campo dei campi visibili di `C`.
- È possibile elencare i descrittori dei costruttori visibili di `C`.
- È possibile elencare i descrittori di metodo dei metodi visibili di `C`.
- È possibile ottenere un riferimento alla classe oggetto della classe base (o superclasse) di `C`.
- È possibile ottenere riferimenti agli oggetti classe delle interfacce implementate dal `C`.

La parola introspezione si riferisce alla possibilità di un oggetto di ispezionare la sua classe.

Torna all'indice

Creazione dinamica degli oggetti

Dato un oggetto `c` della classe `Class<C>`, e' possibile creare oggetti di classe `C`:

- Utilizzando `c.newInstance()`, anche se deprecato.
- Utilizzando `c` per accedere a uno dei costruttori descrittori di `c` e invocare il costruttore con qualche argomenti.

> *Nota che, se `C` e' conosciuto, allora la creazione dinamica dell'oggetto non necessita' di un cast esplicito.*

```
// Creazione stringa vuota
String s = String.class.newInstance();
```

Accesso dinamico ai campi

Dato un oggetto `c` della classe `Class<C>`, e' possibile accedere al descrittore del campo del campo visibile di `c`. Oltre a descrivere i campi, i descrittori di campo possono essere utilizzati per ottenere o impostare il valore del campo per alcuni oggetti.

Dato un oggetto `c` della classe `Class<C>`, un descrittore di campo `f` ottenuto da `c`, e un oggetto `o` di classe `C`, e' possibile:

1. Usare `f.get(o)` per leggere il valore corrente del campo per `o`.
2. Usare `f.set(o, v)` per impostare il valore corrente del campo su `v` per `o`.

> *La classe `java.lang.reflect.Field` non e' generica, e fino a prova contraria, non fornisce al compilatore il tipo del campo descritto.*

Invocazione dinamica dei metodi

Dato un oggetto `c` della classe `Class<C>`, e' possibile accedere al descrittore del metodo dei metodi visibili di `c`. Oltre a descrivere i metodi, i descrittori di metodo possono essere utilizzati per invocare il metodo descritto con argomenti adeguati.

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

> *La classe `java.lang.reflect.Method` non è generica e pertanto non fornisce al compilatore (ad esempio) il tipo restituito del metodo descritto.*

Torna all'indice

Dynamic Proxies

Dato un array `a` di oggetti di classe associati alle interfacce, un proxy dinamico è un oggetto che implementa le interfacce in `a` e richiama il codice utente dopo le invocazioni dei metodi.

> *E' un oggetto che funge da rappresentante di un altro oggetto.*

I proxies sono creati usando `java.lang.reflect.proxy.Proxy.newInstance`. Il codice utente richiamato durante l'invocazione del metodo è arbitrario ed è un'implementazione dell'interfaccia funzionale `java.lang.reflect.InvocationHandler`. Al codice utente viene fornito il descrittore del metodo richiamato. Il codice utente può restituire un valore, che viene utilizzato come valore di ritorno dell'invocazione che ha attivato l'attivazione del codice utente.

I proxy dinamici possono essere utilizzati per una varietà di scopi, tra cui:

- **Autenticazione e autorizzazione:** I proxy dinamici possono essere utilizzati per autenticare e autorizzare gli utenti prima di consentirgli di accedere a un oggetto reale.
- **Logging:** I proxy dinamici possono essere utilizzati per registrare le interazioni di metodo con un oggetto reale.
- **Profilatura:** I proxy dinamici possono essere utilizzati per profilare le prestazioni di un oggetto reale.
- **Decoupling:** I proxy dinamici possono essere utilizzati per decouple un client da un oggetto reale.

Ecco un esempio di come utilizzare un proxy dinamico per l'autenticazione:

```
public class AuthenticationProxy implements MyInterface {

    private MyObject realObject;
    private AuthenticationService authenticationService;

    public AuthenticationProxy(
                            MyObject realObject,
                            AuthenticationService authenticationService) {

        this.realObject = realObject;
        this.authenticationService = authenticationService;
    }

    @Override
    public void doSomething() {
        // Verifica l'autenticazione dell'utente
        if (!authenticationService.isAuthenticated()) {
            throw new AuthenticationException("Utente non autenticato");
        }

        // Invia la richiesta all'oggetto reale
        realObject.doSomething();
    }
}
```

In questo esempio, il proxy dinamico verifica l'autenticazione dell'utente prima di consentire all'utente di chiamare il metodo `doSomething()` sull'oggetto reale. Se l'utente non è autenticato, viene lanciata un'eccezione.

I proxy dinamici sono un potente strumento che può essere utilizzato per una varietà di scopi. Sono facili da usare e possono essere utilizzati per migliorare la sicurezza, la funzionalità e le prestazioni delle applicazioni.

Torna all'indice

Indice

Aspect-Oriented Programming (AOP)

L'AOP è un paradigma di programmazione che consente di aggiungere funzionalità aggiuntive, chiamate **aspetti**, al codice esistente. Gli aspetti possono essere utilizzati per aggiungere funzionalità come la registrazione, la profilatura, la sicurezza e l'autenticazione.

L'AOP è stato proposto come il passo successivo alla programmazione orientata agli oggetti (OOP) sin dai primi anni 2000. L'AOP è principalmente inteso per promuovere il riutilizzo del codice.

Nell'ambito dell'OOP (quasi) puro sostenuto da Java, l'AOP può essere drasticamente semplificato:

- Gli aspetti sono caratteristiche degli oggetti che non sono prontamente fornite dalle loro classi.
- Un fornitore di aspetti è un oggetto che può collegare/scollegare un aspetto da un oggetto o che può creare un oggetto con un aspetto richiesto.

Gli aspetti sono caratteristiche di oggetti che non sono prontamente fornite dalle loro classi. Ad esempio, un aspetto potrebbe aggiungere la funzionalità di registrazione a un oggetto che non fornisce tale funzionalità per impostazione predefinita.

Gli aspetti dovrebbero essere:

- **Indipendente dalle caratteristiche degli oggetti a cui sono collegati**. Ciò significa che gli aspetti dovrebbero essere in grado di funzionare con qualsiasi oggetto, indipendentemente dalle sue classi o interfacce.
- **Liberamente componibili**. Ciò significa che un oggetto può essere collegato a più aspetti, ottenendo così più funzionalità.
- **Ortogonali**. Ciò significa che la composizione di aspetti fornisce la somma di funzionalità indipendenti.

Il paradigma AOP può essere implementato in Java utilizzando proxy dinamici. I proxy dinamici sono oggetti che fungono da intermediari tra un client e un oggetto reale.

> *Dato un oggetto o, un fornitore di aspetti associa l'aspetto a o tramite un proxy dinamico che intercetta tutte le invocazioni ai metodi pubblici di o.*

Ecco un esempio di come utilizzare l'AOP per aggiungere la funzionalità di registrazione a un oggetto:

```
public class MyObject {
    public void doSomething() {
        // Fai qualcosa
    }
}

@Aspect
public class LoggingAspect {

    @Around("execution(* doSomething(..))")
    public void logMethodCall(ProceedingJoinPoint joinPoint) throws Throwable {
        // Registra la chiamata al metodo
        System.out.println("Chiamata al metodo doSomething");

        // Prosegui con la chiamata al metodo reale
        joinPoint.proceed();
    }
}
```

In questo esempio, l'aspetto `LoggingAspect` viene utilizzato per registrare la chiamata al metodo `doSomething()`. L'aspetto utilizza l'annotazione `@Around` per definire un advice che verrà eseguito prima e dopo la chiamata al metodo reale.

Quando viene chiamato il metodo `doSomething()`, l'aspetto `LoggingAspect` registra la chiamata al metodo. Quindi, l'aspetto procede con la chiamata al metodo reale.

Alcuni aspetti generali che vengono spesso considerati quando si implementa l'AOP:

- **Condivisione:** Gli aspetti di condivisione sono utilizzati per garantire che gli oggetti condivisi siano utilizzati in modo sicuro. Ad esempio, un aspetto di condivisione potrebbe implementare un mutex per garantire che un oggetto condiviso non venga modificato da più thread contemporaneamente.
- **Registrazione:** Gli aspetti di registrazione vengono utilizzati per tracciare le interazioni con gli oggetti. Ad esempio, un aspetto di registrazione potrebbe registrare le chiamate ai metodi degli oggetti in un file di log.
- **Persistenza:** Gli aspetti di persistenza vengono utilizzati per salvare gli oggetti in un database o in un altro sistema di archiviazione. Ad esempio, un aspetto di persistenza potrebbe implementare un metodo per salvare un oggetto quando viene creato o modificato.
- **Attivazione:** Gli aspetti di attivazione vengono utilizzati per attivare o disattivare le funzionalità di un oggetto. Ad esempio, un aspetto di attivazione potrebbe implementare un metodo per disabilitare la funzionalità di logging quando un'applicazione è in modalità di test.
- **Remotismo:** Gli aspetti di remoto vengono utilizzati per consentire agli oggetti di interagire tra loro in remoto. Ad esempio, un aspetto di remoto potrebbe implementare un metodo per inviare una richiesta a un oggetto remoto.
- **Ricaricabilità:** Gli aspetti di ricaricabilità vengono utilizzati per aggiornare gli oggetti senza interrompere l'esecuzione dell'applicazione. Ad esempio, un aspetto di ricaricabilità potrebbe implementare un metodo per ricaricare un oggetto da un file di configurazione.

In genere, solo i metodi dalle interfacce implementate vengono considerati quando si implementano aspetti. Ciò è dovuto al fatto che gli aspetti vengono applicati agli oggetti in base alle loro interfacce, non alle loro classi.

Torna all'indice

Shared Aspect

Un oggetto condiviso è un oggetto che deve garantire la mutua esclusione per l'esecuzione delle sue modalità. Solo i metodi delle interfacce implementate sono interessanti perché sono i metodi esportati. Un proxy dinamico è sufficiente per intercettare tutte le invocazioni a metodi interessanti:

- Il gestore di invocazione fornisce oggetti utilizzati come blocco di sincronizzazione.
- Il gestore di invocazione entra in una regione critica protetta dal lock prima di invocare il metodo di destinazione, ed esce dalla regione critica immediatamente dopo.
- Il gestore di invocazione esce dalla regione critica anche in caso di eccezioni.

Esempio: Un aspetto di condivisione potrebbe essere utilizzato per garantire che un oggetto condiviso non venga modificato da più thread contemporaneamente. Ad esempio, il seguente aspetto implementa un mutex per garantire che un oggetto `MyObject` possa essere utilizzato solo da un thread alla volta:

```
@Aspect
public class SharingAspect {

    @Around("execution(* MyObject.*(..))")
    public void synchronize(ProceedingJoinPoint joinPoint) throws Throwable {
        // Ottieni un mutex per l'oggetto
        Object mutex = joinPoint.getThis();

        // Blocca il mutex
        synchronized (mutex) {
            // Esegui il metodo reale
            joinPoint.proceed();
        }
    }
}
```

Torna all'indice

Logging Aspect

Un oggetto di registrazione è un oggetto che traccia le invocazioni ai suoi metodi in un registro dei messaggi. Solo i metodi delle interfacce implementate sono interessanti perché sono i metodi esportati. Un proxy dinamico è sufficiente per intercettare tutte le invocazioni a metodi interessanti:

- Il gestore delle chiamate registra prima e dopo aver invocato il metodo di destinazione.
- Il gestore delle chiamate registra anche in caso di eccezioni.

Esempio: Un aspetto di registrazione potrebbe essere utilizzato per tracciare le chiamate ai metodi di un oggetto. Ad esempio, il seguente aspetto registra le chiamate ai metodi di un oggetto `MyObject` in un file di log:

```
@Aspect
public class LoggingAspect {

    @Around("execution(* MyObject.*(..))")
    public void logMethodCall(ProceedingJoinPoint joinPoint) throws Throwable {
        // Registra la chiamata al metodo
        System.out.println("Chiamata al metodo " + joinPoint.getSignature().getName());

        // Prosegui con la chiamata al metodo reale
        joinPoint.proceed();
    }
}
```

Torna all'indice

Persistent Aspect

Un oggetto persistente è un oggetto che sopravvive allo spegnimento del sistema in cui è stato creato o modificato. I proxy dinamici non sono necessari perché all'utente viene richiesto esplicitamente:

- Effettua il commit delle modifiche nell'archivio persistente.
- Modifiche di rollback (ricaricando i dati dall'archivio persistente).

È possibile ottenere una semplice persistenza caricando/salvando oggetti serializzabili su file:

- Gli oggetti che implementano `java.io.Serializable` possono essere facilmente serializzati con `java.io.ObjectOutputStream` e deserializzati con `java.io.ObjectInputStream`.
- Gli oggetti serializzabili forniscono un campo `serialVersionUID` privato per disambiguare diverse versioni delle loro classi.

  Esempio: Un aspetto di persistenza potrebbe essere utilizzato per salvare gli oggetti in un database. Ad esempio, il seguente aspetto salva un oggetto `MyObject` in un database quando viene creato o modificato:

```
@Aspect
public class PersistenceAspect {

    @AfterReturning("execution(* MyObject.*(..))")
    public void saveObject(Object object) {
        // Salva l'oggetto nel database
        // ...
    }
}
```

Torna all'indice

Active Aspect

Un oggetto attivo è un oggetto che esegue i propri metodi in un pool di thread dedicato. Ciò significa che i metodi di un oggetto attivo non vengono eseguiti nel thread che li ha invocati, ma in un thread diverso, che fa parte del pool di thread dedicato all'oggetto. Solo i metodi che implementano interfacce sono interessati perché questi sono metodi esportati. Un proxy dinamico è sufficiente per intercettare tutte le invocazioni dei metodi interessati. Un'interfaccia attiva è un'interfaccia che fornisce metodi con firme simili ai metodi nelle interfacce implementate dall'oggetto attivo. La differenza principale è che i metodi di un'interfaccia attiva restituiscono i risultati utilizzando futures e callback.

Esempio: Consideriamo l'esempio di un oggetto attivo che rappresenta un server web. L'oggetto attivo implementa l'interfaccia `WebServer`, che definisce i metodi per gestire le richieste HTTP.

La seguente è una possibile implementazione dell'interfaccia `WebServer`:

```
public interface WebServer {

    void handleRequest(HttpRequest request, HttpResponse response);
    void shutdown();

}
```

L'interfaccia `WebServer` non fornisce alcuna informazione sul modo in cui i metodi `handleRequest()` e `shutdown()` vengono eseguiti. Per specificare che questi metodi devono essere eseguiti in un pool di thread dedicato, possiamo creare un'interfaccia attiva che estende `WebServer`. La seguente è una possibile implementazione dell'interfaccia attiva `ActiveWebServer`:

```
public interface ActiveWebServer extends Active<WebServer> {

    Future<Void> handleRequest(HttpRequest request, HttpResponse response);
    void shutdown();

}
```

L'interfaccia `ActiveWebServer` fornisce due metodi aggiuntivi:

- `Future<Void> handleRequest()`: questo metodo restituisce un oggetto `Future` che rappresenta il risultato dell'invocazione del metodo `handleRequest()`.
- `void shutdown()`: questo metodo non restituisce alcun valore.

Per creare un oggetto attivo che implementi l'interfaccia `ActiveWebServer`, possiamo utilizzare un proxy dinamico. Un proxy dinamico è un oggetto che viene creato dinamicamente per rappresentare un altro oggetto. In questo caso, il proxy dinamico rappresenterà l'oggetto che implementa l'interfaccia `WebServer`.

La seguente è una possibile implementazione di un proxy dinamico per l'interfaccia `ActiveWebServer`:

```
public class ActiveWebServerProxy implements ActiveWebServer {

    private WebServer server;

    public ActiveWebServerProxy(WebServer server) {
        this.server = server;
    }

    @Override
    public Future<Void> handleRequest(
                    HttpRequest request, HttpResponse response) {
        // Crea un oggetto Future per rappresentare il risultato dell'invocazione del metodo
        // handleRequest()
        Future<Void> future = new Future<>();

        // Esegue il metodo handleRequest() nell'oggetto server
        new Thread(() -> {
            try {
                server.handleRequest(request, response);
                future.setDone();
            } catch (Exception e) {
                future.setException(e);
            }
        }).start();

        return future;
    }

    @Override
    public void shutdown() {
        server.shutdown();
    }

}
```

Il proxy dinamico implementa tutti i metodi dell'interfaccia `ActiveWebServer`. Il metodo `handleRequest()` esegue il metodo `handleRequest()` dell'oggetto server in un thread separato. Il metodo `shutdown()` esegue il metodo `shutdown()` dell'oggetto server.

Per utilizzare l'oggetto attivo, possiamo creare un'istanza del proxy dinamico. Ad esempio:

```
WebServer server = new WebServer();
ActiveWebServer activeServer = new ActiveWebServerProxy(server);
```

Quindi, possiamo invocare i metodi dell'oggetto attivo come se si trattasse di un normale oggetto. Ad esempio:

```
activeServer.handleRequest(new HttpRequest(), new HttpResponse());
activeServer.shutdown();
```

In questo modo, possiamo garantire che i metodi dell'oggetto attivo vengano eseguiti in un pool di thread dedicato.

Torna all'indice

Remote Aspect

Un oggetto remoto è un oggetto che può essere invocato da un altro oggetto che si trova in un altro processo o computer.

Per implementare un oggetto remoto, è necessario fornire un modo per i client di inviare richieste all'oggetto remoto e per l'oggetto remoto di inviare risposte ai client.

Una soluzione comune è utilizzare una comunicazione socket. In questo caso, l'oggetto remoto crea una socket per accettare richieste dai client. Quando un client invia una richiesta, l'oggetto remoto la riceve sulla socket e la elabora. Dopo aver elaborato la richiesta, l'oggetto remoto invia una risposta al client sulla socket.

L'oggetto remoto crea una socket per accettare richieste dai client. Quando un client invia una richiesta, l'oggetto remoto la riceve sulla socket e la elabora. Dopo aver elaborato la richiesta, l'oggetto remoto invia una risposta al client sulla socket.

Un oggetto remoto basato su eventi è un oggetto remoto che invia eventi ai client.

Un evento è un messaggio che viene inviato da un oggetto a un altro oggetto. Gli eventi possono essere utilizzati per notificare ai client che si è verificato un evento di interesse.

Per implementare un oggetto remoto basato su eventi, è necessario fornire un modo per i client di registrarsi per eventi e per l'oggetto remoto di inviare eventi ai client registrati.

Una soluzione comune è utilizzare un canale di eventi. In questo caso, l'oggetto remoto crea un canale di eventi. I client possono registrarsi per eventi sull'oggetto remoto passando un callback al canale di eventi. Quando l'oggetto remoto genera un evento, lo invia al canale di eventi. Il canale di eventi quindi invia l'evento ai client registrati.

Un oggetto remoto asincrono è un oggetto remoto che può essere invocato in modo asincrono.

Invocazione asincrona significa che l'invocazione del metodo non blocca il thread che ha invocato il metodo. Il metodo viene eseguito in un thread separato e il thread che ha invocato il metodo riceve una notifica quando il metodo è terminato.

Per implementare un oggetto remoto asincrono, è necessario utilizzare un meccanismo di chiamata asincrona. Una soluzione comune è utilizzare una coda di lavoro. In questo caso, l'oggetto remoto crea una coda di lavoro. I client possono inviare richieste all'oggetto remoto inviando le richieste alla coda di lavoro. La coda di lavoro quindi esegue le richieste in un thread separato.

L'oggetto remoto crea una coda di lavoro. I client possono inviare richieste all'oggetto remoto inviando le richieste alla coda di lavoro. La coda di lavoro quindi esegue le richieste in un thread separato.

Esempio: Consideriamo l'esempio di un oggetto remoto che rappresenta un servizio di calcolo. Il servizio di calcolo può essere utilizzato per eseguire calcoli complessi.

L'oggetto remoto implementa l'interfaccia `IComputationService`. L'interfaccia `IComputationService` definisce un metodo `compute()` che esegue un calcolo complesso.

La seguente è una possibile implementazione dell'interfaccia `IComputationService`:

```
public interface IComputationService {
        double compute(double x, double y);
}
```

La seguente è una possibile implementazione del servizio di calcolo in modo sincrono:

```
public class ComputationService implements IComputationService {

    @Override
    public double compute(double x, double y) {
        // Esegue il calcolo complesso
        double result = Math.pow(x, 2) + Math.pow(y, 2);

        // Restituisce il risultato
        return result;
    }
}
```

In questa implementazione, il metodo `compute()` esegue il calcolo complesso nel thread che ha invocato il metodo. Ciò significa che il thread che ha invocato il metodo potrebbe essere bloccato per un periodo di tempo significativo, a seconda della complessità del calcolo.

Per implementare il servizio di calcolo in modo asincrono, possiamo utilizzare una coda di lavoro. La seguente è una possibile implementazione del servizio di calcolo in modo asincrono:

```
public class ComputationService implements IComputationService {

    private final ExecutorService executorService = Executors.newSingleThreadExecutor();

    @Override
    public Future<Double> compute(double x, double y) {
        // Crea un oggetto Future per rappresentare il risultato del calcolo
        Future<Double> future = new Future<>();

        // Esegue il calcolo complesso in un thread separato
        executorService.submit(() -> {
            double result = Math.pow(x, 2) + Math.pow(y, 2);
            future.set(result);
        });

        // Restituisce l'oggetto Future
        return future;
    }
}
```

In questa implementazione, il metodo `compute()` crea un oggetto `Future` per rappresentare il risultato del calcolo. L'oggetto `Future` viene utilizzato per notificare il thread che ha invocato il metodo quando il calcolo è terminato.

Il metodo `compute()` esegue il calcolo complesso in un thread separato, utilizzando l'executor service. Ciò significa che il thread che ha invocato il metodo non viene bloccato.

Client

Il client può utilizzare l'oggetto remoto in modo sincrono o asincrono.

Per utilizzare l'oggetto remoto in modo sincrono, il client può semplicemente invocare il metodo desiderato. Ad esempio:

```
IComputationService service = new ComputationService();

double result = service.compute(1.0, 2.0);
```

In questo caso, il thread del client verrà bloccato fino a quando il calcolo non sarà terminato.

Per utilizzare l'oggetto remoto in modo asincrono, il client può creare un oggetto `Future` per rappresentare il risultato del calcolo. Ad esempio:

```
IComputationService service = new ComputationService();

Future<Double> future = service.compute(1.0, 2.0);

// Fai qualcos'altro...

double result = future.get();
```

In questo caso, il thread del client non verrà bloccato. Il thread del client potrà continuare a fare qualcos'altro mentre il calcolo viene eseguito in background. Quando il calcolo è terminato, il metodo `get()` restituirà il risultato del calcolo.

Torna all'indice

Reloadable Aspect

L'aspetto ricaricabile, o Reloadable Aspect, fa riferimento alla capacità di ricaricare classi in un ambiente Java.

1. **Classi e Class Loaders:**
    - Le classi vengono caricate in memoria tramite i class loader, che creano i descrittori di classe e rendono le classi disponibili per la JVM.
    - Nel contesto di Java, i class loader hanno alcune limitazioni:
        - Non possono scaricare classi (unloading).
        - Classi caricate da class loader diversi sono considerate diverse, anche se hanno lo stesso nome completamente qualificato.
2. **Ricaricabilità:**
    - Per consentire la ricarica dinamica delle classi (hot swapping), è necessario utilizzare un nuovo class loader per ogni ricarica.

- Inoltre, è necessario gestire le dipendenze delle classi in modo coerente durante la ricarica.

3. **Oggetti Ricaricabili:**
   - Anche se le singole istanze di oggetti non possono essere ricaricate, le istanze di una classe ricaricabile sono oggetti ricaricabili.
   - La factory class (classe che crea un'istanza di un oggetto) di un oggetto non può essere cambiata.

Esempio: Supponiamo di avere una classe `ReloadableExample` che vogliamo rendere ricaricabile:

```
public class ReloadableExample {
    private int counter = 0;

    public void increment() {
        counter++;
        System.out.println("Counter: " + counter);
    }
}
```

1. **Implementazione Reloadable Aspect:**
   - Creiamo un nuovo class loader quando desideriamo ricaricare la classe `ReloadableExample`.
   - Gestiamo attentamente le dipendenze per evitare problemi durante la ricarica.

2. **Ricarica della Classe:**
   - Quando vogliamo aggiornare la logica della classe `ReloadableExample`, creiamo un nuovo class loader.
   - Carichiamo la nuova versione della classe e istanziamo un nuovo oggetto `ReloadableExample` con la nuova logica.
   - I vecchi oggetti rimangono invariati finché non vengono sostituiti.

```
ClassLoader newClassLoader = new URLClassLoader(new URL[] {
                                        new URL("file:/path/to/classes/")
                                });

Class<?> reloadedClass = newClassLoader.loadClass("ReloadableExample");

ReloadableExample reloadedInstance =
                                                (ReloadableExample) reloadedClass.newInstance();

reloadedInstance.increment();  // Utilizza la nuova logica della classe
```

Questo approccio richiede una gestione attenta delle dipendenze, e non tutte le classi possono essere facilmente rendibili ricaricabili. Inoltre, alcune librerie potrebbero avere difficoltà a gestire classi ricaricabili. La ricarica dinamica delle classi è spesso utilizzata in ambienti di sviluppo o server in esecuzione continua per consentire modifiche del codice senza dover riavviare l'applicazione.

Torna all'indice

Indice

Test-Driven Development (TDD)

1. **Costi nello sviluppo del software:**
   - I costi del software spesso dominano i costi complessivi di un sistema, superando i costi hardware.
   - I costi di manutenzione, distribuiti nel corso degli anni di utilizzo del sistema, costituiscono una parte significativa delle spese legate al software.
2. **Costi di manutenzione:**
   - I costi di manutenzione superano i costi di sviluppo nei progetti software.
   - Il riutilizzo strutturato di framework e librerie software ben testati e documentati mira a ridurre i costi di manutenzione.
3. **Introduzione del Test-Driven Development (TDD):**
   - Il TDD viene introdotto per sottolineare l'importanza di un testing adeguato per tutte le parti di un sistema software.
   - Enfatizza che i componenti software non possono essere efficacemente riutilizzati se non sono stati adeguatamente testati.

Differenza tra Framework e Libreria

1. **Framework:**

- Un framework è una struttura pre-costruita e riutilizzabile che fornisce una base per la creazione di applicazioni software.
- Offre un insieme di regole, linee guida e astrazioni che gli sviluppatori seguono nella creazione di un'applicazione.
- L'inversione del controllo è una caratteristica comune dei framework, in cui il framework detta il flusso di controllo dell'applicazione.
- Gli sviluppatori estendono e personalizzano il framework fornendo implementazioni specifiche per funzioni o metodi predefiniti.

2. **Libreria:**
   - Una libreria è una raccolta di codice pre-scritto e routine che gli sviluppatori possono utilizzare nelle loro applicazioni.
   - Fornisce funzioni o metodi specifici che possono essere chiamati dall'applicazione per svolgere determinati compiti.
   - Il controllo del flusso dell'applicazione rimane nelle mani dello sviluppatore; la libreria viene invocata quando necessario.
   - Le librerie sono tipicamente meno prescrittive rispetto ai framework, consentendo agli sviluppatori una maggiore flessibilità nella progettazione dell'applicazione.

Esempio:

- Supponiamo di costruire un'applicazione web. Un framework per applicazioni web (ad esempio, Django, Ruby on Rails) fornisce un modo strutturato per costruire l'intera applicazione, inclusa la gestione delle richieste, la gestione dei database e la definizione delle route.
- D'altra parte, una libreria come jQuery fornisce funzioni specifiche per compiti come la manipolazione del DOM in JavaScript. Gli sviluppatori possono utilizzare queste funzioni in modo selettivo, senza una struttura globale prescritta per l'intera applicazione.

Torna all'indice

Costi dei Sistemi Software

1. **Divisione dei costi:**
   - I costi di un sistema software sono tipicamente suddivisi in due categorie principali:
     - **Costi diretti:** Direttamente associati alle attività svolte per il sistema.
       - Esempi: Costi degli sviluppatori, costi degli strumenti di sviluppo, costi di librerie e framework esterni, ecc.
     - **Costi Indiretti:** Associati a tutte le attività necessarie per supportare le attività svolte per il sistema.
       - Esempi: Costi di consulenti amministrativi, costi di consulenti legali, ecc.
2. **Rapporto tra costi diretti e indiretti:**
   - Di solito, i costi indiretti rappresentano una percentuale significativa rispetto ai costi diretti.
   - La percentuale di costi indiretti rispetto ai costi diretti può variare, ma generalmente oscilla tra il 50% (per le piccole e medie imprese) e il 100% (per le grandi imprese).

Esempio:

- Supponiamo di sviluppare un'applicazione software per un'azienda. I costi diretti includeranno i salari degli sviluppatori, l'acquisto di strumenti di sviluppo e l'utilizzo di framework esterni. D'altra parte, i costi indiretti potrebbero includere le spese per consulenti amministrativi o legali che supportano le attività di sviluppo. La consapevolezza di questi costi aiuta a gestire il bilancio complessivo del progetto software.

Torna all'indice

Evoluzione e Manutenzione del Software

1. **Necessità dell'evoluzione del software:**
   - I sistemi software devono evolvere perché:
     - I requisiti iniziali non sono stati catturati correttamente.
     - I requisiti cambiano durante il ciclo di vita del sistema.
2. **Inevitabilità dell'evoluzione:**
   - L'evoluzione di un sistema software è inevitabile, anche se i requisiti iniziali sono stati catturati correttamente e lo sviluppo iniziale ha prodotto un buon prodotto.
3. **Attività di manutenzione:**
   - Le attività di manutenzione sono svolte per far evolvere un sistema software contemporaneamente al suo utilizzo. Queste attività comprendono:
     - **Manutenzione Correttiva (Corrective Maintenance):** Risolvere anomalie o difetti emersi durante l'uso.
     - **Manutenzione Perfezionante (Perfective Maintenance):** Migliorare le qualità rilevanti del sistema, anche se non ci sono difetti evidenti.
     - **Manutenzione Adattativa (Adaptive Maintenance):** Adattare il sistema ai cambiamenti dell'ambiente circostante.

> ***Esempio:*** *Supponiamo che un'applicazione software per la gestione di un magazzino abbia inizialmente previsto solo la gestione di prodotti fisici, ma in seguito i requisiti cambiano, e ora è necessario gestire anche prodotti digitali. In questo caso, l'evoluzione del sistema richiederà attività di manutenzione adattativa per adattare il sistema alle nuove esigenze.*

4. **Costi della Manutenzione:**
   - I costi di manutenzione sono spesso più del 50% dei costi totali associati a un sistema software nell'intero ciclo di vita.
     - Spesso si avvicinano al 75%.
     - Sono rilevanti anche a causa del lungo periodo di tempo in cui il sistema è operativo.
5. **Ripartizione dei Costi di Manutenzione:**
   - I costi di manutenzione sono normalmente suddivisi nei seguenti modi:
     - **Costi di manutenzione correttiva:** Circa il 20%.

    - **Costi di manutenzione perfettiva:** Circa il 60%.
    - **Costi di manutenzione adattativa:** Circa il 20%.
6. **Riduzione dei Costi di manutenzione perfettiva:**
    - È quindi necessario dedicare molto sforzo alla riduzione dei costi di manutenzione perfezionante assicurando un'analisi accurata e test del software prima del suo rilascio.

> ***Esempio:*** *Supponiamo che un software di contabilità aziendale venga costantemente aggiornato per soddisfare nuove regolamentazioni fiscali. Questo processo di adattamento richiederà costi di manutenzione adattativa. Tuttavia, una fase di analisi e test accurati iniziali potrebbe ridurre i costi associati alle modifiche successive, migliorando così l'efficienza del processo di manutenzione.*

7. **Studi sui costi di manutenzione:**
    - Studi ben noti sui costi di manutenzione nei sistemi implementati forniscono evidenze empiriche che la maggior parte delle anomalie potrebbe essere individuata attraverso revisioni sistematiche degli artefatti del progetto.
        - In particolare, il testing strutturato e accurato è un buon modo per garantire che le anomalie non si propaghino ai sistemi implementati.
8. **Dati interessanti sulla necessità di testing strutturato:**
    - Alcuni dati interessanti per quantificare la necessità di testing strutturato includono:
        - 1 anomalia su 10 trovate durante il testing si propaga al sistema implementato.
        - Il costo di risolvere un'anomalia aumenta di un fattore 10 se il sistema è già implementato.
        - Qualsiasi ritardo nel trovare un'anomalia influenza gravemente il costo di risolvere l'anomalia.

> ***Esempio:*** *Se durante il testing di un'applicazione si scoprono 10 anomalie, è probabile che solo una di esse si verifichi nel sistema implementato. Risolvere un problema dopo che il sistema è già in produzione è significativamente più costoso. Inoltre, qualsiasi ritardo nel rilevare un problema durante il testing potrebbe comportare costi aggiuntivi significativi nel correggerlo successivamente. Pertanto, l'approccio di test accurato può aiutare a ridurre i costi associati alle anomalie post-implementazione.*

> *Gli "artefatti del progetto" si riferiscono a tutti i documenti, codici sorgenti, diagrammi, specifiche e altri oggetti prodotti durante il ciclo di vita di sviluppo del software. Questi elementi costituiscono il materiale di lavoro e la documentazione generata durante le fasi di progettazione, sviluppo, testing e manutenzione di un progetto software.*

Torna all'indice

Testing

Alcune considerazioni fondamentali sul testing del software:

1. **Sufficiente copertura dei casi di test:**
    - Se il comportamento di una parte di un sistema viene testato in un numero sufficientemente elevato di casi, è possibile considerare accettabile il comportamento di tale parte anche nei (auspicabilmente pochi) casi rimanenti.
    - L'obiettivo è coprire una vasta gamma di scenari possibili per garantire che il software funzioni correttamente in varie condizioni.
2. **Limitazioni del testing:**
    - Sebbene il testing sia un mezzo efficace per individuare anomalie, non può dimostrare che una parte di un sistema è corretta al 100%.
    - Il testing può fornire un elevato grado di fiducia, ma non è esaustivo nel dimostrare la correttezza assoluta di un sistema.
3. **Divisione del testing:**
    - **Testing in the small (o Unit Testing):** Si concentra sul test delle singole parti (unità) del software. L'obiettivo è assicurarsi che ciascuna unità funzioni correttamente.
    - **Testing in the large:** Coinvolge il test dell'intero sistema. L'obiettivo è garantire che tutte le parti del sistema funzionino correttamente quando sono integrate.

Unit testing

L'unit testing è un processo di test di unità di codice in isolamento. L'obiettivo dell'unit testing è garantire che ogni unità di codice funzioni come previsto.

L'unit testing viene solitamente eseguito dagli sviluppatori software. Gli sviluppatori possono utilizzare diversi strumenti e tecniche per eseguire l'unit testing.

Module testing

Il module testing è un processo di test di moduli di codice. Un modulo di codice è un gruppo di unità di codice che lavorano insieme per eseguire una funzione specifica.

L'obiettivo del module testing è garantire che ogni modulo di codice funzioni come previsto.

Il module testing viene solitamente eseguito dagli sviluppatori software. Gli sviluppatori possono utilizzare diversi strumenti e tecniche per eseguire il module testing.

Integration testing

L'integration testing è un processo di test dell'interazione tra moduli di codice. L'obiettivo dell'integration testing è garantire che i moduli di codice interagiscano tra loro come previsto.

L'integration testing viene solitamente eseguito dai testatori professionisti. I testatori possono utilizzare diversi strumenti e tecniche per eseguire l'integration testing.

System testing

Il system testing è un processo di test dell'intero sistema software. L'obiettivo del system testing è garantire che il sistema software funzioni come previsto in tutte le sue funzionalità.

Il system testing viene solitamente eseguito dai testatori professionisti. I testatori possono utilizzare diversi strumenti e tecniche per eseguire il system testing.

Acceptance testing

L'acceptance testing è un processo di test del sistema software da parte degli utenti finali. L'obiettivo dell'acceptance testing è garantire che il sistema software soddisfi i requisiti degli utenti finali.

L'acceptance testing viene solitamente eseguito dagli utenti finali. Gli utenti finali possono utilizzare diversi strumenti e tecniche per eseguire l'acceptance testing.

> *I tipi di test possono essere eseguiti da diverse persone. L'unit testing e il module testing sono spesso eseguiti dagli sviluppatori software (component testing). L'integration testing, il system testing e l'acceptance testing sono spesso eseguiti da testatori professionisti (user testing).*

Torna all'indice

Testing in the small & in the slrge

1. **Testing in the large:**
   - Tratta il sistema come una scatola nera.
   - Il testing è mirato a verificare che il comportamento del sistema sia conforme alle aspettative.
   - Il set di casi di test è selezionato utilizzando gli artefatti prodotti per specificare i requisiti del sistema.
   - Questo approccio si concentra sulla funzionalità del sistema nel suo complesso, senza esaminare internamente il codice sorgente.
2. **Testing in the small:**
   - Tratta il sistema come una scatola bianca.
   - Esamina una parte sufficientemente piccola del sistema ispezionando il codice consegnato.
   - Il set di casi di test è selezionato utilizzando il codice considerato.
   - Tipicamente, si suddivide in:
     - **Testing delle istruzioni (Statement Testing):** Verifica che ogni istruzione nel codice venga eseguita almeno una volta.
     - **Testing dei rami (Branch Testing):** Garantisce che ogni ramo nel flusso di controllo del programma venga attraversato.
     - **Testing di branch e condizione (Branch and Condition Testing):** Assicura che ogni condizione e ogni ramo vengano valutati in tutte le possibili combinazioni.

Torna all'indice

Statement testing

Il concetto di **Statement Testing**, anche chiamato **Coverage Testing**, è basato sull'idea che nessuna parte del codice può essere considerata testata se non è stata eseguita. In altre parole, per rilevare eventuali anomalie, è necessario eseguire almeno una volta le parti del codice che le producono.

Principali punti chiave:

1. **Coverage testing:**
   - Il termine "coverage" si riferisce al grado di copertura del codice sorgente da parte dei test.
   - L'obiettivo è eseguire abbastanza test da garantire che tutte le istruzioni del codice siano state eseguite almeno una volta.
2. **Esecuzione del codice:**
   - L'esecuzione del codice è fondamentale per identificare e correggere anomalie.
   - Solo le parti del codice che sono effettivamente eseguite possono rivelare anomalie durante il testing.
3. **Set di test:**
   - Un insieme di casi di test T può essere utilizzato per eseguire il **Statement Testing** di un codice C.
   - Dopo l'esecuzione di tutti i casi di test in T, ogni istruzione in C dovrebbe essere stata eseguita almeno una volta.

> *Il **Statement Testing** si concentra sull'esecuzione di ogni singola istruzione nel codice sorgente per garantire che tutte siano state testate. Ciò contribuisce a identificare eventuali parti di codice che potrebbero non essere state eseguite durante il testing, indicando la necessità di ulteriori casi di test per migliorare la copertura.*

Torna all'indice

Branch testing

Il concetto di **Branch Testing**, spesso chiamato **Path Coverage Testing**, si concentra sull'analisi di tutti i percorsi di esecuzione del codice, puntando a coprire tutti i rami decisionali.

Principali punti chiave:

1. **Path coverage testing:**
   - Il termine "path coverage" si riferisce al fatto che l'obiettivo è coprire tutti i percorsi di esecuzione possibili nel codice.
   - Si tratta di un'approfondita analisi dei percorsi di esecuzione all'interno del codice.
2. **Rami decisionali:**
   - I rami decisionali si verificano quando il flusso del programma deve prendere una decisione, ad esempio in una struttura di controllo come un'istruzione `if` o un ciclo `for`.
   - L'obiettivo del Branch Testing è garantire che tutti i rami decisionali siano stati attraversati almeno una volta.
3. **Set di test:**
   - Un insieme di casi di test T può essere utilizzato per eseguire il **Branch Testing** di un codice C.
   - Dopo l'esecuzione di tutti i casi di test in T, tutti i percorsi di esecuzione in C dovrebbero essere stati attraversati almeno una volta.
4. **Anomalie complesse:**
   - Poiché alcune anomalie possono derivare da percorsi di esecuzione intricati, il Branch Testing è utile per identificare tali situazioni.

> *Il **Branch Testing** è progettato per coprire tutti i rami decisionali del codice, contribuendo a rilevare anomalie associate a percorsi di esecuzione complessi e garantendo una copertura più completa del flusso del programma.*

Torna all'indice

Branch and Condition Testing

Il **Branch and Condition Testing**, spesso chiamato **Condition Coverage Testing**, è mirato all'analisi delle condizioni che generano diversi percorsi di esecuzione nel codice considerato.

Punti chiave:

1. **Condition Coverage Testing:**
   - Questo tipo di testing si concentra sull'analisi delle condizioni all'interno del codice.
   - Il termine "condition coverage" indica che l'obiettivo è coprire tutte le condizioni nel codice durante l'esecuzione dei test.
2. **Percorsi di Esecuzione Intricati:**
   - Gli errori possono spesso derivare da percorsi di esecuzione complessi causati da diverse condizioni interrelate.
   - Il Branch and condition testing è progettato per affrontare questa complessità, garantendo che tutte le condizioni siano state considerate.
3. **Set di Test:**
   - Un insieme di casi di test T può essere utilizzato per eseguire il Branch and Condition Testing di un codice C.
   - Dopo l'esecuzione di tutti i casi di test in T, tutte le condizioni in C dovrebbero essere state considerate, tenendo conto di tutte le cause per i loro valori di verità.
4. **Anomalie Derivate da Condizioni Interrelate:**
   - Le anomalie possono essere causate da condizioni che interagiscono tra loro in modi intricati. Questo tipo di testing è progettato per identificare tali situazioni.

> *Il **Branch and Condition Testing** mira ad affrontare la complessità delle condizioni interrelate nel codice, contribuendo a una copertura più completa delle possibilità di esecuzione e a una maggiore robustezza del software.*

Torna all'indice

JUnit

JUnit e' uno strumento comune utilizzato per supportare il testing strutturato e accurato del codice Java.

1. **JUnit Overview:**
   - **JUnit** è uno strumento comune nel contesto del **testing di codice Java**.
   - È progettato per fornire un supporto efficace per la scrittura e l'esecuzione di test in ambiente Java.
2. **Lavoro su Test e Test Suites:**
   - **JUnit** lavora con due concetti principali: **Test (casi di test)** e **Test Suites**.
   - I **Test** rappresentano unità individuali di test, mentre le **Test Suites** sono insiemi di test.
3. **Supporto in Eclipse:**
   - **Eclipse** fornisce un supporto diretto per **JUnit**.
   - Questo è utile per promuovere lo sviluppo guidato dai test (Test-Driven Development, TDD) in cui i test sono scritti prima del codice di implementazione.

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

public class Tests {
	@Test
	public void getNameNotNull() {
		Person tester = new Person();
		assertNotNull(tester.getName());
	}

	@Test(expected=IllegalArgumentException.class)
	public void setNameNotNull() {
		Person tester = new Person();
		tester.setName(null);
	}
}
```

Esempio di Test con JUnit:

- Viene fornito un esempio di test JUnit nel codice Java riportato.
- La classe `Tests` contiene due metodi annotati con `@Test`, che sono i casi di test effettivi:
  - `getNameNotNull`: Verifica che il metodo `getName` di un oggetto `Person` restituisca un valore non nullo.
  - `setNameNotNull`: Verifica che l'assegnazione di un valore nullo al metodo `setName` di un oggetto `Person` generi un'eccezione di tipo `IllegalArgumentException`.

Utilizzo di Assert:

- Il metodo `assertEquals` viene utilizzato per confrontare valori.
- La classe `org.junit.Assert` fornisce varie asserzioni utili per verificare le condizioni nei test.

Torna all'indice

Indice

UML

UML (Unified Modeling Language) è un linguaggio di modellazione standardizzato utilizzato per visualizzare, specificare, costruire e documentare i sistemi software. Gli UML sono importanti in quanto forniscono un mezzo comune e comprensibile a tutti gli stakeholder (sviluppatori, analisti, clienti, ecc.) per comunicare e comprendere il design e la struttura di un sistema.

Class Diagram

Spiegazione consigliata: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/ (https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/).

![[43.png]]

Class Diagram (Diagramma delle Classi) in UML è uno dei diagrammi più fondamentali e ampiamente utilizzati. Rappresenta la struttura statica del sistema mostrando le classi, i loro attributi, i metodi e le relazioni tra di esse.

1. **Classe**
   - **Notazione:** È rappresentata come un rettangolo suddiviso in tre sezioni orizzontali.
   - **Sezioni:**
     - **Sezione superiore:** Nome della classe.
     - **Sezione centrale:** Attributi della classe con i rispettivi tipi di dati.
     - **Sezione inferiore:** Metodi della classe con le loro firme.
2. **Attributo**
   - **Notazione:** Nome dell'attributo seguito da due punti e dal tipo di dato.
   - **Esempio:** `nome: String`
3. **Metodo**
   - **Notazione:** Nome del metodo seguito da parentesi tonde contenenti i parametri di input e il tipo di ritorno.
   - **Esempio:** `calcolaPrezzo(prodotto: Prodotto): double`

4. **Associazione**
    - **Definizione:** Rappresenta una relazione tra due classi.
    - **Notazione:** Linea che collega due classi, con eventuali moltiplicità (numeri) sulle estremità per indicare la cardinalità della relazione.
    - **Esempio:** `1 -----------> 0..*`
5. **Multiplicità**
    - **Definizione:** Indica il numero di istanze della classe associata.
    - **Notazione:** Numeri o asterischi sulle estremità dell'associazione.
    - **Esempio:** `0..*` indica "zero o più".
6. **Generalizzazione (Ereditarietà)**
    - **Definizione:** Indica una relazione "è-un" tra classi.
    - **Notazione:** Linea con una freccia dalla sottoclasse alla superclasse.
    - **Esempio:** `Studente` (sottoclasse) eredita da `Persona` (superclasse).
7. **Interfaccia**
    - **Definizione:** Rappresenta un contratto, una collezione di firme di metodi che le classi implementano.
    - **Notazione:** Rettangolo con il nome preceduto dalla parola chiave "interface".
    - **Esempio:** `<<interface>> Pagamento`
8. **Classe Astratta**
    - **Definizione:** Una classe che non può essere istanziata, ma può essere usata come superclasse.
    - **Notazione:** Nome in corsivo o con una linea tratteggiata.
    - **Esempio:** `Shape` (classe astratta) con sottoclassi `Cerchio` e `Quadrato`.
9. **Dependency**
    - **Definizione:** Indica che una classe dipende da un'altra, ma senza una relazione forte.
    - **Notazione:** Linea tratteggiata con una freccia dalla classe dipendente alla dipendenza.
    - **Esempio:** `ClasseA` ----> `ClasseB`
10. **Composizione e Aggregazione**
    - **Definizione:** Indicano relazioni di "parte-di" tra classi.
    - **Notazione:** Rombo pieno (composizione) o rombo vuoto (aggregazione) sulla linea di connessione.
    - **Esempio:** `Scuola` (composizione) ha `Aule` (parti).
11. **Note e Commenti**
    - **Definizione:** Commenti o note per chiarire o fornire dettagli aggiuntivi.
    - **Notazione:** Nota attaccata a una classe o a un'altra forma con una linea tratteggiata.
    - **Esempio:** `// Questa classe rappresenta uno studente.`
12. **Pacchetto**
    - **Definizione:** Raggruppamento di classi correlati.
    - **Notazione:** Rettangolo con il nome del pacchetto.
    - **Esempio:** `it.mioProgetto.modello`

Torna all'indice

Sequence Diagram

Sequence Diagram (Diagramma di Sequenza) in UML è un diagramma di comportamento che rappresenta l'interazione tra oggetti o parti di un sistema nel tempo. Descrive come gli oggetti collaborano tra di loro per completare un'attività o un processo.

1. **Partecipanti (Attori)**
    - **Definizione:** Oggetti o entità che partecipano all'interazione.
    - **Notazione:** Rappresentati da box verticali in cima al diagramma, con il nome dell'attore.
2. **Oggetti**
    - **Definizione:** Istanze di classi o entità coinvolte nell'interazione.
    - **Notazione:** Box verticali sotto l'attore, con il nome dell'oggetto e la sua classe tra parentesi.
3. **Messaggi**
    - **Definizione:** Le comunicazioni tra gli oggetti o attori durante l'interazione.
    - **Notazione:** Linee di freccia tra gli oggetti, con etichette che indicano il tipo di messaggio e i parametri associati.
4. **Vita dell'Oggetto (Lifeline)**
    - **Definizione:** Rappresenta la vita di un oggetto durante l'interazione.
    - **Notazione:** Linea verticale (lifeline) sotto l'oggetto, con una freccia in cima che indica il suo ciclo di vita.
5. **Attivazione (Activation Bar)**
    - **Definizione:** Indica il periodo di tempo in cui un oggetto è attivo e sta eseguendo un'operazione.
    - **Notazione:** Barra orizzontale sulla vita dell'oggetto, posizionata sotto i messaggi che coinvolgono l'oggetto.
6. **Messaggi Asincroni e Sincroni**
    - **Definizione:**
        - **Messaggio Asincrono:** Non blocca il mittente; il mittente può continuare a eseguire altre attività senza attendere una risposta.
        - **Messaggio Sincrono:** Blocca il mittente fino a quando il destinatario non ha elaborato il messaggio.
    - **Notazione**

        - Messaggio Asincrono: Freccia tratteggiata.
        - Messaggio Sincrono: Freccia solida.
7. **Creazione ed Eliminazione di Oggetti**
    - **Definizione:**
        - **Creazione:** Indica quando un nuovo oggetto viene creato durante l'interazione.
        - **Eliminazione (Destroy):** Indica quando un oggetto viene eliminato.
    - **Notazione:**
        - Creazione: Punto con una "x" al centro sulla vita dell'oggetto.
        - Eliminazione: Barra orizzontale con una "x" sulle vita dell'oggetto.

Torna all'indice

State Diagram

![[45.png]]

Il Diagramma di Stato in UML è una rappresentazione visuale che modella il comportamento di un oggetto o di un sistema in termini dei vari stati attraverso cui può transire durante la sua esecuzione. Questo diagramma è particolarmente utile per modellare entità che passano attraverso stati distinti in risposta agli stimoli esterni o ai cambiamenti interni.

1. **Stati (States)**
    - **Definizione:** I diversi stati che un oggetto o un sistema può assumere.
    - **Notazione:** Rappresentati da riquadri rotondi, etichettati con il nome dello stato.
2. **Transizioni (Transitions)**
    - **Definizione:** Le condizioni o gli eventi che causano il passaggio da uno stato all'altro.
    - **Notazione:** Frecce che collegano gli stati, annotate con la condizione o l'evento che attiva la transizione.
3. **Eventi (Events)**
    - **Definizione:** Stimoli esterni o cambiamenti interni che innescano una transizione.
    - **Notazione:** Etichette lungo le frecce di transizione, indicando l'evento scatenante.
4. **Azioni (Actions)**
    - **Definizione:** Le attività o operazioni eseguite quando si verifica una transizione.
    - **Notazione:** Annotazioni sotto le frecce di transizione, indicando l'azione associata.
5. **Stati Finali (Final States)**
    - **Definizione:** Indicano la conclusione o la terminazione di un processo.
    - **Notazione:** Riquadri rotondi con un cerchio nero al loro interno.
6. **Stati Iniziali (Initial States)**
    - **Definizione:** Lo stato iniziale da cui inizia l'esecuzione.
    - **Notazione:** Una freccia entrante in uno stato rotondo da un punto iniziale.
7. **Regole di Guarigione (Guard Conditions)**
    - **Definizione:** Condizioni booleane che determinano se una transizione può verificarsi.
    - **Notazione:** Specificate accanto alle frecce di transizione, evidenziando le condizioni di passaggio.

Torna all'indice

Activity Diagram

![[46.png]]

Il Diagramma delle Attività in UML è uno strumento visuale che modella il flusso di lavoro o le attività all'interno di un sistema. È ampiamente utilizzato per rappresentare processi, procedure o algoritmi, mostrando come diverse attività si collegano e si svolgono nel contesto del sistema.

1. **Attività (Activity)**
    - **Definizione:** Rappresenta un'unità di lavoro o un'azione eseguita all'interno del sistema.
    - **Notazione:** Rappresentato da un rettangolo con il nome dell'attività al suo interno.
2. **Flusso di Controllo (Control Flow)**
    - **Definizione:** Indica l'ordine sequenziale delle attività all'interno del diagramma.
    - **Notazione:** Linee frecce che collegano le attività, mostrando la sequenza di esecuzione.
3. **Decisioni (Decision Nodes)**
    - **Definizione:** Rappresentano punti di decisione nel flusso di controllo, dove il percorso può divergere in base a una condizione.
    - **Notazione:** Rombi con una condizione scritta al loro interno. Le frecce indicano i diversi percorsi possibili.
4. **Fork e Join Nodes**
    - **Definizione:** Indicano punti nel diagramma in cui più flussi di controllo si separano (fork) o si riuniscono (join).
    - **Notazione:** Una barra orizzontale indica un fork, mentre una barra verticale indica un join.
5. **Nodi di Unione (Merge Nodes)**
    - **Definizione:** Specificano punti in cui flussi di controllo separati convergono e proseguono insieme.
    - **Notazione:** Rappresentati come cerchi.
6. **Swimlanes**
    - **Definizione:** Divisione dello spazio del diagramma in "corsie" verticali o orizzontali, ciascuna assegnata a un'entità specifica (es. ruolo, attore o sistema).

- **Notazione:** Rappresentate da linee orizzontali o verticali attraverso il diagramma.

7. **Partizioni (Partitions)**
   - **Definizione:** Suddivisione di attività correlate all'interno di una swimlane specifica.
   - **Notazione:** Linee tratteggiate che dividono la swimlane in sezioni.
8. **Flusso di Oggetto (Object Flow)**
   - **Definizione:** Indica il passaggio di oggetti tra diverse attività.
   - **Notazione:** Linea tratteggiata con una freccia che indica il flusso degli oggetti.

Torna all'indice

Indice

Design Pattern

I design patterns (pattern di progettazione) nel contesto del software sono soluzioni generali e riutilizzabili per problemi comuni che si verificano frequentemente nel design del software.

1. **Definizione:**
   - Un design pattern non è un design completo trasformabile direttamente in codice sorgente.
   - È una descrizione o un modello su come risolvere un problema, utilizzabile in diverse situazioni.
2. **Ruolo:**
   - I design patterns rappresentano le migliori pratiche formalizzate che il progettista può utilizzare per risolvere problemi comuni durante la progettazione di un sistema.
3. **Natura delle Soluzioni:**
   - I pattern di progettazione forniscono soluzioni generiche e adattabili, piuttosto che progettazioni finite direttamente implementabili.
4. **Principale Caratteristica:**
   - Sono best practice formalizzate, e il loro utilizzo consente di affrontare in modo efficiente problemi comuni senza dover riprogettare da zero ogni volta.
5. **Orientamento agli Oggetti:**
   - I design patterns orientati agli oggetti mostrano tipicamente le relazioni e le interazioni tra classi e oggetti, senza specificare le classi o gli oggetti finali che saranno coinvolti nell'applicazione.

Gli sviluppatori possono utilizzare i design patterns come strumenti concettuali per risolvere problemi ricorrenti durante la progettazione e lo sviluppo del software, facilitando la creazione di soluzioni robuste e manutenibili.

GoF

I design patterns hanno guadagnato popolarità nell'ingegneria del software dopo la pubblicazione del libro "Design Patterns: Elements of Reusable Object-Oriented Software" nel 1994, scritto dal cosiddetto Gang of Four (GoF), composto da E. Gamma, R. Helm, R. Johnson, J. Vlissides. Ecco una dettagliata spiegazione:

1. **Origine e Popolarità:**
   - I design patterns sono diventati popolari nel campo dell'ingegneria del software con la pubblicazione del libro del GoF nel 1994.
   - Il libro, intitolato "Design Patterns: Elements of Reusable Object-Oriented Software", è stato scritto dai quattro autori noti come Gang of Four.
2. **Categorie dei Design Patterns:**
   - Il libro GoF discute vari design patterns suddividendoli nelle seguenti classi:
     - **Creational Patterns (Pattern Creazionali):** Intenti a creare oggetti, evitando che il programmatore istanzi direttamente gli oggetti.
     - **Structural Patterns (Pattern Strutturali):** Utilizzano l'ereditarietà per comporre interfacce e definiscono modi per comporre oggetti per ottenere nuove funzionalità.
     - **Behavioral Patterns (Pattern Comportamentali):** Sono specificamente concentrati sui protocolli di comunicazione tra gli oggetti.

Questi design patterns forniscono un approccio strutturato per affrontare tipi specifici di problemi di progettazione, offrendo soluzioni che sono state validate e riconosciute come buone pratiche dalla comunità di sviluppatori.

Torna all'indice

Creational Patterns

I Creational Patterns, o Pattern Creazionali, sono una categoria di design patterns che si concentrano sulla creazione degli oggetti:

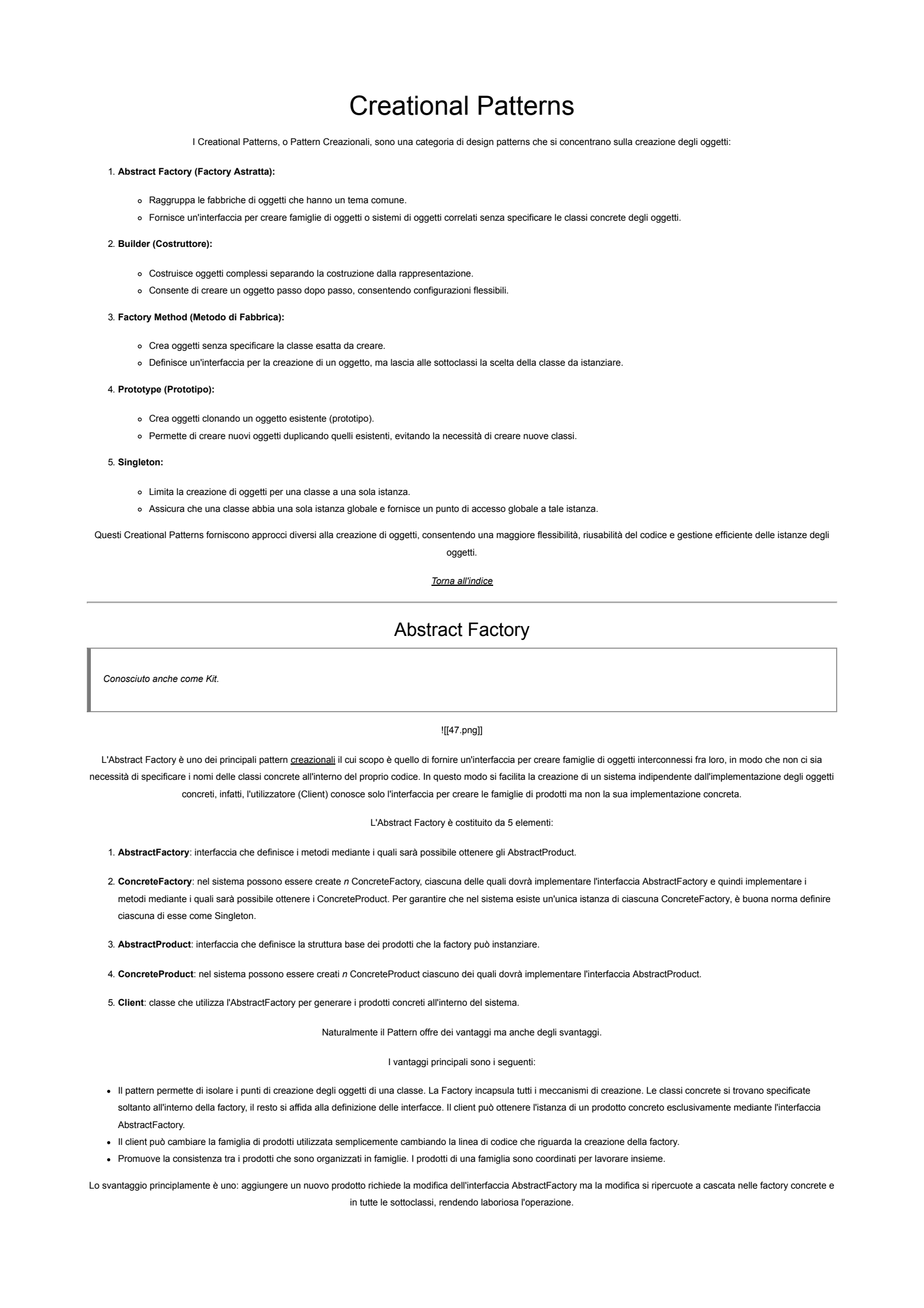
1. **Abstract Factory (Factory Astratta):**
    - Raggruppa le fabbriche di oggetti che hanno un tema comune.
    - Fornisce un'interfaccia per creare famiglie di oggetti o sistemi di oggetti correlati senza specificare le classi concrete degli oggetti.
2. **Builder (Costruttore):**
    - Costruisce oggetti complessi separando la costruzione dalla rappresentazione.
    - Consente di creare un oggetto passo dopo passo, consentendo configurazioni flessibili.
3. **Factory Method (Metodo di Fabbrica):**
    - Crea oggetti senza specificare la classe esatta da creare.
    - Definisce un'interfaccia per la creazione di un oggetto, ma lascia alle sottoclassi la scelta della classe da istanziare.
4. **Prototype (Prototipo):**
    - Crea oggetti clonando un oggetto esistente (prototipo).
    - Permette di creare nuovi oggetti duplicando quelli esistenti, evitando la necessità di creare nuove classi.
5. **Singleton:**
    - Limita la creazione di oggetti per una classe a una sola istanza.
    - Assicura che una classe abbia una sola istanza globale e fornisce un punto di accesso globale a tale istanza.

Questi Creational Patterns forniscono approcci diversi alla creazione di oggetti, consentendo una maggiore flessibilità, riusabilità del codice e gestione efficiente delle istanze degli oggetti.

Torna all'indice

Abstract Factory

> *Conosciuto anche come Kit.*



L'Abstract Factory è uno dei principali pattern creazionali il cui scopo è quello di fornire un'interfaccia per creare famiglie di oggetti interconnessi fra loro, in modo che non ci sia necessità di specificare i nomi delle classi concrete all'interno del proprio codice. In questo modo si facilita la creazione di un sistema indipendente dall'implementazione degli oggetti concreti, infatti, l'utilizzatore (Client) conosce solo l'interfaccia per creare le famiglie di prodotti ma non la sua implementazione concreta.

L'Abstract Factory è costituito da 5 elementi:

1. **AbstractFactory**: interfaccia che definisce i metodi mediante i quali sarà possibile ottenere gli AbstractProduct.
2. **ConcreteFactory**: nel sistema possono essere create *n* ConcreteFactory, ciascuna delle quali dovrà implementare l'interfaccia AbstractFactory e quindi implementare i metodi mediante i quali sarà possibile ottenere i ConcreteProduct. Per garantire che nel sistema esiste un'unica istanza di ciascuna ConcreteFactory, è buona norma definire ciascuna di esse come Singleton.
3. **AbstractProduct**: interfaccia che definisce la struttura base dei prodotti che la factory può instanziare.
4. **ConcreteProduct**: nel sistema possono essere creati *n* ConcreteProduct ciascuno dei quali dovrà implementare l'interfaccia AbstractProduct.
5. **Client**: classe che utilizza l'AbstractFactory per generare i prodotti concreti all'interno del sistema.

Naturalmente il Pattern offre dei vantaggi ma anche degli svantaggi.

I vantaggi principali sono i seguenti:

- Il pattern permette di isolare i punti di creazione degli oggetti di una classe. La Factory incapsula tutti i meccanismi di creazione. Le classi concrete si trovano specificate soltanto all'interno della factory, il resto si affida alla definizione delle interfacce. Il client può ottenere l'istanza di un prodotto concreto esclusivamente mediante l'interfaccia AbstractFactory.
- Il client può cambiare la famiglia di prodotti utilizzata semplicemente cambiando la linea di codice che riguarda la creazione della factory.
- Promuove la consistenza tra i prodotti che sono organizzati in famiglie. I prodotti di una famiglia sono coordinati per lavorare insieme.

Lo svantaggio principalmente è uno: aggiungere un nuovo prodotto richiede la modifica dell'interfaccia AbstractFactory ma la modifica si ripercuote a cascata nelle factory concrete e in tutte le sottoclassi, rendendo laboriosa l'operazione.

Il pattern AbstractFactory può essere utilizzato in un gran numero di situazioni reali. Per cercare di acquisire una certa dimestichezza con questo pattern e capirne meglio il funzionamento illustriamo un esempio di utilizzo in un contesto reale. Il nostro esempio simula la renderizzazione di una figura geometrica. Per semplicità implementiamo un'unica ConcreteFactory e soltanto due prodotti che non fanno altro che stampare una stringa a video.

Esempio - Abstract Factory

Analizziamo ora in dettaglio le singole interfacce/classi necessarie per implementare il pattern. Partiamo da **FiguraFactory** che rappresenta la nostra AbstractFactory. Definisce i metodi che ciascuna ConcreteFactory deve implementare: `createRettangolo()` e `createCerchio()`. Entrambi i metodi restituiscono un'istanza della classe Figura.

FiguraFactory:

```
public abstract class FiguraFactory {
         public abstract Figura createRettangolo();
         public abstract Figura createCerchio();
}
```

MiaFiguraFactory rappresenta la nostra unica ConcreteFactory. Tale classe deve estendere la classe astratta FiguraFactory e quindi, implementare i due metodi definiti: `createRettangolo()` e `createCerchio()` che restituiscono rispettivamente, un'istanza della classe MioRettangolo e un'istanza della classe MioCerchio.

MiaFiguraFactory:

```
public class MiaFiguraFactory extends FiguraFactory {
        public Figura createCerchio() {
                return new MioCerchio();
        }
        public Figura createRettangolo() {
                return new MioRettangolo();
        }
}
```

Figura rappresenta il nostro AbstractProduct che definisce la struttura base di un generico prodotto della famiglia. Per semplicità definiamo esclusivamente il metodo `disegna()`.

Classe Figura:

```
public abstract class Figura {
         public abstract void disegna();
}
```

MioCerchio e MioRettangolo sono i nostri ConcreteProduct che estendono la classe astratta Figura. Per semplicità i due metodi stampano soltanto una stringa a video.

ConcreteProduct: MioCerchio e MioRettangolo

```
public class MioCerchio extends Figura {
        public void disegna() {
                System.out.println("Io sono il cerchio");
        }
}

public class MioRettangolo extends Figura {
        public void disegna() {
                System.out.println("Io sono il rettangolo");
        }
}
```

Test rappresenta il nostro Client, cioè la classe che utilizza l'AbstractFactory per ottenere un'istanza dei nostri prodotti concreti. È possibile vedere come il Client non deve sapere nulla delle classi concrete che deve utilizzare.

Client: Classe Test

```
public class Test {
	public static void main(String[] args) {
		FiguraFactory factory = new MiaFiguraFactory();
		Figura c = factory.createCerchio();
		Figura r = factory.createRettangolo();
		c.disegna();
		r.disegna();
	}
}
```

Torna all'indice

Builder

Il Build Pattern e' un pattern creazionale che in molte situazioni può rappresentare una valida alternativa alla costruzione di oggetti mediante costruttori.

La necessità di introdurre meccanismi alternativi a quelli forniti da Java per la creazione di oggetti nasce dal fatto che talvolta le strutture sono molto complesse e non sempre è banale impostare un costruttore ben formato. Pensiamo ai casi in cui il numero di attributi sia molto alto oppure ai casi in cui ci sono attributi che possono anche non essere valorizzati. La probabilità di fare un errore scrivendo il costruttore a mano è molto alta.

L'obiettivo finale è quello di separare la creazione dell'oggetto dalla sua rappresentazione. In tale maniera l'algoritmo per la creazione dell'oggetto è indipendente dalle varie parti che costituiscono l'oggetto e da come vengono assemblate.

La creazione delle istanze e la loro gestione vengono quindi separate in modo da rendere il programma più semplice.

Un aspetto molto interessante è che questi meccanismi permettono di creare un oggetto passo passo, verificandone l'idoneità ad ogni passaggio (pensiamo a quando vogliamo costruire un oggetto con dati provenienti dai risultati di un parser) e soprattutto ci permette di nascondere la logica di controllo che sarebbe magari stata presente nell'eventuale costruttore.

> *Il Builder Pattern è usato per creare istanze di oggetti molto complessi con costruttori telescopici nella maniera più semplice.*

![[58.png]]

Analizziamo in dettaglio i vari componenti:

- **Product:** definisce il tipo di oggetto complesso che sarà generato dal **Builder Pattern**.
- **Builder:** questa **classe astratta** va a definire i vari passaggi per creare correttamente gli oggetti. Ogni metodo è generalmente astratto e le implementazioni sono fornite dalle sottoclassi concrete. Il metodo `getProduct()` è utilizzato per restituire il prodotto finale. Talvolta il Builder viene sostituito da un'interfaccia.
- **ConcreteBuilder:** possono esserci diverse sottoclassi concrete `ConcreteBuilder`. Queste sottoclassi forniscono i meccanismi per la creazione di oggetti complessi.
- **Director:** la classe Director controlla l'algoritmo per la creazione dei vari oggetti. Quando viene istanziata, il suo costruttore viene invocato. Contiene un parametro che indica quale `ConcreteBuilder` utilizzare per la creazione degli oggetti. Durante il processo di creazione, i vari metodi del `ConcreteBuilder` vengono richiamati e alla fine delle operazioni, il metodo `getProduct()` viene utilizzato per ottenere il prodotto finale.

Esempio - Builder

> *Esempio presente in Effective Java di Joshua Bloch.*

```
public class Animal {
	private final String id;
	private String name;
	private String pedigreeName;
	private String owner;
	private String race;
	private String residence;
	private Boolean isVaccinated;
	private Boolean isChampion;
	private List sons;
	private Sex sex;
	private Double weight;
	private Double height;

	public Animal(String name, String pedigreeName, String id, String owner, String race, String residence, Boolean isVaccinated,
		this.name = name;
		this.pedigreeName = pedigreeName;
		this.id = id;
		this.owner = owner;
		this.race = race;
		this.residence = residence;
		this.isVaccinated = isVaccinated;
		this.isChampion = isChampion;
		this.sons = sons;
		this.sex = sex;
		this.weight = weight;
		this.height = height;
	}

	public enum Sex {
		MALE,
		FEMALE
	}
} // ! Animal
```

Applichiamo ora il pattern:

```
public final class AnimalBuilder {
        private String id;
        private String name;
        private String pedigreeName;
        private String owner;
        private String race;
        private String residence;
        private Boolean isVaccinated;
        private Boolean isChampion;
        private List<String> sons;
        private Animal.Sex sex;
        private Double weight;
        private Double height;

        private AnimalBuilder(String id) {
                this.id = id;
        }

        public static AnimalBuilder newBuilder(String id) {
                return new AnimalBuilder(id);
        }

        public AnimalBuilder name(String name) {
                this.name = name;
                return this;
        }

        public AnimalBuilder pedigreeName(String pedigreeName) {
                this.pedigreeName = pedigreeName;
                return this;
        }

        public AnimalBuilder owner(String owner) {
                this.owner = owner;
                return this;
        }

        public AnimalBuilder race(String race) {
                this.race = race;
                return this;
        }

        public AnimalBuilder residence(String residence) {
                this.residence = residence;
                return this;
        }

        public AnimalBuilder isVaccinated(Boolean isVaccinated) {
                this.isVaccinated = isVaccinated;
                return this;
        }

        public AnimalBuilder isChampion(Boolean isChampion) {
                this.isChampion = isChampion;
                return this;
        }

        public AnimalBuilder sons(List<String> sons) {
                this.sons = sons;
                return this;
        }

        public AnimalBuilder sex(Animal.Sex sex) {
                this.sex = sex;
                return this;
```

```
        }

        public AnimalBuilder weight(Double weight) {
                this.weight = weight;
                return this;
        }

        public AnimalBuilder height(Double height) {
                this.height = height;
                return this;
        }

        public Animal build() {
                return new Animal(name, pedigreeName, id, owner, race, residence, isVaccinated, isChampion, sons, sex, weight, height
        }
} // ! AnimalBuilder
```

Un oggetto potrà ora essere istanziato come:

```
Animal pluto = AnimalBuilder.newBuilder("0000001")
        .name("0000001")
        .pedigreeName("PlutoSecondo")
        .owner("Marco Rossi")
        .race("labrador")
        .residence("Via x")
        .isVaccinated(true)
        .isChampion(false)
        .sons(null)
        .sex(Animal.Sex.MALE)
        .weight(40.5)
        .height(30.0)
        .build();
```

> *Premettiamo che in questo particolare caso, la classe astratta Builder non è strettamente indispensabile. Può essere aggiunta senza modificare radicalmente la struttura presentata qui sotto.*

Troviamo diversi vantaggi nell'utilizzo di questo pattern creazionale, infatti possiamo creare oggetti cloni, o comunque molto simili, minimizzando il codice da scrivere. Il metodo utilizzato è simile al seguente, facendo riferimento al builder istanziato sopra:

```
Animal animal3A = animalBuilder.build();
Animal animal3AClone = animalBuilder.build();
Animal animal3B = animalBuilder.sex(Animal.Sex.FEMALE).build();
```

Qui si creano due oggetti uguali e un oggetto simile ai due precedenti, ma con sesso opposto. Un vantaggio molto importante è quello di concentrare la validazione della classe in un unico metodo e di ottenere quindi oggetti pressochè immutabili.

Va precisato che la versione presentata è leggermente diversa da quella presentata nel modello originale. L'unico svantaggio dell'utilizzo del pattern è il fatto che vada necessariamente definita una classe builder per ogni oggetto, aumentando nettamente il tempo di sviluppo.

Torna all'indice

Factory Method

> *Conosciuto anche come Virtual Constructor.*

Si tratta di un pattern creazionale basato su classi e viene utilizzato per creare degli oggetti senza conoscerne i dettagli ma delegando un Creator che, in base alle informazioni ricevute, saprà quale oggetto restituire. Questo pattern consente di separare il Client dal Framework permettendo di modificare i dettagli implementativi senza dovere modificare il Client.

Questo pattern è composto dai seguenti partecipanti:

- Creator: dichiara la `Factory` che avrà il compito di ritornare l'oggetto appropriato.
- ConcreteCreator: effettua l'overwrite del metodo della `Factory` al fine di ritornare l'implementazione dell'oggetto.
- Product: definisce l'interfaccia dell'oggetto che deve essere creato dalla `Factory`.
- ConcreteProduct: implementa l'oggetto in base ai metodi definiti dall'interfaccia `Product`.

Tale pattern presenta i seguenti vantaggi/svantaggi:

1. Rappresenta un gancio alle sottoclassi: tramite il `Creator` è possibile scegliere quale classe concreta utilizzare e decidere di cambiarla senza avere nessun impatto verso il `Client`.
2. Consente di collegare gerarchie di classi in modo parallelo: i `ConcreteCreator` possono collegarsi con i `ConcreteProduct` e generare un collegamento parallelo tra gerarchie diverse.

Esempio - Factory Method

Come esempio pensiamo al caso in cui ci rechiamo in un centro commerciale per acquistare un paio di scarpe sportive, in particolare da ginnastica, quindi chiediamo al commesso di turno che ci rimanda al commesso specializzato nel settore di nostro interesse che ci consegnerà le scarpe di ginnastica che cercavamo.

Vediamo come si presenta il pattern in UML in base all'esempio:

Vediamo come si presenta la classe Cliente:

```
public class Cliente {
    public static void main(String[] args) {
        Commesso commesso = new Commesso();

        Scarpe scarpe = commesso.getScarpe("ginnastica");

        System.out.println(scarpe.getClass());
    }
} // ! Cliente
```

Vediamo la definizione del prodotto nella sua definizione e nelle sue implementazioni che nel nostro caso sono vuote per semplicità:

```
public interface Scarpe { }

public class ScarpeGinnastica implements Scarpe { }

public class ScarpeTennis implements Scarpe { }
```

Di seguito abbiamo l'implementazioni della Factory:

```
public class Commesso {
    public Scarpe getScarpe(String tipo) {

        Scarpe scarpe = null;

        if(tipo.equals("ginnastica"))
            scarpe = CommessoGinnastica.getScarpe();
        else if(tipo.equals("tennis"))
            scarpe = CommessoTennis.getScarpe();

        return scarpe;
    }
} // ! Commesso

public class CommessoGinnastica extends Commesso {
    public static Scarpe getScarpe(){
        return new ScarpeGinnastica();
    }
} // ! CommessoGinnastica

public class CommessoTennis extends Commesso {
    public static Scarpe getScarpe(){
        return new ScarpeTennis();
    }
} // ! CommessoTennis
```

Torna all'indice

Prototype

Si tratta di un pattern creazionale basato su oggetti e viene utilizzato per creare un nuovo oggetto clonando un oggetto già esistente detto prototipo. Questo pattern risulta utile affinchè il Client possa creare nuovi oggetti senza conoscerne i dettagli implementativi ma avvalendosi della clonazione.

> *La creazione del clone avviene a RunTime e non a CompileTime, pertanto il clone viene creato in sede di esecuzione.*

Durante la creazione del clone dell'oggetto occorre prestare molta attenzione alla creazione degli oggetti annidati. Una classe può contenere al suo interno dei riferimenti ad altre classi, pertanto la clonazione dell'oggetto principale deve effettuare la clonazione anche di tutti gli altri oggetti al suo interno.

La clonazione dell'intero albero degli oggetti genera un clone detto ***deep-clone*** in quanto copia tutti gli oggetti presenti. Se la clonazione si limita solo all'oggetto principale "contenitore" allora nel clone verranno mantenuti gli stessi riferimenti agli oggetti secondari: in questo caso si parla di ***shallow-clone***.

![[61.png]]

Tale pattern presenta i seguenti vantaggi/svantaggi:

1. Aggiungere / rimuovere prodotti a RunTime: è possibile decidere a Runtime se aggiungere nuovi oggetti.
2. Specificare nuovi oggetti cambiando il loro valore: invece di creare nuove classi per definire nuovi comportamenti, possiamo cambiare il valore di un oggetto per definire un nuovo comportamento. In questo modo vengono ridotti i numeri delle classi.

Esempio - Prototype

Come esempio pensiamo al caso in cui vogliamo creare dei template di tabelle di Hash da poter essere facilmente clonate all'occorrenza. In Java una tabella di hash può essere realizzata utilizzando per esempio le classi HashMap, IdentityHashMap, LinkedHashMap ognuna con caratteristiche diverse. Realizziamo una classe astratta Hash e specializziamo le classi di hash di nostro interesse.

Vediamo come si presenta in UML in base all'esempio:

![[62.png]]

La classe astratta Hash implementa l'interfaccia Clonable per dichiarare la propria volontà di clonazione.

La classe astratta Hash, ereditando di default dalla classe Object, eredita il metodo `protected native Object clone() throws CloneNotSupportedException;` che implementa di default la clonazione. Nel nostro caso richiameremo semplicemente il metodo nativo `super.clone()`:

```
public abstract class Hash implements Cloneable {

    @Override
    public Object clone() throws CloneNotSupportedException{
        return super.clone();
    }

    public abstract void addItem(Object key, Object value);

    public abstract int getSize();

}
```

Le classi seguenti MyLinkedHashMap, MyHashMap e MyIdentityHashMap ereditano dalla classe Hash ed effettuano l'overriding del metodo `clone()` *SENZA* deep-copy *MA* shadow-copy ossia non duplicano gli oggetti contenuti in esse.

```
public class MyLinkedHashMap extends Hash {
    private LinkedHashMap hash = new LinkedHashMap();

    @Override
    public Object clone() throws CloneNotSupportedException{
        return (MyLinkedHashMap) super.clone();
    }

    @Override
    public void addItem(Object key, Object value) {
        hash.put(key, value);
    }

    @Override
    public int getSize() {
        return hash.size();
    }
}
```

> *Le classi MyHashMap e MyIdentityHashMap non vengono implementate nell'esempio perche' sono praticamente uguali a MyLinkedHashMap.*

La classe Client ha il compito di invocare il template di interesse e richiedere la clonazione. Nel nostro esempio viene creato un template della classe MyLinkedHashMap per poi essere popolato con una chiave ed a questo punto viene clonato l'oggetto. Abbiamo fatto una shallow-copy, ossia una clonazione superficiale.

L'oggetto MyLinkedHashMap è stato duplicato e ce ne accorgiamo dal fatto che l'hashcode è diverso. L'oggetto annidato LinkedHashMap non è stato clonato e ce ne accorgiamo dal fatto che se proviamo ad aggiungere una nuova chiave al template questa viene "ritrovata" anche nell'oggetto clonato.

```
public class Client {

    public static void main(String[] args) throws CloneNotSupportedException {

        Hash hash = new MyLinkedHashMap();
        hash.addItem("key1", "value1");

        System.out.println("Prototype");
        System.out.println("ClassName: " + hash.getClass().getCanonicalName());
        System.out.println("ClassHashCode:" + hash.hashCode());

        Hash hashCloned = (MyLinkedHashMap) hash.clone();
        System.out.println("Clone:");
        System.out.println("ClassName: " + hashCloned.getClass().getCanonicalName());
        System.out.println("ClassHashCode:" + hashCloned.hashCode());

        System.out.println("Prototype Hashtable size: " + hash.getSize());
        System.out.println("Cloned Hashtable size: " + hashCloned.getSize());

        System.out.println("Adding new key");
        hash.addItem("key2", "value2");

        System.out.println("Prototype Hashtable size: " + hash.getSize());
        System.out.println("Cloned Hashtable size: " + hashCloned.getSize());

    } // ! main()
} // ! Client
```

Output:

```
$JAVA_HOME/bin/java patterns.prototype.Client
Prototype
ClassName: patterns.prototype.A
ClassHashCode:16130931
Clone:
ClassName: patterns.prototype.A
ClassHashCode:26315233
Prototype Hashtable size: 1
Cloned Hashtable size: 1
Adding new key
Prototype Hashtable size: 2
Cloned Hashtable size: 2
```

Guardando l'Object Diagram abbiamo questa situazione di condivisione dell'oggetto hash:

![[63.png]]

Per evitare questo problema occorre prevedere la deep-copy definiendo nell'overriding del metodo `clone()` la clonazione anche dell'oggetto annidato. Pertanto in tutte le classi che ereditano direttamente dalla classe astratta Hash modifichiamo il metodo `clone()` in questo modo:

```
@Override
public Object clone() throws CloneNotSupportedException{
    MyLinkedHashMap myLinkedHashMap = (MyLinkedHashMap) super.clone();
    myLinkedHashMap.hash = (LinkedHashMap) myLinkedHashMap.hash.clone();
    return myLinkedHashMap;
}
```

In questo modo quando eseguiamo la classe Client vediamo che l'inserimento di una nuova chiave nella nostra tabella di hash di template non determina alcuna modifica nella tabella di hash clonata.

Output:

```
$JAVA_HOME/bin/java patterns.prototype.Client
Prototype
ClassName: patterns.prototype.MyLinkedHashMap
ClassHashCode:16130931
Clone:
ClassName: patterns.prototype.MyLinkedHashMap
ClassHashCode:23660326
Prototype Hashtable size: 1
Cloned Hashtable size: 1
Adding new key
Prototype Hashtable size: 2
Cloned Hashtable size: 1
```

Guardando l'Object Diagram abbiamo questa situazione di NON condivisione dell'oggetto hash:

![[64.png]]

A seconda della complessità dell'oggetto template ed a seconda dell'alberatura annidata degli oggetti presenti nella classe template, occorrerà ricordarsi di clonare tutti gli oggetti presenti nel template.

Torna all'indice

Singleton

![[57.jpg]]

Il singleton è un pattern creazionale che viene utilizzato per mantenere una singola istanza di una classe e fornire un accesso globale a questa. L'utilizzo del pattern si presenta quando:

- Deve esistere esattamente una singola istanza di una classe, e deve essere accessibile dal cliente da un punto d'accesso ben preciso.
- Quando soltanto l'istanza della classe deve essere estesa mediante una sottoclasse, che il cliente deve essere in grado di utilizzare senza modificare il codice.

Il singleton è composto da un solo elemento:

- Singleton, che è responsabile della creazione dell'oggetto e definisce un'operazione `getInstance` che permette al cliente di accedere all'istanza univoca della classe.

> *Il cliente accede all'istanza univoca mediante l'operazione* `getInstance`.

I vantaggi principali sono:

- Accesso controllato ad una singola istanza: questo questo pattern si occupa di encapsulare l'istanza, avendo il controllo completo su di essa e gestendo come/quando i vari clienti possono accederci.
- Riduzione del namespace: il pattern è un miglioramento rispetto alle variabili globali, racchiudendo le variabili all'interno dell'istanza.

Tipo reference pre-inizializzato

```
public class Singleton {
        private static Singleton instance_ = new Singleton();

        public static Singleton getInstance(){
                return instance_;
        }
}
```

> *Questo caso va bene se il singleton viene usato in un sistema non concorrente (single thread e un unico core).*

Inizializzazione concorrente

```
public class Singleton {
        private static Singleton instance_ = null;

        public static Singleton getInstace() {
                if(instance_ == null) {
                        syncronized(Singleton.class) {
                                if(instance_ == null)
                                        instance_ = new Singleton();
                        }
                }

                return instance_;
        }

        private Singleton() { }
}
```

Torna all'indice

Structural Patterns

Gli structural design pattern sono un insieme di pattern che si concentrano sulla composizione di classi e oggetti per formare strutture più complesse. Questi pattern sono progettati per gestire la composizione di classi e oggetti in modo flessibile, consentendo la creazione di sistemi più estensibili e riutilizzabili. Ecco una breve panoramica di alcuni degli structural design pattern principali:

1. **Adapter Pattern (Pattern Adattatore):**
   - **Scopo:** Converte l'interfaccia di una classe in un'altra interfaccia che un cliente si aspetta.
   - **Utilizzo comune:** Quando si desidera utilizzare una classe esistente che non ha l'interfaccia desiderata.
2. **Bridge Pattern (Pattern Ponte):**
   - **Scopo:** Separa un'astrazione dalla sua implementazione, consentendo loro di evolvere indipendentemente.
   - **Utilizzo comune:** Quando si desidera evitare una connessione fissa tra un'astrazione e la sua implementazione.
3. **Composite Pattern (Pattern Composito):**
   - **Scopo:** Consente di trattare oggetti singoli e composizioni di oggetti in modo uniforme.
   - **Utilizzo comune:** Per creare strutture ad albero e rappresentare parti-tutto.
4. **Decorator Pattern (Pattern Decoratore):**
   - **Scopo:** Aggiunge responsabilità a un oggetto dinamicamente.
   - **Utilizzo comune:** Per estendere le funzionalità di un oggetto in modo flessibile e senza modificare il suo codice.
5. **Proxy Pattern (Pattern Proxy):**
   - **Scopo:** Fornisce un surrogato o un segnaposto per controllare l'accesso a un oggetto.
   - **Utilizzo comune:** Per implementare il controllo dell'accesso, il caricamento pigro o la registrazione.

Torna all'indice

Adapter

> *Conosciuto anche come Wrapper.*

Si tratta di un pattern strutturale basato su classi o su oggetti in quanto è possibile ottenere entrambe le rappresentazioni. Viene utilizzato quando si intende utilizzare un componente software ma occorre adattare la sua interfaccia per motivi di integrazione con l'applicazione esistente.

Questo comporta la definizione di una nuova interfaccia che deve essere compatibile con quella esistente in modo tale da consentire la comunicazione con l'interfaccia da "adattare". In tale contesto possono essere anche effettuate delle trasformazioni di dati per cui l'Adapter si occuperà di interfacciarsi con il nuovo sistema e fornisce anche le regole di mapping dei dati. Come abbiamo accennato, tale pattern può essere basato sia su classi che su oggetti pertanto l'instanza della classe da adattare può derivare da ereditarietà oppure da associazione.

Questo pattern è composto dai seguenti partecipanti:

- Client: colui che effettua l'invocazione all'operazione di interesse.
- Target: definisce l'interfaccia specifica del dominio applicativo utilizzata dal Client.
- Adaptee: definisce l'interfaccia di un diverso dominio applicativo da dover adattare per l'invocazione da parte del Client.
- Adapter: definisce l'interfaccia compatibile con il Target che maschera l'invocazione dell'Adaptee.

Abbiamo visto precedentemente che il pattern può essere basato su Classi o su Oggetti, in base a questo possiamo schematizzare in UML la relazione esistente tra l'adattatore e l'adattato (Adapter e Adaptee):

1. Sotto forma di ereditarietà come nel caso seguente: ![[65.png]]
2. Sotto forma di associazione come nel caso seguente: ![[66.png]]

Tale pattern presenta i seguenti vantaggi/svantaggi:

1. Class Adapter: prevede un rapporto di ereditarietà tra Adapter e Adaptee, in cui Adapter specializza Adaptee, pertanto non è possibile creare un Adapter che specializzi più Adaptee. Se esiste una gerarchia di Adaptee occorre creare una gereachia di Adapter.
2. Object Adapter: prevede un rapporto di associazione tra Adapter e Adaptee, in cui Adapter instanzia Adaptee, pertanto è possible avere un Adapter associato con più Adaptee.

Esempio - Adapter

Come esempio pensiamo al caso in cui dobbiamo gestire l'elenco dei dipendenti di una società. I loro dati vengono memorizzati in un java bean dal nome Impiegati che contiene tutte le informazioni personali (nel nostro caso per semplicità indichiamo solo il cognome).

Per effetto di una fusione societaria con una società straniera, il numero dei dipendenti aumenta ed occorre integrare il loro java bean dal nome Employer con quello esistente dal nome Impiegati. Semanticamente è uguale ma sintatticamente è diverso, pertanto creiamo la classe AdattatoreEmployer per adattare la classe Employer.

Sappiamo che possiamo utilizzare sia l'Object Adapter che il Class Adapter: la differenza principale dipende dal fatto che nel secondo caso è richiesta l'ereditarietà multipla e in java non è possibile, o meglio, non è possibile ereditare più classi ma è possibile implementare più interfacce. Questo significa che qualora disponiamo di una interfaccia Target ed un'interfaccia Adaptee possiamo utilizzare anche il Class Adapter.

Object Adapter

Per cominciare iniziamo ad implementare Object Adapter per eseguire l'esempio precedente. Vediamo come si presenta il pattern in UML in base all'esempio nel caso di Object Adapter:

![[67.png]]

A questo punto passiamo a creare la classe Impiegato e la classe Employer:

```
public class Impiegato {
    private String cognome = null;

    public String getCognome() {
        return cognome;
    }

    public void setCognome(String cognome) {
        this.cognome = cognome;
    }
} // ! Impiegato
```

```
public class Employer {
    private String lastName = null;

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
} // ! Employer
```

Creiamo la classe Adapter AdattatoreEmployer che eredita la classe Impiegato ed è associata con la classe Employer:

```
public class AdattatoreEmployer extends Impiegato {
    Employer employer = null;

    public AdattatoreEmployer(Employer employer) {
        this.employer = employer;
    }

    @Override
    public String getCognome() {
        return employer.getLastName();
    }

    @Override
    public void setCognome(String cognome) {
        employer.setLastName(cognome);
    }
} // ! AdattatoreEmployer
```

L'invocazione avviene ad opera della classe Client che passa per l'Adapter per invocare la classe Employer utilizzando gli stessi metodi utilizzati per invocare la classe Impiegato:

```
public class Client {
    public static void main(String[] args) {
        Impiegato impiegato = new Impiegato();
        impiegato.setCognome("Rossi");
        System.out.println("Impiegato: " + impiegato.getCognome());

        AdattatoreEmployer adattatoreEmployer =
                                                        new AdattatoreEmployer(new Employer());

        adattatoreEmployer.setCognome("Verdi");
        System.out.println("AdattatoreEmployer: " + adattatoreEmployer.getCognome());
    } // ! main()
} // ! Client
```

L'output del Client è mostrato di seguito:

```
$JAVA_HOME/bin/java patterns.adapter.object.Client
Impiegato: Rossi
AdattatoreEmployer: Verdi
```

Class Adapter

Sicuramente più complesso è il caso del Class Adapter. Vediamo di seguito il Class Diagram che ci mostra che l'Adapter eredita sia da Target che da Adaptee. Nel caso precedente Target ed Adaptee erano due classi, pertanto non era possibile l'ereditarietà multipla in Java, quindi abbiamo previsto due interfacce:

1. Una del Target dal nome InterfacciaImpiegato.
2. Una di Adaptee dal nome InterfacciaEmployer.

![[68.png]]

Le due interfacce definiscono i metodi presenti nelle due organizzazioni:

```
public interface InterfaceImpiegato {
    public String getCognome();
    public void setCognome(String cognome);
}
```

```
public interface InterfaceEmployer {
    public String getLastName();
    public void setLastName(String lastName);
}
```

Le classi Impiegato ed Employer ereditano dalle due interfacce nel modo seguente:

```
public class Impiegato implements InterfaceImpiegato {
    private String cognome = null;

    @Override
    public String getCognome() {
        return cognome;
    }
    @Override
    public void setCognome(String cognome) {
        this.cognome = cognome;
    }
}
```

```
public class Employer implements InterfaceEmployer {
    private String lastName = null;

    @Override
    public String getLastName() {
        return lastName;
    }

    @Override
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

L'Adapter dal nome AdattatoreEmployer può ereditare dalle due interfacce ed espone i metodi previsti da Target ed Adaptee.

```
public class AdattatoreEmployer extends Employer
                                                    implements InterfaceEmployer, InterfaceImpiegato {

    @Override
    public String getCognome() {
        return getLastName();
    }

    @Override
    public void setCognome(String cognome) {
        setLastName(cognome);
    }
}
```

L'invocazione da parte del Client consente di invocare Employer passando per la classe Adapter AdattatoreEmployer, come nel caso precedente Object Adapter.

```
public class Client {
    public static void main(String[] args) {
        Impiegato impiegato = new Impiegato();
        impiegato.setCognome("Rossi");
        System.out.println("Impiegato: " + impiegato.getCognome());

        AdattatoreEmployer adattatoreEmployer = new AdattatoreEmployer();
        adattatoreEmployer.setCognome("Verdi");
        System.out.println("AdattatoreEmployer: " + adattatoreEmployer.getCognome());
    } // ! main()
} // ! Client
```

L'output della classe Client è il seguente:

```
$JAVA_HOME/bin/java patterns.adapter.classes.Client
Impiegato: Rossi
AdattatoreEmployer: Verdi
```

Torna all'indice

Bridge

Si tratta di un pattern strutturale basato su oggetti che viene utilizzato per disaccoppiare dei componeti software. In questo modo è possibile effettuare uno switch a Run-Time, garantire il disaccoppiamento, nascondere l'implementazione, estendere la specializzazione delle classi.

Per esempio: si vuole cambiare l'interfaccia grafica della nostra applicazione da Motif a XWindow preservando la funzionalità di tutti i componenti grafici: in poche parole si vuole cambiare il LookAndFeel di tutti i tasti ma fare in modo che continuino a fare sempre la stessa cosa.

1. La prima idea, ma errata, sarebbe quella di creare 2 classi per ogni tasto, esempio ButtonXWindow e ButtonMotif; RadioXWindow e RadioMotif e via dicendo. In questo modo ci sarebbe un proliferare di classi da gestire. Ovviamente se viene introdotto un nuovo LookAndFeel occorrerà inserire tutte le classi di gestione dei tasti nuovi. In UML sarebbe così rappresentabile:

   ![[52.png]] Occorre separare la funzionalità e l'estetica, come?

2. Un'altra idea, più corretta, è quella di creare 2 gerachie: una per la funzionalità ed una per l'estetica. La funzionalità è composta dal tasto: Button, Radio. L'estetica è data dal LookAndFell: XWindow, Motif. Per disaccoppiare le gerachie definiamo 2 interfacce: la funzionalità nell'interfaccia Tasti e l'estetica nell'interfaccia LookAndFeel, successivamente possiamo implementare le 2 gerachie creando le classi concrete. In UML sarebbe così rappresentabile:

   ![[53.png]]

Partecipanti e Struttura Questo pattern è composto dai seguenti partecipanti:

1. **Client**: colui che effettua l'invocazione all'operazione di interesse.
2. **Abstraction**: definisce l'interfaccia del dominio applicativo utilizzata dal Client.
3. **RefinedAbstraction**: definisce l'implementazione dell'interfaccia utilizzata.
4. **Implementor**: definisce l'interfaccia da usare come Bridge e riferibile agli oggetti concreti da utilizzare.
5. **ConcreteImplementor**: implementa l'interfaccia Implementor usata come Bridge per il transito degli oggetti.

Possiamo schematizzare in UML:

![[54.png]]

Conseguenze Tale pattern presenta i seguenti vantaggi/svantaggi:

1. Disaccoppia l'interfaccia dall'implementazione: disaccoppiando Abstraction e Implementor è possibile gestire i cambiamenti delle classi concrete senza cablare nel codice dei riferiementi diretti.
2. Migliora l'estendibilità: è possibile estendere la gerarchia di Abstraction e Implementor senza problemi.
3. Nasconde l'implementazione al client: il Client non si deve porre il problema di conoscere l'implementazione delle classi concrete.

Esempio - Bridge

Facciamo un altro esempio: pensiamo al caso in cui ci rechiamo in un ristorante-pizzeria e facciamo un'ordinazione. Il cameriere addetto alla pizzeria prenderà la nostra ordinazione indipendentemente dal tipo di pizza che scegliamo.

Rappresentiamo questa situazione in questo Class Diagram UML:

![[55.png]]

L'interfaccia Cameriere definisce il metodo ordinazione che prende come parametro il Pasto: nel nostro caso sceglieremo una pizza.

```
public interface Cameriere {
    Pasto ordinazione(Pasto pasto);
}
```

La classe CamerierePizzeria implementa l'interfaccia Cameriere e ritorna il tipo di pasto che abbiamo scelto.

```
public class CamerierePizzeria implements Cameriere {
    public Pasto ordinazione(Pasto pasto) {
        return pasto;
    }
}
```

L'interfaccia Pasto definisce il tipo di piatto, pertanto qualunque pietanza ipotizzabile in un ristorante-pizzeria:

```
public interface Pasto {
    Pasto getPiatto();
}
```

La classe PizzaCapricciosa implementa come viene fatta la pizza Capricciosa.

```
public class PizzaCapricciosa implements Pasto {
    public Pasto getPiatto() {
        return this;
    }
}
```

Mentre la classe PizzaMargherita implementa come viene fatta la pizza Margherita.

```
public class PizzaMargherita implements Pasto {
    public Pasto getPiatto() {
        return this;
    }
}
```

Siamo arrivati alla classe Cliente che effettua l'ordinazione. Il nostro cliente ordina una pizza Margherita al cameriere addetto alle pizze.

```
public class Cliente {
    public static void main(String[] args) {
        Cameriere cameriere = new CamerierePizzeria();
        Pasto ordinazione = cameriere.ordinazione(new PizzaMargherita());
        System.out.println(ordinazione);
    }
}
```

L'ordine della pizza Margherita è stato eseguito. Possiamo aggiungere qualunque tipo di pizza implementando l'interfaccia Pasto disaccoppiandola con la classe CamerierePizzeria. Nascondiamo l'implementazione della pizza al cameriere che non è tenuto a sapere come viene fatta.

Estensione Visto e considerato che abbiamo parlato di un ristorante-pizzeria, immaginiamo di avere un angolo ristorante servito da un cameriere diverso da quello della pizzeria. Ovviamente abbiamo anche un menù ristorante che presenta altri piatti oltre alle pizze. Il cameriere addetto al ristorante implementa il metodo ordinazione dall'interfaccia Cameriere e si chiamerà CamerieреRistorante. I pasti del ristorante implementano l'interfaccia Pasto.

Rappresentiamo questa situazione in questo Class Diagram UML:

Vediamo come si presenta la classe CamerieреRistorante che si occupa di servire i clienti del ristorante.

```
public class CamerieреRistorante implements Cameriere {
    public Pasto ordinazione(Pasto pasto) {
        return pasto;
    }
}
```

Queste invece sono le classi che si occupano di implementare l'interfaccia Pasto per gestire altre pietanze: PastaFagioli e PastaPomodoro .

```
public class PastaFagioli implements Pasto {
    public Pasto getPiatto() {
        return this;
    }
}
```

```
public class PastaPomodoro implements Pasto {
    public Pasto getPiatto() {
        return this;
    }
}
```

Eseguiamo la classe Cliente che effettua 2 ordinazioni: PizzaMargherita ed un piatto di PastaPomodoro.

```
public class Cliente {
    public static void main(String[] args) {
        Cameriere[] cameriere  = new Cameriere[2];

        cameriere[0] = new CamerierePizzeria();
        Pasto pasto = cameriere[0].ordinazione(new PizzaMargherita());
        System.out.println(pasto);

        cameriere[1] = new CamerieреRistorante();
        pasto = cameriere[1].ordinazione(new PastaPomodoro());
        System.out.println(pasto);
    }
}
```

Torna all'indice

Composite

Si tratta di un pattern strutturale basato su oggetti che viene utilizzato quando si ha la necessità di realizzare una gerarchia di oggetti in cui l'oggetto contenitore può detenere oggetti elementari e/o oggetti contenitori. L'obiettivo è di permettere al Client che deve navigare la gerarchia, di comportarsi sempre nello stesso modo sia verso gli oggetti elementari e sia verso gli oggetti contenitori.

Partecipanti e Struttura Questo pattern è composto dai seguenti partecipanti:

- Client: colui che effettua l'invocazione all'operazione di interesse.
- Component: definisce l'interfaccia degli oggetti della composizione.
- Leaf: rappresenta l'oggetto foglia della composizione. Non ha figli. Definisce il comportamento "primitivo" dell'oggetto della composizione.
- Composite: definisce il comportamento degli oggetti usati come contenitori ed detiene il riferimento ai componenti "figli".

In UML, usando il Class Diagram, possiamo schematizzare le relazioni in questo modo:

Tale pattern presenta i seguenti vantaggi/svantaggi:

1. Definisce la gerarchia: Gli oggetti della gerarchia possono essere composti da oggetti semplici e/o da oggetti contenitori che a loro volta sono composti ricorsivamente da altri oggetti semplici e/o da oggetti contenitori.
2. Semplifica il client: il Client tratta gli oggetti semplici e gli oggetti contenitori nello stesso modo. Questo semplifica il suo lavoro il quale astrae dalla specifica implementazione.
3. Semplifica la modifica dell'albero gerarchico: l'alberatura è facilmente modificabile aggiungendo/rimuovendo foglie e contenitori.

Esempio - Composite

Come esempio pensiamo al FileSystem che presenta una struttura ad albero e che può essere composto da elementi semplici (i files) e da contenitori (le cartelle). L'obiettivo del nostro esercizio è quello di permettere al Client di accedere e navigare il File System senza conoscere la natura degli elementi che lo compongono in modo da consentire al Client di trattare tutti gli elementi nello stesso modo.

Per fare questo il Client userà la stessa interfaccia per l'accesso mentre l'implementazione nasconderà la gestione degli oggetti a seconda della loro reale natura.

Per fare questo schematizziamo la struttura delle classi usando il Class Diagram seguente:

Creiamo l'interfaccia di interrogazione per l'accesso a files e cartelle. Creiamo i metodi add/remove per aggiungere/rimuovere files/cartelle ed il metodo print per visualizzare il suo nome.

```
public interface MyFileSystem {
    public void add(MyFileSystem myFileSystem);
    public void remove(MyFileSystem myFileSystem);
    public void print();
}
```

Implementiamo la classe che gestisce i Files. In questo caso i metodo `add` e `remove` non sono implementabili.

```
public class MyFile implements MyFileSystem {

    private String myFileName = null;

    public MyFile(String myFileName) {
        this.myFileName = myFileName;
    }

    @Override
    public void print() {
        System.out.println(myFileName);
    }

    @Override
    public void add(MyFileSystem myFileNameSystem) {
        System.out.println("Impossible to add!");
    }

    @Override
    public void remove(MyFileSystem myFileNameSystem) {
        System.out.println("Impossible to remove!");
    }
} // ! MyFile
```

Implementiamo la classe che gestisce le Cartelle. In questo caso i metodo add/remove aggiungono/rimuovono nuovi files/cartelle alla cartella corrente.

```
public class MyFolder implements MyFileSystem {

    private String myFolderName;
    private ArrayList<MyFileSystem> folder;

    public MyFolder(String myFolderName) {
        this.myFolderName = myFolderName;
        folder = new ArrayList<MyFileSystem>();
    }

    @Override
    public void print() {
        System.out.println(myFolderName);
        for (int i = 0; i < folder.size(); i++) {
            folder.get(i).print();
        }
    }

    @Override
    public void add(MyFileSystem myFileSystem) {
        folder.add(myFileSystem);
    }

    @Override
    public void remove(MyFileSystem myFileSystem) {
        folder.remove(myFileSystem);
    }
} // ! MyFolder
```

Il metodo print viene invocato su tutti gli oggetti dell'albero, siano essi files o cartelle. Il comportamento sarà diverso nei 2 casi. Nella classe MyFile si limita a stampare il nome del file. Nella classe MyFolder, oltre a stampare il nome della cartella, presenta un loop utilizzato per invocare files/cartelle per consentire la ricorsione su tutta l'alberatura del Files System.

```
folder.get(i).print();
```

La classe Client crea l'alberatura del File System e poi visualizza il suo contenuto.

```
public class Client {
    public static void main(String[] args) {
        MyFileSystem f2 = new MyFile("F2");
        MyFileSystem f3 = new MyFile("F3");
        MyFileSystem c2 = new MyFolder("C2");
        c2.add(f2);
        c2.add(f3);

        MyFileSystem f1 = new MyFile("F1");
        MyFileSystem c1 = new MyFolder("C1");
        c1.add(f1);
        c1.add(c2);

        c1.print();
    }
}
```

L'output del Client è il seguente:

```
$JAVA_HOME/bin/java patterns.composite.Client
C1
F1
C2
F2
F3
```

Graficamente l'albero del File System può essere rappresentato in questo modo:

![[71.png]]

Torna all'indice

Decorator

Si tratta di un pattern strutturale basato su oggetti che viene utilizzato per aggiungere a RunTime delle funzionalità ad un oggetto. In Java, e più in generale nella programmazione ad oggetti, per aggiungere delle funzionalità ad una classe viene utilizzata l'ereditarietà che prevede la creazione di classi figlie che specializzano il comportamento della classe padre ma tutto ciò avviene a CompileTime.

Pertanto se in sede di definizione della struttura delle classi non vengono previste delle specifiche funzionalità, queste non saranno disponibili a RunTime. Al fine di superare questo limite, attraverso la decorazione è possibile aggiungere nuove funzionalità senza dover alterare la struttura delle classi ed i rapporti di parentela in quanto è possibile agire a RunTime per modificare il comportamento di un oggetto.

Per esempio: si vuole conoscere il tempo di esecuzione di un metodo ma tale funzionalità non è prevista nel metodo di nostro interesse. Come fare? Creiamo una classe "Decorator" da invocare al posto della classe originaria e che si occuperà di monitorare il tempo trascorso nell'invocazione del metodo originario. Come? Mantenendo una associazione alla classe originaria e calcolando il tempo di esecuzione del metodo. Vediamo l'esempio in seguito, in sede di implementazione.

Partecipanti e Struttura Questo pattern è composto dai seguenti partecipanti:

- Client: colui che effettua l'invocazione alla funzionalità di interesse.
- Component: definisce l'interfaccia degli oggetti per i quali verranno aggiunte nuove funzionalità.
- ConcreteComponent: definisce un oggetto al quale verrà aggiunta una nuova funzionalità.
- Decorator: definisce l'interfaccia conforme all'interfaccia del Component e mantiene l'associazione con l'oggetto Component.
- ConcreteDecorator: implementa l'interfaccia Decorator al fine di aggiungere nuove funzionalità all'oggetto.

Possiamo schematizzare in UML con il Class Diagram:

![[72.png]]

Tale pattern presenta i seguenti vantaggi/svantaggi:

1. Maggiore flessibilità rispetto alla eredità: permette di aggiungere funzionalità in modo molto più semplice rispetto all'ereditarietà
2. Funzionalità solo se richieste: consente di aggiungere delle funzionalità solo se occorrono realmente senza ereditare una struttura di classi che prevede un insieme di funzionalità di cui se ne utilizzeranno solo una parte. Nel caso in cui tali funzionalità sono anche a pagamento, consente di scegliere solo quelle strettamente necessarie da acquistare, coprendo esigenze di budget.
3. Aumento di micro-funzionalità: la presenza di molte classi Decorator di cui ognuna di esse aggiunge una micro funzionalità, può creare problemi in fase di comprensione o di debug del codice.

Esempio - Decorator

Un esempio noto del pattern Decorator lo troviamo nelle librerie java ed esattamente nelle classi di java.io.InputStream in cui i partecipanti sono cosi suddivisi:

- Component: la classe astratta InputStream.
- ConcreteComponent: le classi ByteArrayInputStream, FileInputStream, ObjectInputStream, PipedInputStream, SequenceInputStream e StringBufferInputStream.
- Decorator: la classe FilterInputStream.
- ConcreteDecorator: le classi BufferedInputStream, DataInputStream, LineNumberInputStream e PushbackInputStream.

L'utilizzo di questo pattern consente di poter scegliere la funzionalità di nostro interesse quando occorre leggere uno stream di dati, per esempio utilizzando BufferedInputStream è possibile bufferizzare lo stream.

Per esempio, possiamo wrappare la classe concreta FileInputStream di tipo ConcreteComponent con la classe concreta BufferedInputStream di tipo ConcreteDecorator, come nell'esempio seguente:

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;

public class FileDecorator {

    public static void main(String[] args) throws FileNotFoundException {
        InputStream inputStream = new BufferedInputStream(
                                                                    new FileInputStream("myFile"));
    }

} // ! FileDecorator
```

Riprendiamo ed estendiamo l'esempio iniziale: pensiamo al caso in cui abbiamo l'esigenza di monitorare l'invocazione di un metodo ma non abbiamo la possibilità di modificare il codice. Utilizziamo il pattern Decorator per esigenze di debug pertanto "wrappiamo" un metodo con delle semplici istruzioni print-screen.

Rappresentiamo questa situazione nel Class Diagram:

Creiamo l'interfaccia Component che dichiara il metodo interessanto.

```
public interface MyComponent {
    public void operation();
}
```

Implementiamo il metodo dichiarato nell'interfaccia MyComponent creando la classe ConcreteComponent.

```
public class ConcreteComponent implements MyComponent {
    public void operation() {
        System.out.println("Hello World");
    }
}
```

Definiamo l'interfaccia MyDecorator che si occupa di ereditare il metodo interessato da MyComponent e di interporsi con le classi di decorazione concrete.

```
interface MyDecorator extends MyComponent {
}
```

Creiamo la classe LoggingDecorator che implementa l'interfaccia MyDecorator ed aggiunge le informazioni di debug prima e dopo l'esecuzione del metodo interessato.

```
public class LoggingDecorator implements MyDecorator {

    MyComponent myComponent = null;

    public LoggingDecorator(MyComponent myComponent){
        this.myComponent = myComponent;
    }

        @Override
    public void operation() {
        System.out.println("First Logging");
        myComponent.operation();
        System.out.println("Last Logging");
    }
} // ! LoggingDecorator
```

La classe Client invoca la classe concreta LoggingDecorator passando al costruttore il componente concreto, successivamente invoca il metodo operation().

```
class Client {
    public static void main(String[] args) {
        MyComponent myComponent = new LoggingDecorator(new ConcreteComponent());
        myComponent.operation();
    }
}
```

L'output è il seguente:

```
$JAVA_HOME/bin/java patterns.decorator.Cliente
First Logging
Hello World
Last Logging
```

Partendo dall'esempio, possiamo creare una moltitudine di classi concrete Decorator che aggiungono nuove funzionalità. Per esempio possiamo creare una classe WaitingDecorator che preveda una pausa durante l'esecuzione. Vediamo come diventa il Class Diagram a seguito dell'inseriemento di questa nuova classe.


```
public class WaitingDecorator implements MyDecorator {

    MyComponent myComponent = null;

    public WaitingDecorator(MyComponent myComponent) {
        this.myComponent = myComponent;
    }

        @Override
    public void operation() {
        try {
            System.out.println("Waiting...");
            Thread.sleep(1000);
        }
        catch (Exception e) {}

        myComponent.operation();
    }

} // ! WaitingDecorator
```

Il Client invoca in modo annidato i Decorator tramite il loro costruttore, come segue:

```
public class Client {
    public static void main(String[] args) {
        MyComponent myComponent = new LoggingDecorator(
                                                                    new WaitingDecorator(
                                                                            new ConcreteComponent()));
        myComponent.operation();
    }
}
```

L'output è il seguente:

```
$JAVA_HOME/bin/java patterns.decorator.Cliente
First Logging
Waiting...
Hello World
Last Logging
```

Torna all'indice

Proxy

Si tratta di un pattern strutturale basato su oggetti che viene utilizzato per accedere ad un un oggetto complesso tramite un oggetto semplice.

Questo pattern può risultare utile se l'oggetto complesso:

- richiede molte risorse computazionali;
- richiede molto tempo per caricarsi;
- è locato su una macchina remota e il traffico di rete determina latenze ed overhead;
- non definisce delle policy di sicurezze e consente un accesso indiscriminato;
- non viene mantenuto in cache ma viene rigenerato ad ogni richiesta;

In tutti questi casi è possibile disposte delle politiche di gestione e/o di ottimizzazione.

A seconda del contesto, viene aggiunto un prefisso per descrivere il caso di riferimento, esempio:

- Virtual Proxy Pattern: ritarda la creazione e l'inizializzazione dell'oggetto poiché richiede grosse risorse (es: caricamento immagini).
- Remote Proxy Pattern: fornisce una rappresentazione locale dell'oggetto remoto (es: accesso ad oggetto remoto tramite RMI).
- Protection Proxy Pattern: fornisce un controllo sull'accesso dell'oggetto remoto (es: richiesta username/password per l'accesso).
- Smart Proxy Pattern: fornisce una ottimizzazione dell'oggetto (es: caricamento in memoria dell'oggetto).

Il proxy espone gli stessi metodi dell'oggetto complesso che maschera e questo permette di adattare facilmente l'oggetto senza richiedere modifiche.

Partecipanti e Struttura Questo pattern è composto dai seguenti partecipanti:

- Client: colui che effettua l'invocazione all'operazione di interesse.
- SubjectInterface: definisce l'interfaccia utilizzata dal Client che viene implementata dal Proxy e dal RealSubject.
- RealSubject: definisce l'oggetto reale di cui il Proxy avrà il compito di surrogare.
- Proxy: definisce la classe che avrà il compito di surrogare l'oggetto reale mantenendo una Reference a tale oggetto, creando e distruggendo l'oggetto ed esponendo gli stessi metodi pubblici dell'oggetto reale definiti dall'interfaccia.

Possiamo schematizzare in UML:

![[76.png]]

Esempio - Proxy

Molti esempi sono presenti nel seguente capitolo [[04-aop]].

Torna all'indice

Behavioral Patterns

I design pattern comportamentali, definiti dal Gang of Four (GoF), forniscono soluzioni per organizzare il comportamento delle classi e degli oggetti. Ecco una breve spiegazione di alcuni dei design pattern comportamentali principali:

1. **Command (Comando):** Il pattern Command incapsula una richiesta come un oggetto, consentendo di parametrizzare client con richieste diverse, accodare richieste, o supportare operazioni annullabili. Consiste di tre componenti principali: il Client (chi emette la richiesta), il Command (l'oggetto che incapsula la richiesta), e il Receiver (l'oggetto che effettua l'azione).

2. **Interpreter (Interprete):** Il pattern Interpreter fornisce un modo per interpretare il linguaggio di un sistema. Definisce una grammatica per il linguaggio e un interprete che utilizza questa grammatica per interpretare frasi del linguaggio. È utile quando c'è la necessità di eseguire operazioni su alberi di sintassi astratta.

3. **Iterator (Iteratore):** L'Iterator pattern fornisce un modo per accedere sequenzialmente agli elementi di una collezione senza esporre i dettagli interni della collezione. Un oggetto iteratore è utilizzato per attraversare gli elementi di una collezione in modo standard, indipendentemente dalla rappresentazione interna della collezione.

4. **Observer (Osservatore):** Il pattern Observer definisce una dipendenza uno a molti tra oggetti in modo che quando uno oggetto cambia stato, tutti i suoi osservatori vengano notificati e aggiornati automaticamente. Gli osservatori si registrano per ricevere notifiche e vengono informati quando avviene un cambiamento nell'oggetto osservato.

5. **Visitor (Visitatore):** Il pattern Visitor rappresenta un'operazione da eseguire su elementi di una struttura di oggetti senza modificare le classi degli elementi. Definisce un'interfaccia Visitor con un metodo per ogni tipo di elemento che può essere visitato. Le classi concrete implementano questa interfaccia per definire il comportamento specifico quando un elemento viene visitato.

Torna all'indice

Command

Si tratta di un pattern comportamentale basato su oggetti e viene utilizzato quando si ha la necessità di disaccoppiare l'invocazione di un comando dai suoi dettagli implementativi, separando colui che invoca il comando da colui che esegue l'operazione.

Tale operazione viene realizzata attraverso questa catena: \(Client \to Invocatore \to Ricevitore\)

- Il Client non è tenuto a conoscere i dettagli del comando ma il suo compito è solo quello di chiamare il metodo dell' Invocatore che si occuperà di intermediare l'operazione.
- L'Invocatore ha l'obiettivo di incapsulare, nascondere i dettagli della chiamata come nome del metodo e parametri.
- Il Ricevitore utilizza i parametri ricevuti per eseguire l'operazione

Ma tra l'Invocatore ed il Ricevitore viene posto il Command ossia il comando da eseguire. Il Command è una semplice interfaccia che viene implementata da una o più classi concrete che invocano il Receiver.

Partecipanti e Struttura Questo pattern è composto dai seguenti partecipanti:

- Client: colui che richiede il comando ed imposta il Receiver.
- Invoker: colui che effettua l'invocazione del comando.
- Command: interfaccia generica per l'esecuzione del comando.
- ConcreteCommand: implementazione del comando che consente di collegare l'Invoker con il Receiver.
- Receiver: colui che riceve il comando e sa come eseguirlo.

Tale pattern presenta i seguenti vantaggi/svantaggi:

1. Riduce l'accoppiamento: il Command disaccoppia l'Invoker dal Receiver, ossia colui che invoca da colui che esegue facendo in modo che i dettagli implementativi siano a conoscenza solo del Receiver.
2. Facile estendibilità: è possibile aggiungere facilmente nuovi comandi implementando l'interfaccia Command

Esempio - Command

Creiamo un'interfaccia Ordine che funge da comando, dopo di chè creiamo una classe magazzino che funge da richiesta. Inoltre creiamo le classi concrete Compra e Vendi che implementano l'interfaccia Ordine, la quale eseguirà la vera elaborazione dei comandi.

Infine, creiamo la classe command che funge da invoker, la classe può prendere ed effettuare ordini. La classe identifica l'oggetto che eseguirà il comando in funzione del tipo.

Step 1: Creiamo l'interfaccia Ordine

```
public interface Ordine {
    void esegui();
}
```

Step 2: Creiamo la classe Magazzino, che funge da richiesta

```
public class Magazzino {

    private String nome = "Prodotto_x";
    private int quantità = 25;

    public void compra(){
        System.out.println("[MAGAZZINO]: [ NOME: " + nome + ", QUANTITA': " + quantità + " ] COMPRATO");
    }
    public void vendi(){
        System.out.println("[MAGAZZINO]: [ NOME: " + nome + ", QUANTITA': " + quantità + " ] VENDUTO");
    }

} // ! Magazzino
```

Step 3: Creiamo le classi concrete che estendono l'interfaccia

```
public class Compra implements Ordine {
    private Magazzino magazzino;

    public Compra(Magazzino magazzino){
        this.magazzino = magazzino;
    }

        @Override
    public void esegui() {
        magazzino.compra();
    }
}
```

```
public class Vendi implements Ordine {
    private Magazzino magazzino;

    public Vendi(Magazzino magazzino){
        this.magazzino = magazzino;
    }

        @Override
    public void esegui() {
        magazzino.vendi();
    }
}
```

Step 4: Creiamo la classe Command

```
import java.util.ArrayList;
import java.util.List;

public class Command {
    private List ordineList = new ArrayList();

    public void prendiOrdine(Ordine order){
        ordineList.add(order);
        System.out.println("[COMMAND]: prendiOrdine()::Ordine aggiunto alla lista!");
    }

    public void piazzaOrdine(){
        for (Ordine ordine : ordineList) {
            ordine.esegui();
        }
        System.out.println("[COMMAND]: piazzaOrdine()::Ordini eseguiti!");
        ordineList.clear();
    }
} // ! Command
```

Step 5: Creiamo il Client

```
public class Client {
    public static void main(String[] args) {
        Magazzino magazzino = new Magazzino();

        Ordine compraOrdine = new Compra(magazzino);
        Ordine vendiOrdine = new Vendi(magazzino);

        Command broker = new Command();
        broker.prendiOrdine(compraOrdine);
        broker.prendiOrdine(vendiOrdine);

        broker.piazzaOrdine();
    }
}
```

Output

```
[COMMAND]: prendiOrdine()::Ordine aggiunto alla lista!
[COMMAND]: prendiOrdine()::Ordine aggiunto alla lista!
[MAGAZZINO]: [ NOME: Prodotto_x, QUANTITA': 25 ] COMPRATO
[MAGAZZINO]: [ NOME: Prodotto_x, QUANTITA': 25 ] VENDUTO
[COMMAND]: piazzaOrdine()::Ordini eseguiti!
```

Torna all'indice

Interpreter

Si tratta di un pattern comportamentale basato su classi e viene utilizzato quando si vuole definire una grammatica e il relativo interprete. La grammatica è costituita da tutte le espressioni che possono essere utilizzate mentre l'interprete permette di valutare il risultato complessivo.

Pensiamo ad una lingua straniera, per esempio lo spagnolo, che è costituita da una serie di vocabili e da regole grammaticali che disciplinano il loro ordine ed uso. Un interprete conoscendo i vocabili e le regole grammaticali riuscirà a capire il significato di una frase. Anche la matematica può essere usata come metafora, in cui i numeri rappresentano i dati e le operazioni rappresentano le funzioni, entrambi devono essere disposti secondo un preciso ordine quindi dovranno rispettare delle regole sintattiche. Un matematico è colui che conoscendo queste regole sarà in grado di capire il risultato di una operazione. Pensiamo per esempio ad una semplice addizione: 3 + 4. In questo caso la grammatica è costituita da 2 espressioni: dai numeri "3" e "4" e dall'operatore "+", l'interprete analizza le espressioni per ottenere il risultato, ossia "7".

Il termine "espressione" è utilizzato per definire un simbolo ed il suo comportamento.

- *Nel caso di un numero*: il simbolo è costituito da un carattere rappresentato dauno o più di questi valori "0123456789" mentre il comportamento è costituito dal valore del simbolo quindi il carattere "1" ha il valore del numero 1 (in Java `Integer.parseInt("1")`).
- *Nel caso di operazioni aritmentiche*: l'addizione utilizza il simbolo "+" ed il suo comportamento è costituito dall'operazione di somma.

Le espressioni possono essere semplici o composte a seconda che aggregano o meno altre espressioni e vengono definite:

- Terminali: quando definiscono in modo autonomo il comportamento, come nel caso di un numero che definisce il suo simbolo ed il suo comportamento
- Non terminali: quando dipendono da altre espressioni, come nel caso dell'addizione che definisce il suo simbolo ma il suo comportamento è associato ai numeri utilizzati per effettuare la somma.

Quindi possiamo creare una nostra calcolatrice elementare, per fare questo dobbiamo definire una grammatica costituita da numeri ed operazioni aritmetiche, pertanto abbiamo bisogno di 5 espressioni: numeri, addizioni, sottrazioni, moltiplicazioni e divisioni. Tali espressioni vengono valutare da un interprete che elabora il risultato.

Di seguito vediamo un esempio che utilizza questo pattern per realizzare una semplice calcolatrice.

Partecipanti e Struttura Questo pattern è composto dai seguenti partecipanti:

- Client: colui che costruisce un albero sintattico costituito da TerminalExpression e NonTerminalExpression che dovrà essere elaborato dall'interprete.
- Context: contiene le informazioni che l'interprete dovrà utilizzare.
- AbstractExpression: definisce un comportamento astratto che le espressioni devono implementare.
- TerminalExpression: implementa il comportamento dell'espressione semplice.
- NonterminalExpression: implementa il comportamento dell'espressione composta richiamando il comportamento delle espressioni semplici.

![[79.png]]

Tale pattern presenta i seguenti vantaggi/svantaggi:

1. Facilità nel cambiare la grammatica: attraveso l'estensione delle classi è possibile inserire e modificare la grammatica.

2. Difficoltà nel gestire una grammatica complessa: quando la grammatica contiene molte regole risulta molto complicato riuscire a gestirla e comprendere il flusso dell'albero sintattico.

Esempio - Interpreter

Realizziamo una calcolatrice che utilizza tutte e quattro le operazioni aritmentiche elementari. In questo caso inseriamo gli elementi in un modo più intuitivo utilizzando il classico metodo "infisso" cioè numero, operatore, numero, operatore ecc, così come facciamo quando utilizziamo la calcolatrice. In questo modo possiamo sia usare sempre lo stesso operatore (\(5 + 7 + 9\)) che usare diversi operatori (\(3 + 4 - 2 * 6\)). Ovviamente dobbiamo creare delle classi che gestiscono tutte e quattro le operazioni aritmetiche e lasciare al client l'ordine in cui tali operazioni vengono svolte (\(3-4+6\) oppure \(3+4-6\)).

> *Solitamente ai fini computazionali il metodo "infisso" è poco usato e si preferisce usare il metodo "prefisso" o "postfisso" per motivi prestazionali, in quanto questi metodi utilizzano meglio la memoria e consentono una migliore gestione delle priorità degli operatori.*

Vediamo come si presenta il pattern in UML in base all'esempio della calcolatrice:

L'interfaccia Espressione definisce il metodo "interpreta" che le classi concrete dovranno implementare.

```
public interface Espressione {
    public int interpreta(Contesto operazione);
}
```

La classe Contesto prevede la gestione dell'operazione corretta in base all'operatore. Il metodo revOperazione è utilizzato per invertire l'ordine dello Stack e processare i numeri e gli operatori nell'ordine di inserimento.

```
public class Contesto {

    private Stack numeri = null;
    private Stack operatori = null;

    public Contesto(String operazione) {
        this.numeri = new Stack();
        this.operatori = new Stack();

        for (String token : revOperazione(operazione)) {
            if (token.equals("+")) {
                operatori.add(new Addizione());
            } else if (token.equals("-")) {
                operatori.add(new Sottrazione());
            } else if (token.equals("/")) {
                operatori.add(new Divisione());
            } else if (token.equals("*")) {
                operatori.add(new Moltiplicazione());
            } else {
                numeri.add(new Numero(token));
            }
        }
    }

    public Espressione getNumero() {
        return numeri.pop();
    }

    public void setNumero(Espressione exp) {
        numeri.push(exp);
    }

    public Espressione getOperatore() {
        return operatori.pop();
    }

    private String[] revOperazione(String operazione) {
        List listOperation = Arrays.asList(operazione.split(" "));
        Collections.reverse(listOperation);
        return (String[]) listOperation.toArray();
    }
} // ! Contesto
```

La classe Numero è il primo esempio di una classe concreta e terminale che identifica il simbolo del numero e memorizza il suo valore.

```
public class Numero implements Espressione {
    private int numero;

    public Numero(String numero){
        this.numero = Integer.parseInt(numero);
    }

    @Override
    public int interpreta(Contesto contesto) {
        return numero;
    }
}
```

La classe Addizione:

```
public class Addizione implements Espressione {

    @Override
    public int interpreta(Contesto contesto) {
        int risultato = contesto.getNumero().interpreta(contesto) +
                                    contesto.getNumero().interpreta(contesto);
        contesto.setNumero(new Numero(risultato + ""));
        return risultato;
    }
}
```

La stessa cosa viene effettuata anche per la classe Sottrazione:

```
public class Sottrazione implements Espressione {

    @Override
    public int interpreta(Contesto contesto) {
        int risultato = contesto.getNumero().interpreta(contesto) -
                                    contesto.getNumero().interpreta(contesto);
        contesto.setNumero(new Numero(risultato + ""));
        return risultato;
    }

}
```

La classe Moltiplicazione:

```
public class Moltiplicazione implements Espressione {

    @Override
    public int interpreta(Contesto contesto) {
        int risultato = contesto.getNumero().interpreta(contesto) *
                                    contesto.getNumero().interpreta(contesto);
        contesto.setNumero(new Numero(risultato + ""));
        return risultato;
    }

}
```

La classe Divisione:

```
public class Divisione implements Espressione {

    @Override
    public int interpreta(Contesto contesto) {
        int risultato = contesto.getNumero().interpreta(contesto) /
                                    contesto.getNumero().interpreta(contesto);
        contesto.setNumero(new Numero(risultato + ""));
        return risultato;
    }

}
```

La classe Client definisce l'operazione da effettuare e la inserisce nel Contesto, poi esegue le operazioni presenti: addizione, sottrazione, divisione, moltiplicazione attraverso il riconoscimento (interpretazione) dell'operazione

```
public class Client {

    public static void main(String[] args) {
        //Contesto delle variabili ed operatori
        String operazione = "45 + 38 - 13 / 21 * 16";
        Contesto contesto = new Contesto(operazione);

        //Risultato
        int risultato = 0;
        while (true) {
            try {
                Espressione operatore = contesto.getOperatore();
                risultato = operatore.interpreta(contesto);
            } catch (java.util.EmptyStackException ese) {
                break;
            }
        }
        System.out.println(operazione + " = " + risultato );
    }
} // ! Client
```

Il risultato è lo svolgimento dell'operazione.

```
$JAVA_HOME/bin/java patterns.interpreter.calcolatrice.Client
45 + 38 - 13 / 21 * 16 = 48
```

> *Altro esempio: esercizio 8 visto a lezione.*

Torna all'indice

Iterator

Si tratta di un pattern comportamentale basato su oggetti e viene utilizzato quando, dato un aggregato di oggetti, si vuole accedere ai suoi elementi senza dover esporre la sua struttura. L'obiettivo di questo pattern è quello di disaccoppiare l'utilizzatore e l'implementatore dell'aggregazione di dati, tramite un oggetto intermedio che esponga sempre gli stessi metodi indipendentemente dall'aggregato di dati.

E' costituito da 3 soggetti: l'*Utilizzatore* dei dati, l'*Iteratore* che intermedia i dati e l'*Aggregatore* che detiene i dati secondo una propria logica.

Partecipanti e Struttura

Questo pattern è composto dai seguenti partecipanti:

- **Iterator**: colui che espone i metodi di accesso alla struttura dati.
- **ConcreteIterator**: implementa l'Iteratore e tiene il puntatore alla struttura dati.
- **Aggregator**: definisce l'interfaccia per creare un oggetto di tipo Iteratore.
- **ConcreteAggregator**: implementa l'interfaccia di creazione di un oggetto Iteratore.

![[81.png]]

Tale pattern presenta i seguenti vantaggi/svantaggi:

- Unica interfaccia di accesso: l'accesso ai dati avviene tramite l'Iterator che espone un'unica interfaccia e nasconde le diverse implementazioni degli Aggregator.
- Diversi iteratori di accesso: l'Aggregator può essere attraversato tramite diversi Iterator in cui ogni Iterator nasconde un algoritmo diverso.

Esempio - Iterator

Creeremo un'interfaccia Iterator che narra il metodo di navigazione e un'interfaccia Container che ritrasforma l'iteratore. Dopo di chè creeremo le classi concrete che implementano le due interfacce.

Step 1: Creiamo le due interfacce

```
public interface Container {
    public Iterator getIterator();
}
```

```
public interface Iterator {
    public boolean hasNext();
    public Object next();
}
```

Step 2: Creiamo la classe Repository che implementa l'interfaccia Container

Attenzione! La classe possiede all'interno un'altra classe il cui scopo è quello di implementare Iterator.

```
public class NameRepository implements Container {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

   @Override
   public Iterator getIterator() {
      return new InnerNameIterator();
   }

   private class InnerNameIterator implements Iterator {

      int index;

      @Override
      public boolean hasNext() {
         if(index < names.length)
            return true;

         return false;
      }

      @Override
      public Object next() {
         if(this.hasNext())
            return names[index++];

         return null;
      }

   } // ! InnerNameIterator
} // ! NameRepository
```

Step 3: Creiamo il main

```
public class Main {
    public static void main(String[] args) {
        Repository repository = new Repository();

        System.out.println("[MAIN]: Recupero nomi in Repository");

        for(Iterator iterator = repository.getIterator(); iterator.hasNext();){
            String name = (String) iterator.next();
            System.out.println("[MAIN]: Nome -> " + name);
        }
    }
}
```

Output:

```
[MAIN]: Recupero nomi in Repository
[MAIN]: Nome -> Roberto
[MAIN]: Nome -> Giovanni
[MAIN]: Nome -> Giulia
[MAIN]: Nome -> Andrea
```

Torna all'indice

Observer

Si tratta di un pattern comportamentale basato su oggetti che viene utilizzato quando si vuole realizzare una dipendenza uno-a-molti in cui il cambiamento di stato di un soggetto venga notificato a tutti i soggetti che si sono mostrati interessati.
Un esempio molto semplice è rappresentato dalle newsletters in cui gli utenti interessati a degli argomenti inseriscono il loro indirizzo email ed a fronte di novità inerenti gli argomenti, riceveranno una email di notifica. In questo modo viene applicata una gestione ad eventi, cioè al verificarsi di una notizia i soggetti interessati verranno informati tramite email. In questo modo l'interessato evita di fare polling, cioè evita di fare continue richieste al soggetto osservato per sapere se è avvenuto o meno un cambiamento ma al contrario verrà notificato in push dal soggetto osservato nel caso in cui dovesse interviene una modifica.

Questo pattern viene impegato in molte librerie, nei toolkit delle GUI e nel pattern architetturale MVC.

Nel pattern MVC abbiamo la presenza di 3 soggetti: il Model, la View ed il Controller. Questi soggetti svolgono compiti diversi e tra di loro è presente una separazione di responsabilità c.d. "separation of concern". Ma c'è da dire che tra di loro esiste un forte legame in merito al cambiamento di stato. In particolare il Controller è interessato ai cambiamenti di stato della View, mentre la View è interessata ai cambiamenti di stato del Model. Questo comporta che nel caso in cui dovessero avvenire dei cambiamenti il Model notifica alla View mentre la View notifica al Controller. Quindi il pattern Observer trova applicazione 2 volte nell'MVC su coppie di soggetti diversi (Model-View e View-Controller). La View svolge un ruolo doppio poichè si trova ad essere *osservata* dal Controller e nello stesso tempo ad essere *osservatore* nei confronti del Model. A differenza del Model e del Controller che invece giocano un ruolo singolo , infatti il Model è *osservato* dalla View mentre il Controller è un *osservatore* della View.

![[82.png]]

Il ruolo di osservatore è il ruolo svolto da colui che si mostra interessato ai cambiamenti di stato, c.d. Observer. Il ruolo di osservato è il ruolo svolto da colui che viene monitorato, c.d. Subject o Observable.

Partecipanti e Struttura

Questo pattern è composto dai seguenti partecipanti:

1. **Subject**: espone l'interfaccia che consente agli osservatori di iscriversi e cancellarsi; mantiene una reference a tutti gli osservatori iscritti
2. **Observer**: espone l'interfaccia che consente di aggiornare gli osservatori in caso di cambio di stato del soggetto osservato.
3. **ConcreteSubject**: mantiene lo stato del soggetto osservato e notifica gli osservatori in caso di un cambio di stato.
4. **ConcreteObserver**: implementa l'interfaccia dell'Observer definendo il comportamento in caso di cambio di stato del soggetto osservato

Tale pattern presenta i seguenti vantaggi/svantaggi:

- Astratto accoppiamento tra Subject e Observer: il Subject sa che una lista di Observer sono interessati al suo stato ma non conosce le classi concrete degli Observer, pertanto non vi è un accoppiamento forte tra di loro.
- Notifica diffusa: il Subject deve notificare a tutti gli Observer il proprio cambio di stato, gli Observer sono responsabili di aggiungersi e rimuoversi dalla lista.

Vediamo come si presenta il Pattern Observer utilizzando il Class Diagram in UML:

![[83.png]]

Esempio - Observer

Nnelle librerie Java il *Subject* e l'*Observer* sono già presenti con le classi *java.util.Observable e java.util.Observer*.

![[84.png]]

Pertanto, per non reinventare la ruota ogni volta, possiamo utilizzare queste classi per il nostro esempio.

Vediamo come si presenta il nostro Class Diagram in UML:

![[85.png]]

Creiamo la classe ConcreteObserver implementando l'interfaccia Observer che definisce il metodo `update()` con 2 parametri: il soggetto osservato Observable ed un Object utile a passare degli argomenti.

```
import java.util.Observable;
import java.util.Observer;

public class ConcreteObserver implements Observer {

    @Override
    public void update(Observable o, Object arg) {
        System.out.println("Sono " + this + ": il Subject e stato modificato!");
    }
        // In questo caso o == this
}
```

Modifichiamo la classe ConcreteSubject in modo che estenda la classe Observable. Nel metodo `setState()` occorre invocare anche il metodo `setChanged()` che esprime la volontà di notificare gli osservatori. Infatti se commentiamo questo metodo, gli osservatori non saranno notificati nonostante l'invocazione del metodo `notifyObservers()`. Ciò avviene in quanto nel metodo `notifyObservers()` è presente una semplice condizione `if (!changed)` che forza l'uscita dal metodo qualora il cambio non venga confermato.

```
import java.util.Observable;

public class ConcreteSubject extends Observable {

    private boolean state;

        @Override
    public void setState(boolean state) {
        this.state = state;
        setChanged();
        notifyObservers();
    }

        @Override
    public boolean getState() {
        return this.state;
    }

} // ! ConcreteSubject
```

Adesso vediamo il codice del metodo `notifyObservers()` della classe `java.util.Observable`:

```
public void notifyObservers(Object arg) {
    Object[] arrLocal;

    synchronized (this) {
        if (!changed)
            return;

        arrLocal = obs.toArray();
        clearChanged();
    }

    for (int i = arrLocal.length - 1; i >= 0; i--)
        ((Observer) arrLocal[i]).update(this, arg);
}
```

Creiamo il codice del Client importando `java.util.Observer` e rinominando il metodo `deleteObserver()` invece di `removeObserver()` dell'esempio precedente:

```
import java.util.Observer;

public class Client {

    public static void main(String[] args) {
        ConcreteSubject subject = new ConcreteSubject();
        Observer observer1 = new ConcreteObserver();
        Observer observer2 = new ConcreteObserver();

        //aggiungo 2 observer che saranno notificati
        subject.addObserver(observer1);
        subject.addObserver(observer2);

        //modifico lo stato
        subject.setState(true);

        //rimuovo il primo observer che non sarà + notificato
        subject.deleteObserver(observer1);

        //modifico lo stato
        subject.setState(false);

    }
} // ! Client
```

Output:

```
$JAVA_HOME/bin/java patterns.Client
Sono patterns.observer.ConcreteObserver@f62373: il Subject e' stato modificato!
Sono patterns.observer.ConcreteObserver@19189e1: il Subject e' stato modificato!
Sono patterns.observer.ConcreteObserver@f62373: il Subject e' stato modificato!
```

Torna all'indice

Visitator

Si tratta di un pattern comportamentale basato su oggetti e viene utilizzato per eseguire delle operazioni sugli elementi di una struttura. L'utilizzo di questo pattern consente di definire le operazioni di un elemento senza doverlo modificare.

Ma com'è possibile? Solitamente ogni classe definisce le proprie proprietà e le proprie operazioni nel rispetto del principio della singola responsabilità (SRP) ed usando il concetto di ereditarietà può condividere le operazioni alle classi figlie.

Ma cosa succede se ci accorgiamo a posteriori che dobbiamo introdurre una nuova operazione?

- Se le operazioni sono state definite a *livello di classe*, l'introduzione di un nuovo metodo comporterà la modifica della classe interessata, violando il principio open-closed (OCP).
- Se le operazioni sono state definite a *livello di interfaccia*, l'introduzione di un nuovo metodo comporterà la modifica di tutte le classi figlie.

Ovviamente se questa situazione si presenta frequentemente, la manutenzione del codice non sarà agevole.

Per evitare questo problema sarà possibile seguire un'altra strada, ossia disaccoppiare gli oggetti che definiscono lo stato dagli oggetti che definiscono il comportamento ed in questo modo sarà più semplice inserire nuovi metodi.
Il pattern Visitor ci consente di implementare questa separazione tra stato e comportamento e realizzare il legame tra questi oggetti tramite la definizione di 2 metodi presenti nelle due strutture.

1. Nella *prima struttura*, che definisce lo stato, è presente il metodo `accept()` che invoca il metodo `visit()`.
2. Nella *seconda struttura*, che definisce il comportamento, è presente il metodo `visit()`.

In questo modo sarà possibile aggiungere nuove operazioni semplicemente definendo nuove classi nella seconda struttura che si occuperà poi di elaborare lo stato della prima.

Vediamo la rappresentanzione UML usando il Class Diagram:

![[48.png]]

Pertanto, parlando in termini del pattern Visitor:
In base alla competenza:

1. la *prima struttura* definisce gli *Element* che detengono lo stato.
2. la *seconda struttura* definisce i *Visitor* che detengono i comportamenti.

In base all'ordine di invocazione:

1. Il Client invoca il metodo `accept()` presente nell'*Element* passandogli in ingresso un oggetto *Visitor*.
2. L'*Element* invoca il metodo `visit()` del Visitor passandogli se stesso (oggetto `this`) come parametro.
3. Il *Visitor*, disponendo della referenza all'*Element* (tramite l'oggetto `this`) accede alle proprietà dell'Element ed eseguire le operazioni.

Vediamo la rappresentanzione UML usando il Sequence Diagram:

![[49.png]]

Questo pattern utilizza la tecnica del Double Dispatch (https://dellabate.wordpress.com/2012/11/28/simple-double-e-multi-dispatch/) al fine di consentire questo scambio di messaggi tra l'Element ed il Visitor, pertanto risulta un po' complesso considerando che utilizza polimorfismo, overriding ed overloading.

Partecipanti e Struttura

Questo pattern è composto dai seguenti partecipanti:

- *Element*: definisce il metodo `accept()` che prende un Visitor come argomento.
- *ConcreteElement*: implementa un oggetto Element che prende un Visitor come argomento.
- *ObjectStructure*: contiene una collezione di Element che può essere visitata dagli oggetti Visitor.
- *Visitor*: dichiara un metodo `visit()` per ogni Element; il nome del metodo ed il parametro identificano la classe Element che ha effettuato l'invocazione.
- *ConcreteVisitor*: implementa il metodo `visit()` e definisce l'algoritmo da applicare per l'Element passato come parametro.

Vediamo come si presenta il Pattern Visitor utilizzando il Class Diagram in UML:

![[50.png]]

Conseguenze

Tale pattern presenta i seguenti vantaggi/svantaggi:

- Facilità nell'aggiungere nuovi Visitor: definendo un nuovo Visitor sarà possibile aggiungere una nuova operazione ad un Element.
- Difficoltà nell'aggiungere nuovi Element: definire un nuovo Element comporterà la modifica dell'interfaccia Visitor e di tutte le implementazioni.
- Separazione tra stato ed algoritmi: gli algoritmi di elaborazioni sono nascosti nelle classi Visitor e non vengono esposti nelle classi Element.
- Iterazione su struttura eterogenea: la classe Visitor è in grado di accedere a tipi diversi, senza la necessità che tra di essi ci sia un vincolo di parentela. In poche parole, il metodo `visit()` può definire come parametro un tipo \(X\) oppure un tipo \(Y\) senza che tra di essi ci sia alcuna relazione di parentela, diretta o indiretta.
- Accumulazione dello stato: un Visitor può accumulare delle informazioni di stato a seguito dell'attraversamento degli Element.
- Violazione dell'incapsulamento: i Visitor devono poter accedere allo stato degli Element e questo può comportare la violazione dell'incapsulamento.

Esempio - Visitor

Calcoliamo l'area e il perimetro di un rettangolo utilizzando il design pattern Visitor.

![[51.png]]

Definiamo l'interfaccia Visitor:

```
public interface Visitor {
        public void visitRettangoloArea(ElementRettangolo element);
        public void visitRettangoloPerimetro(ElementRettangolo elemento);
}
```

Implementiamo la classe VisitorRettangoloArea. Ovviamente il metodo relativo al calcolo del perimetro non dovrà essere implementato:

```
public class VisitorRettangoloArea implements Visitor {
    @Override
    public void visitRettangoloArea(ElementRettangolo element) {
        int area = element.getAltezza() * element.getLarghezza();
        System.out.println("L'area del rettangolo e': " + area);
    }
    @Override
    public void visitRettangoloPerimetro(ElementRettangolo element) {
        throw new UnsupportedOperationException("Not supported.");
    }
}
```

Adesso definiamo l'altro Visitor, VisitorRettangoloPerimetro, che si occupa di calcolare il perimentro del rettangolo:

```
public class VisitorRettangoloPerimetro implements Visitor {
    @Override
    public void visitRettangoloArea(ElementRettangolo element) {
        throw new UnsupportedOperationException("Not supported.");
    }
    @Override
    public void visitRettangoloPerimetro(ElementRettangolo element) {
        int per = element.getAltezza() + elemento.getLarghezza();
        per = per * 2;
            System.out.println("Il perimetro del rettangolo e': " + per);
    }
}
```

Adesso nella classe Element, ElementRettangolo, definiamo un comportamento diverso a seconda dell'oggetto passato come paramentro al metodo `accept()`:

```
public class ElementRettangolo {
    private int altezza;
    private int larghezza;

    public int getAltezza() {
        return this.altezza;
    }

    public void setAltezza(int altezza) {
        this.altezza = altezza;
    }

    public int getLarghezza() {
        return this.larghezza;
    }

    public void setLarghezza(int larghezza) {
        this.larghezza = larghezza;
    }

    public void accept(Visitor visitor) {
        if (visitor instanceof VisitorRettangoloArea)
            visitor.visitRettangoloArea(this);
        else if (visitor instanceof VisitorRettangoloPerimetro)
            visitor.visitRettangoloPerimetro(this);
    }
}
```

Infine nella classe Client possiamo invocare il nostro Client che si occupa di creare il rettangolo e successivamente di invocare le operazioni relative al calcolo dell'aria e del perimetro in base al tipo di Visitor che viene passato:

```
public class Client {
    public void test() {
        ElementRettangolo element = new ElementRettangolo();

        element.setAltezza(10);
        element.setLarghezza(20);

        element.accept( new VisitorRettangoloPerimetro() ); // 60
        element.accept( new VisitorRettangoloArea() ); // 200
    }
}
```

Torna all'indice

Indice

SCM

Il Software Configuration Management (SCM) è una disciplina nel campo dell'ingegneria del software che si occupa di gestire e controllare l'evoluzione di un prodotto software durante il suo ciclo di vita. L'obiettivo principale dello SCM è garantire che il software sviluppato risponda in modo efficace ai requisiti, sia correttamente testato e manutenibile nel tempo.

> *The goal is to maximize [programmer] productivity by minimizing [co-ordination] mistakes.*

Di seguito, esploreremo i concetti chiave dello SCM:

1. **Gestione delle Configurazioni:**
    - **Baseline:** Una baseline è uno stato specifico di un sistema software identificato e conservato in modo da poter essere recuperato in futuro. Può rappresentare una versione stabile e verificata del software.
    - **Controllo delle Modifiche:** Le modifiche al software vengono gestite in modo controllato. Ogni modifica deve essere documentata, valutata e implementata seguendo un processo ben definito.
2. **Controllo di Versione:**
    - **Repository:** Tutti gli artefatti software (codice sorgente, documentazione, configurazioni, ecc.) sono mantenuti in un repository centrale. I repository consentono il versionamento del software.
    - **Branching e Merging:** Consentono ai team di lavoro di sviluppare in parallelo senza interferire l'uno con l'altro. Le modifiche effettuate in rami separati possono poi essere integrate (merge) nel ramo principale.
3. **Gestione delle Modifiche:**
    - **Change Request:** Ogni modifica proposta deve essere documentata attraverso una Change Request. Questa richiesta include informazioni sulla modifica proposta, la sua motivazione e l'impatto previsto.
    - **Revisione e Approvazione:** Le modifiche sono soggette a revisione da parte del team e richiedono l'approvazione prima di essere implementate.
4. **Build e Rilascio:**
    - **Build Automation:** L'automazione del processo di creazione del software (build) assicura che ogni versione del software sia riproducibile e costruita da una configurazione nota.
    - **Deployment:** La gestione delle configurazioni include anche la pianificazione e il controllo del rilascio del software, garantendo che le versioni approvate siano distribuite in modo appropriato.
5. **Tracciabilità e Audit:**
    - **Tracciabilità:** Fornisce la capacità di seguire le relazioni tra gli artefatti software, come la relazione tra requisiti, codice sorgente e test.
    - **Auditabilità:** La possibilità di ispezionare e verificare le modifiche apportate al software e ai suoi componenti.
6. **Ambiente di Lavoro Collaborativo:**
    - **Collaborazione e Comunicazione:** Uno degli aspetti chiave dello SCM è facilitare la collaborazione e la comunicazione tra i membri del team di sviluppo. Strumenti come sistemi di tracciamento dei bug, chat, e-mail integrati possono supportare questa dimensione dello SCM.

Torna all'indice

Problemi

Esaminiamo nel dettaglio i problemi relativi al Software Configuration Management (SCM):

1. **Identification (Identificazione):**
    - **Problema:** Identificare correttamente i singoli componenti e le configurazioni può essere una sfida.
    - **Esempi di Problemi:**
        - "Questo ha funzionato ieri, cosa è successo oggi?"
        - "Abbiamo l'ultima versione?"
        - "Ho già risolto questo problema. Perché è ancora presente?"
2. **Change Tracking (Tracciamento delle Modifiche):**
    - **Problema:** Tracciare accuratamente quali modifiche sono state apportate a quali moduli e da chi, quando e per quale motivo.
    - **Esempi di Problemi:**

        - "Questo problema è stato risolto?"
        - "Chi è responsabile di questa modifica?"
        - "Questa modifica sembra ovvia, è stata già testata in precedenza?"

3. **Software Production (Produzione del Software):**

    - **Problema:** La costruzione di un programma coinvolge processi come pre-elaborazione, compilazione, collegamento, ecc.
    - **Esempi di Problemi:**
        - "Ho appena corretto questo, qualcosa non è stato compilato?"
        - "Come è stato prodotto questo binario?"
        - "Abbiamo seguito tutti i passaggi necessari nell'ordine corretto?"

4. **Concurrent Updating (Aggiornamenti Concorrenti):**

    - **Problema:** Il sistema dovrebbe offrire possibilità per modifiche concorrenti ai componenti.
    - **Esempi di Problemi:**
        - "Perché le mie modifiche sono scomparse?"
        - "Come inserisco queste modifiche nella mia versione?"
        - "Le nostre modifiche sono in conflitto tra loro?"

Questi problemi sottolineano le sfide fondamentali affrontate nel campo dello SCM. L'identificazione accurata, il tracciamento delle modifiche, la produzione del software senza errori e la gestione degli aggiornamenti concorrenti sono essenziali per garantire la stabilità e la coerenza del software durante il suo ciclo di vita. Soluzioni e pratiche efficaci nello SCM mirano a mitigare questi problemi, fornendo processi, strumenti e procedure chiare per affrontare queste sfide.

Torna all'indice

Shared data

Il problema dei dati condivisi nel contesto dello sviluppo del software si riferisce alla gestione e all'accesso concorrente ai file e alle risorse comuni da parte di più sviluppatori all'interno di un team. Questa problematica può portare a conflitti, sovrascritture indesiderate e altri errori quando più persone lavorano contemporaneamente sugli stessi file o progetti.

Sfide legate ai Dati Condivisi:

1. **Conflitti di Versione:**

    - Se più sviluppatori modificano lo stesso file contemporaneamente, potrebbe verificarsi un conflitto quando si cerca di combinare le versioni, specialmente se le modifiche riguardano le stesse linee di codice.

2. **Sovrascritture Accidentali:**

    - Due sviluppatori potrebbero sovrascrivere involontariamente le modifiche apportate dall'altro, perdendo lavoro e introducendo inconsistenze.

3. **Difficoltà nel Tracciamento delle Modifiche:**

    - Senza un sistema di gestione delle versioni adeguato, può essere difficile tracciare chi ha apportato quali modifiche e quando, complicando la risoluzione dei problemi e la gestione delle versioni.

Soluzione: L'utilizzo di Workspaces Indipendenti:

La soluzione principale per affrontare il problema dei dati condivisi è l'adozione di un modello di sviluppo basato su workspaces indipendenti per ogni sviluppatore. Questo significa che ogni membro del team lavora in un ambiente isolato, che include una copia locale del repository o del progetto.

1. **Vantaggi dei Workspaces Indipendenti:**

    - **Isolamento:** Ogni sviluppatore lavora nel proprio spazio senza interferire con il lavoro degli altri.
    - **Riduzione dei Conflitti:** La probabilità di conflitti di versione e sovrascritture indesiderate è significativamente ridotta.
    - **Tracciamento più Semplice:** È più facile tracciare le modifiche e risolvere i problemi quando ogni sviluppatore gestisce il proprio ambiente.

2. **Utilizzo di Sistemi di Gestione delle Versioni (VCS):**

    - L'adozione di un VCS, come Git o SVN, facilita la gestione delle modifiche, consentendo agli sviluppatori di lavorare in branch separati e di integrare successivamente le modifiche in modo controllato.

3. **Politiche di Lavoro Collaborativo:**

    - Definire politiche e procedure chiare per la collaborazione, come la frequenza degli aggiornamenti dal repository condiviso e le modalità di integrazione delle modifiche, contribuisce a mantenere un flusso di lavoro collaborativo ordinato.

L'utilizzo di workspaces indipendenti combinato con pratiche di sviluppo basate su VCS è una strategia efficace per affrontare il problema dei dati condivisi, consentendo a più sviluppatori di contribuire al progetto senza compromettere l'integrità del codice sorgente e delle risorse condivise.

Cosa accade quando viene introdotta una nuova modifica nel repository/file server? Si è costretti ad introdurla manualmente su ogni workspace privato, con conseguente rischio di incompatibilità nel caso quella determinata copia sia fortemente personalizzata per quel determinato cliente; questo problema prende il nome di **doppia manutenzione**.

Torna all'indice

Double maintenance

Il problema della Doppia Manutenzione si riferisce alla necessità di apportare modifiche o correzioni su più versioni di un software quando viene rilasciata una nuova versione o variante. Questo problema si manifesta soprattutto in ambienti in cui sono supportate versioni multiple di un'applicazione o in situazioni in cui il software è personalizzato per clienti diversi.

Sfide della Doppia Manutenzione:

1. **Sviluppo e Manutenzione di Versioni Diverse:**
    - Quando è necessario mantenere versioni diverse di un'applicazione, ad esempio una versione per un cliente specifico e una generica per il pubblico, le modifiche apportate alla versione generica potrebbero richiedere sforzi duplicati per adattarle alla versione personalizzata.
2. **Rischio di Errori e Incongruenze:**
    - La gestione manuale delle modifiche su più versioni aumenta il rischio di errori, omissioni e incongruenze tra le varie implementazioni.

Soluzione: Repository "Copy-Write":

L'utilizzo di un repository centrale condiviso e sincronizzato tra i vari workspaces privati permette di ottenere facilmente le nuove modifiche. I passi da eseguire sono i seguenti:

1. *Copy* del contenuto della repo in uno spazio di lavoro privato
2. Introduzione della modifica nel proprio workspace
3. *Write* dei contenuti modificati nel repository centrale
4. Notifica agli altri workspaces che possono scaricare le nuove features.

L'adozione di un sistema di questo tipo non permette però di risolvere i problemi legati ad una eventuale modifica concorrente agli stessi files: cosa succede se da un lato viene introdotto un bug-fix nel workspace privato A e contemporaneamente il workspace B scrive nel repository lo stesso file (senza il bug-fix perchè basato su una copia più vecchia)?

Torna all'indice

Simultaneous update

Il problema degli Aggiornamenti Simultanei si verifica quando due o più sviluppatori lavorano contemporaneamente su parti diverse di un sistema software e successivamente cercano di combinare i loro cambiamenti. Questa situazione può portare a conflitti e sovrascritture non desiderate, creando una serie di sfide durante la gestione delle modifiche concorrenti.

Sfide degli Aggiornamenti Simultanei:

1. **Conflitti di Modifica:**
    - Due o più sviluppatori apportano modifiche alla stessa sezione di codice o al medesimo file contemporaneamente. Quando si tenta di unire o integrare questi cambiamenti, possono verificarsi conflitti che richiedono una risoluzione manuale.
2. **Perdita di Modifiche:**
    - Se più sviluppatori modificano lo stesso file o componente e sovrascrivono le modifiche l'uno dell'altro senza consapevolezza, si può verificare la perdita di alcune modifiche.
3. **Difficoltà nella Fusione:**
    - La fusione di rami di sviluppo paralleli può diventare complicata a causa delle sovrapposizioni di modifiche. È possibile che siano necessarie attività manuali intensive per risolvere i conflitti.

Soluzione: Utilizzo del Versioning (Version Control System - VCS):

La principale soluzione per gestire gli aggiornamenti simultanei è l'implementazione di un sistema di controllo versione (VCS). Un VCS consente di tenere traccia delle modifiche apportate al codice sorgente nel tempo, fornendo un meccanismo per gestire, combinare e monitorare le modifiche effettuate da più sviluppatori contemporaneamente.

1. **Caratteristiche del Versioning System:**
    - **Storia delle Modifiche:** Registra la storia di tutte le modifiche apportate al codice, consentendo di visualizzare, annullare o combinare modifiche precedenti.
    - **Branching e Merging:** Permette agli sviluppatori di lavorare su rami (branch) separati e di successivamente unire (merge) le loro modifiche. Ciò facilita la gestione di sviluppi paralleli.
    - **Risoluzione dei Conflitti:** Fornisce strumenti per risolvere i conflitti generati durante la fusione di modifiche concorrenti.
2. **Vantaggi dell'Utilizzo del Versioning System:**

- **Controllo su Modifiche:** Consente agli sviluppatori di lavorare in modo indipendente e tenere traccia delle modifiche.
- **Collaborazione Efficiente:** Facilita la collaborazione tra sviluppatori consentendo loro di integrare e condividere le modifiche senza perdere il lavoro degli altri.
- **Gestione delle Versioni:** Offre un sistema strutturato per gestire diverse versioni del software in modo ordinato.
- **Reversibilità delle Modifiche:** Permette di tornare a versioni precedenti del software in caso di necessità.

I passaggi da effettuare sono i seguenti:

1. *Copy* del contenuto della repository nel workspace privato
2. Introduzione delle modifiche
3. *Add* della nuova versione nel repository, contenente le modifiche introdotte nel privato.

L'utilizzo di un sistema di controllo versione, come Git, SVN o Mercurial, è considerato una best practice nell'ambito dello sviluppo del software. Questi strumenti forniscono un'infrastruttura robusta per gestire gli aggiornamenti simultanei, riducendo i rischi di conflitti e semplificando la gestione del codice sorgente in un ambiente collaborativo.

![[88.png]]

Aver introdotto un sistema di versionamento permette quindi di associare ogni nuova modifica ad una determinata versione della repository; questo però non vieta di aggiungere una versione dei files modificati che si basava su una copia obsoleta, mancante delle nuove features introdotte dall'ultima caricata (sovrascrittura). Le moderne soluzioni di SCM permettono di risolvere anche questo problema attraverso tecniche che esploreremo nei paragrafi seguenti.

Torna all'indice

Model

Il "Model di Babich" si riferisce a un modello proposto da Boris Babich per affrontare problematiche specifiche relative allo sviluppo del software, in particolare quelle legate al problema degli "Aggiornamenti Simultanei". Il modello di Babich si basa su alcuni principi chiave per mitigare le sfide associate all'aggiornamento simultaneo del software.

Principio Fondamentale L'elemento centrale del modello Babich è il principio che i componenti del sistema dovrebbero essere immutabili. In altre parole, una volta creato un componente, non dovrebbe essere modificato, ma piuttosto sostituito con una nuova versione.

![[89.png]]

Problemi Affrontati dal Modello di Babich:

1. **Shared Data (Dati Condivisi):**
   - Affronta il problema dei dati condivisi, dove più parti del sistema accedono e modificano gli stessi dati contemporaneamente.
2. **Double Maintenance (Doppia Manutenzione):**
   - Risolve il problema della doppia manutenzione, che si verifica quando è necessario modificare la stessa porzione di codice in più parti del sistema.
3. **Simultaneous Update (Aggiornamenti Simultanei):**
   - Affronta il problema degli aggiornamenti simultanei, ovvero quando più sviluppatori apportano modifiche a componenti diversi allo stesso tempo.

Come Funziona:

1. **Componenti Immobili:**
   - I componenti del sistema, una volta creati, diventano immutabili. Se è necessario apportare modifiche, viene creata una nuova versione del componente senza alterare l'esistente.
2. **Evita Condivisione Diretta di Dati:**
   - Evita situazioni in cui più parti del sistema condividono direttamente lo stesso spazio di memoria dei dati. Invece, le informazioni sono passate attraverso interfacce ben definite.
3. **Versioning per Componenti:**
   - Quando è necessario apportare modifiche, viene creata una nuova versione del componente, consentendo di mantenere le versioni esistenti intatte.
4. **Prevenzione della Doppia Manutenzione:**
   - Riduce la necessità di effettuare modifiche simili in più parti del sistema, poiché i componenti immutabili vengono sostituiti con nuove versioni.

Vantaggi del Modello di Babich:

- **Riduzione dei Conflitti:** La natura immutabile dei componenti riduce i conflitti dovuti agli aggiornamenti simultanei.
- **Miglior Controllo delle Modifiche:** Le modifiche vengono gestite attraverso la creazione di nuove versioni, garantendo un migliore controllo sulle modifiche apportate al sistema.
- **Minimizzazione della Doppia Manutenzione:** Riduce la necessità di effettuare modifiche simili in più parti del codice, contribuendo a evitare la doppia manutenzione.
- **Miglior Gestione del Versioning:** L'approccio versioning facilita la gestione delle diverse versioni del software.

Il Modello di Babich si basa su concetti chiave di immutabilità e versioning, offrendo una strategia per affrontare problemi specifici legati agli aggiornamenti simultanei e alla condivisione di dati nel contesto dello sviluppo del software.

![[90.png]]

Torna all'indice

Parallel work

La gestione del lavoro parallelo nel contesto del Software Configuration Management (SCM) coinvolge due strategie principali: la strategia pessimistica (locking) e la strategia ottimistica (copy-merge).

Pessimistic (Locking)

Descrizione:

- La strategia pessimistica si basa sull'idea di "bloccare" un componente quando un utente inizia a modificarlo.
- Quando un utente intende apportare modifiche a un componente, richiede un "lock" su quel componente.
- Il lock impedisce ad altri utenti di apportare modifiche al componente bloccato fino a quando il detentore del lock non rilascia il blocco.

Pro e Contro:

- **Pro:** Evita conflitti diretti, poiché solo una persona alla volta può modificare un componente specifico.
- **Contro:** Potenziali ritardi e congestioni se molte persone richiedono lock su componenti molto utilizzati. Inoltre, un utente potrebbe trattenere un lock per un periodo prolungato, causando attese per altri sviluppatori.

![[91.png]]

Ottimistica (Copy-Merge)

Descrizione:

- La strategia ottimistica prevede che più utenti possano apportare modifiche contemporaneamente a componenti diversi.
- Le modifiche vengono apportate in modo indipendente senza alcun blocco preventivo.
- Quando è necessario integrare le modifiche, il sistema tenta di "unire" o "mergiare" automaticamente le modifiche provenienti da diversi utenti.

Pro e Contro:

- **Pro:** Favorisce il lavoro parallelo, poiché gli utenti possono modificare indipendentemente i componenti.
- **Contro:** Può causare conflitti che devono essere risolti durante il merge. La corretta risoluzione dei conflitti richiede una gestione accurata del versioning e un processo di merge ben progettato.

Scelta tra Pessimistic e Optimistic

1. **Pessimistic (Locking):**
   - **Scelta:** Utile quando i componenti sono complessi e richiedono modifiche estese, riducendo così la probabilità di conflitti.
   - **Considerazioni:** Può rallentare il processo se molti utenti richiedono lock su componenti frequentemente utilizzati.
2. **Ottimistica (Copy-Merge):**
   - **Scelta:** Adatto quando ci sono molte modifiche indipendenti che possono essere integrate successivamente.
   - **Considerazioni:** Richiede una gestione accurata dei conflitti durante il merge e una comunicazione efficace tra gli sviluppatori.

La scelta tra le due strategie dipende dalla natura del progetto, dalla frequenza delle modifiche e dalla complessità dei componenti. In alcuni casi, potrebbe essere utile adottare un approccio misto basato sulla natura specifica dei componenti e delle attività di sviluppo.

Torna all'indice

Copy/Merge Work Model

Il modello di lavoro Copy/Merge è un approccio utilizzato nel contesto del controllo di versione e del Software Configuration Management (SCM). Esso coinvolge il concetto di "branching" e "merging".

1. **Branching:**
   - Quando un membro del team deve apportare modifiche significative o sperimentali a una parte del codice, crea un "branch" separato.
   - Un branch è essenzialmente una copia indipendente dell'intero repository o di una parte significativa di esso.
   - Il branch consente di lavorare indipendentemente sulle modifiche senza influenzare direttamente il codice principale (trunk).
2. **Lavoro Indipendente:**
   - Gli sviluppatori lavorano indipendentemente sul proprio branch, apportando le modifiche necessarie per implementare nuove funzionalità o risolvere problemi specifici.
3. **Merging:**

- Dopo aver completato il lavoro nel proprio branch, gli sviluppatori devono "mergiare" le modifiche nel codice principale (trunk) o in altri branch, se necessario.
- Il merge è il processo di integrazione delle modifiche apportate in un branch con le modifiche apportate in un altro branch o nel codice principale.

Vantaggi del Copy/Merge Model:

- **Isolamento delle Modifiche:** I branch consentono agli sviluppatori di lavorare in isolamento, riducendo il rischio di conflitti diretti con il codice principale.
- **Sperimentazione:** I branch possono essere utilizzati per sperimentare nuove funzionalità o approcci senza influenzare il codice principale.
- **Gestione delle Release:** È possibile mantenere branch separati per gestire le diverse versioni di un'applicazione o per preparare una release stabile.

Torna all'indice

Importanza del Branching

1. **Sviluppo Isolato:**
   - Consentire a diversi membri del team di lavorare su funzionalità o correzioni di bug senza interferire l'uno con l'altro.
2. **Sperimentazione:**
   - Offrire uno spazio sicuro per sperimentare nuove idee o approcci senza influire sullo sviluppo principale.
3. **Rilasci Stabili:**
   - Consentire la creazione di branch dedicati per rilasci stabili, garantendo che eventuali correzioni di bug siano integrate solo quando sono pronte.
4. **Concorrenza:**
   - Supportare lo sviluppo parallelo consentendo a diversi team o sviluppatori di lavorare su funzionalità o aree di codice diverse.
5. **Gestione di Feature Set:**
   - Agevolare lo sviluppo di nuove funzionalità senza influire sulle funzionalità esistenti finché non sono pronte per l'integrazione.
6. **Collaborazione Distribuita:**
   - Facilitare la collaborazione tra team distribuiti, in quanto ciascun team può lavorare su un branch separato e successivamente integrare le modifiche.

In sintesi, il branching nel Copy/Merge Work Model è fondamentale per gestire il lavoro in modo isolato, sperimentare, mantenere rilasci stabili e facilitare lo sviluppo parallelo. La capacità di integrare le modifiche in modo controllato tramite il merging contribuisce a mantenere la coerenza del codice e a gestire efficacemente lo sviluppo del software.

Torna all'indice

3-Way merging

Il 3-way merging è un processo utilizzato nei sistemi di controllo versione per integrare le modifiche apportate in due branch o versioni divergenti di un file o di un insieme di file. Questo approccio coinvolge tre "versioni" o "snapshot" del codice sorgente:

1. **Versione "Base" (Base Version):**
   - Questa è la versione comune del file o dei file su cui sono stati creati i branch separati. Può anche essere chiamata "ancestrale" o "originale".
2. **Versione "Source" (Source Version):**
   - Questa è la versione che si trova su un branch. Rappresenta le modifiche apportate su quel branch specifico.
3. **Versione "Target" (Target Version):**
   - Questa è la versione presente in un altro branch. Rappresenta le modifiche apportate su un branch diverso.

Processo di 3-Way Merging:

1. **Identificazione delle Differenze:**
   - Il sistema di controllo versione identifica le differenze tra la versione base, la versione di origine e la versione di destinazione.
2. **Applicazione delle Modifiche:**
   - Il sistema applica le modifiche dalla versione di origine alla versione base e, contemporaneamente, applica le modifiche dalla versione di destinazione alla versione base.
3. **Risoluzione dei Conflitti:**
   - Se ci sono conflitti durante l'applicazione delle modifiche, il sistema richiede un intervento umano per risolvere manualmente tali conflitti.
4. **Creazione della Nuova Versione:**
   - Una volta risolti i conflitti, il sistema crea una nuova versione del file che include tutte le modifiche provenienti dai due branch.

Vantaggi del 3-Way Merging:

- **Riconoscimento Automatico delle Modifiche Comuni:**

- Il processo è progettato per riconoscere automaticamente le modifiche comuni apportate in entrambi i branch, riducendo la necessità di intervento umano quando le modifiche sono compatibili.

- **Gestione dei Conflitti:**
  - Fornisce un meccanismo strutturato per gestire i conflitti, che possono verificarsi quando le modifiche apportate in due branch interessano la stessa porzione di codice.

- **Efficienza nel Merge:**
  - Riduce il rischio di sovrascrivere le modifiche apportate in uno dei branch durante il merge.

Il 3-way merging è particolarmente utile quando si lavora con sviluppo parallelo su branch separati. Consente una maggiore automazione nel processo di integrazione, garantendo al contempo che le modifiche confliggenti siano attentamente gestite per preservare l'integrità del codice sorgente.

![[94.png]]

Torna all'indice

Split-Combine Work Model

Lo *split-combine* è un modello di lavoro parallelo utilizzato nel contesto del Software Configuration Management (SCM). Questo modello è particolarmente utile in situazioni in cui si lavora su rami separati di uno stesso progetto e si desidera riunire successivamente le modifiche apportate in modo efficiente. Vediamo una spiegazione dettagliata del processo *split-combine*:

1. **Split (Divisione):**
   - Inizia con una versione di base (main branch) del progetto. Quando un team inizia a lavorare su un nuovo aspetto o una nuova funzionalità, si crea un nuovo ramo (*branch*) separato dal ramo principale. Questo nuovo ramo è ora un ambiente isolato in cui il team può apportare modifiche senza influenzare il lavoro sul ramo principale.
2. **Development on Separate Branches (Sviluppo su Rami Separati):**
   - I diversi team o sviluppatori lavorano contemporaneamente su rami separati. Ogni ramo può includere modifiche, aggiunte o correzioni specifiche.
3. **Combine (Unione):**
   - Quando le modifiche sono state completate e testate con successo sui rami separati, è il momento di riunire (combining) le modifiche nel ramo principale o in un altro ramo di integrazione. Questo processo è noto come *combine*.
4. **Conflict Resolution (Risoluzione dei Conflitti):**
   - Durante il processo di combinazione (*combine*), potrebbero verificarsi conflitti. I conflitti possono sorgere quando più rami hanno apportato modifiche alla stessa parte del codice. È necessario risolvere manualmente questi conflitti, decidendo quali modifiche includere nella versione finale.
5. **Testing and Integration (Test e Integrazione):**
   - Dopo la risoluzione dei conflitti, il codice risultante deve essere testato per garantire che funzioni correttamente. Successivamente, la versione combinata può essere integrata nel flusso principale del progetto.
6. **Documentation (Documentazione):**
   - È essenziale documentare accuratamente le modifiche apportate in modo che altri membri del team possano comprendere il lavoro svolto e le motivazioni dietro le decisioni di progettazione.

Vantaggi del Modello Split-Combine:

- **Parallel Development:** Permette lo sviluppo parallelo su rami separati.
- **Isolamento delle Modifiche:** Ogni ramo è isolato, consentendo ai team di lavorare indipendentemente.
- **Controllo dei Conflitti:** Il processo di combinazione richiede la risoluzione dei conflitti in modo controllato.
- **Gestione delle Versioni:** Fornisce un modo organizzato per gestire le diverse versioni del codice.

Questo modello è utile in scenari di sviluppo collaborativo, in cui diversi team o sviluppatori lavorano contemporaneamente su parti diverse di un progetto.

![[95.png]]

Torna all'indice

CM

La gestione della configurazione (Configuration management) è un processo di ingegneria dei sistemi per stabilire e mantenere la coerenza delle prestazioni, degli attributi funzionali e fisici di un prodotto con i suoi requisiti, la progettazione e le informazioni operative per tutta la sua vita Il CM, se applicato durante il ciclo di vita di un sistema, fornisce visibilità e controllo delle sue prestazioni, attributi funzionali e fisici. Il CM verifica che un sistema funzioni come previsto e sia identificato e documentato in modo sufficientemente dettagliato per supportare il ciclo di vita previsto. Il processo CM facilita la gestione ordinata delle informazioni di sistema e delle modifiche del sistema.

Gestione della configurazione tradizionale

- Identificazione: la selezione e la gestione degli artefatti importanti per la creazione del prodotto.
- Controllo delle modifiche: il controllo delle modifiche alla configurazione di un prodotto e ai suoi artefatti.
- Status accounting: registrazione e reporting dell'implementazione delle modifiche a un prodotto e ai suoi artefatti.
- Audit: la convalida della configurazione di un prodotto per la conformità alla sua definizione.

Identificazione

L'identificazione nella gestione della configurazione è il processo di selezionare, designare e descrivere gli elementi di configurazione. Gli elementi di configurazione possono essere singoli artefatti, configurazioni più complesse o prodotti costruiti. La gestione della configurazione richiede una corretta identificazione per garantire la tracciabilità e la gestione ordinata delle informazioni del sistema e delle modifiche del sistema nel tempo. Questo processo coinvolge nomi univoci, metadati e archiviazione strutturata per consentire una gestione efficace degli elementi di configurazione. La tracciabilità, sia orizzontale che verticale, è fondamentale per comprendere le relazioni e le dipendenze tra gli elementi di configurazione.

Status accounting

Lo status accounting nella gestione della configurazione è il processo di registrazione e segnalazione delle informazioni necessarie per gestire efficacemente un progetto. Questo processo prevede la risposta a domande specifiche, che possono variare in base alle esigenze delle persone e alle diverse situazioni. L'interrogazione sulla CMDB può essere standardizzata o ad hoc. La registrazione delle informazioni richiede talvolta aggiornamenti allo schema della CMDB e deve garantire che i dati siano acquisiti correttamente e al momento opportuno. L'obiettivo è rendere le informazioni disponibili e utili a tutti i membri del team per mantenere un costante stato di aggiornamento sul progetto.

Controllo delle modifiche

Il controllo delle modifiche nella gestione della configurazione è il processo che gestisce le proposte di modifica, dalla valutazione alla loro attuazione. Le richieste di modifica (CR) possono riguardare vari elementi, come report di problemi, deroge, requisiti, ecc. Ogni CR deve contenere tutte le informazioni necessarie sulla modifica proposta. Il processo di modifica comprende la fase di filtraggio, l'analisi d'impatto e la presentazione alla Change Control Board (CCB) per l'approvazione o il rifiuto. L'analisi d'impatto valuta il tempo, le risorse, i costi e le conseguenze delle modifiche proposte. La CCB, presieduta da un presidente con poteri decisivi, si riunisce regolarmente per prendere decisioni su CR presentati. I risultati possono essere approvati, respinti, rinviati o inoltrati.

Audit

L'audit nella gestione della configurazione è un'indagine indipendente su un elemento di configurazione modificato per verificare la sua conformità a specifiche, standard, accordi contrattuali e altri criteri. L'obiettivo principale è assicurare che il prodotto (CI - Configuration Item) corrisponda alla descrizione nelle specifiche e nella documentazione, verificando anche che il lavoro sia stato eseguito correttamente, rispettando gli standard e le linee guida di sviluppo. Gli audit possono essere effettuati in vari momenti e con diversi gradi di formalità, fungendo da "cancelli di qualità" prima che un CI venga accettato nella baseline.

Torna all'indice

Business value

Come il SCM si traduce in valore per l'azienda?

- Sviluppo più rapido
- Migliore qualità
- Maggiore affidabilità

Sette requisiti critici:

1. **Sicurezza:** Garantire che le informazioni e i dati siano protetti contro accessi non autorizzati o modifiche indesiderate.
2. **Stabilità:** Assicurare che il sistema sia coerente e funzionante senza interruzioni indesiderate o errori critici.
3. **Controllo:** Mantenere un controllo completo sulle modifiche, garantendo che solo le modifiche autorizzate siano implementate.
4. **Auditabilità:** Fornire una traccia chiara e completa di tutte le attività di gestione della configurazione per scopi di audit e conformità.
5. **Riproducibilità:** Essere in grado di riprodurre specifiche configurazioni o versioni di un sistema in modo coerente e affidabile.
6. **Tracciabilità:** Avere la capacità di tracciare l'origine, le modifiche e l'impatto di ciascun componente della configurazione.
7. **Scalabilità:** Garantire che il sistema sia in grado di gestire crescenti dimensioni e complessità, adattandosi alle esigenze in continua evoluzione del progetto o dell'organizzazione.

Una corretta implementazione del Software Configuration Management (SCM) porta a benefici tangibili per l'azienda, come sviluppo più rapido, migliore qualità e maggiore affidabilità. I requisiti critici indicano gli aspetti fondamentali che devono essere affrontati per ottenere tali benefici. La sicurezza, la stabilità e il controllo sono elementi chiave per garantire un ambiente sicuro e gestibile. L'auditabilità, la riproducibilità e la tracciabilità contribuiscono a una gestione più trasparente e controllata delle configurazioni. Infine, la scalabilità è cruciale per adattarsi alle esigenze mutevoli del progetto o dell'organizzazione nel tempo.

Torna all'indice