La classe astratta o interfaccia "Visitor" dichiara un metodo Visit per ogni classe di elementi concreti, il "ConcreteVisitor" definisce tutti i metodi visit

.NET ASSEMBLY - METADATI

Assembly: unità minima per la distribuzione ed il versioning. Composto da uno o più□ file. (module) E' strutturato in

- manifest (metadati che descrivono l'assembly stesso),in
- Type Metadata(metadati che descrivono tutti i tipi contenuti nell'assembly)
- codice IL (codice ottenuto da un qualsiasi linguaggio di programmazione)

- Risorse (immagini, icone...) Metadati: Descrizione dell'assembly, nome, versione, lista file, riferimenti ad altri assembly, permessi. Descrizione dei contenuti dell' assembly, nome, visibilit, interfacce, classe base, metodi ecc. I metadati sono utilizzati da compilatori e vari tool di analisi del codice

CLS - .NET

Il Common Language Specification definisce le regole che i programmi devono seguire per interagire con altre infrastutture, il cls è un sottoinsieme del cts (Common Type System). Regole per gli identificatori, regole per denominazione proprietà ed eventi, regole per costruttori degli oggetti, regole di overload più restrittive, ammesse interfacce multiple con metodi con lo stesso nome, non ammessi puntatori unmanaged

TYPE EVENT

Un evento incapsula un delegato -> Deve essere dichiarato un tipo di delegato prima di poter dichiarare un evento. Per convenzione, i delegati dell'evento hanno due argomenti, la sorgente che genera l'evento e i dati per l'evento

FLYWEIGHT PATTERN

Descrive come condividere oggetti leggeri. Il flyweight un oggetto condiviso che puo' essere usato da piu' clienti simultaneamente. I clienti non devono istanziare direttamente i flyweight ma utilizzare una flyweightfatory -> GetFlyweight (key) . Benchè condiviso non deve essere distinguibile da un oggetto non condiviso. Il Flyweight memorizza lo stato intrinseco, cioè non dipendente dal contesto di utilizzo, percio' condivisibile tra tutti i clienti. Lo stato estrinseco, cioè quello contestualizzabile e non condivisibile, è memorizzato nel client (oppure da questi calcolato) e viene passato al flyweigth nell'atto di invocarne una sua operazione.

CLASSE ASTRATTA

La classe astratta può descrivere una funzionalità complessa, comune ad oggetti omogenei. Può ereditare da o+ interfacce o 0+ classi. Non può essere instanziata, può contenere uno stato, può contenere attributi e metodi statici. Può essere implementata parzialmente, completamente o per niente, le classi che la estendono devono implementare tutte le funzionalitˆ non implementate e possono fornire una realizzazione alternativa. Può gestire la creazione delle istanze delle sottoclassi. La creazione pu˜ essere fatta da costruttori o da factory.

PATTERN DELEGA

Pattern Strategy, che serve a definire un insieme di algoritmi da incapsulare in gerarchia, usa la delega facendo compiere determinate operazioni alle sue sottoclassi

Pattern State, che permette di simulare l'ereditarietà multipla, le sottoclassi implementano il comportamento dello stato della classe base. Pattern Adapter utilizza sempre delega per adattare i metodi di una classe a

quelli di un'altra (cioè invoca i metodi di un'istanza di quella da adottare , modifica il risultato in modo da ottenere omogeneo a quello dell'altra classe)

RISCHI TECNOLOGICI

Rischi tecnologici si riferiscono alla tecnologia corretta, si deve analizzare se si □ in grado di aggregare i vari componenti del progetto (GUI; DB) e quali saranno i possibili futuri sviluppi della tecnologia

DEFINIZIONE DESIGN PATTERN

Un pattern ti permette di risolvere un problema che ti capiterˆ pi□ e pi□ volte, basandoti sempre sulla stessa strategia di risoluzione. Un Design Pattern cattura lÕesperienza acquisita nellÕaffrontare uno specifico problema e permette di riutilizzare questa esperienza in casi simili

TIPI VALORE E TIPI RIFERIMENTO

(leggila tutta) per quanto riguarda il CTS Diversamente da alcuni linguaggi di programmazione, in C# sono disponibili due varietà di tipi di dati: valore e riferimento. È importante stabilire se le prestazioni dell'applicazione sono importanti o se invece si ritiene importante la modalità di gestione di dati e memoria in C#. Quando una variabile viene dichiarata tramite uno dei tipi di dati incorporati di base o una struttura definita dall'utente, rappresenta un tipo di valore. Un'eccezione è rappresentata dal tipo di dati string, che è un tipo di riferimento.

```
int x =42;
```

Un tipo di valore archivia il proprio contenuto in memoria allocata sullo stack. In questo caso, il valore 42 viene ad esempio archiviato in un'area di memoria chiamata stack. Quando la variabile x esce dall'ambito di validità poiché è stata completata l'esecuzione del metodo in cui è stata definita, il valore viene eliminato dallo stack. L'utilizzo dello stack è efficace, ma la durata limitata dei tipi di valore li rende meno adatti per la condivisione di dati tra classi diverse. Al contrario, un tipo di riferimento, come un'istanza di una classe o una matrice, viene allocato in un'area diversa di memoria denominata heap. Nell'esempio riportato di seguito lo spazio necessario per i dieci valori integer che costituiscono la matrice viene allocato nell'heap.

```
int[] numbers = new int[10];
```

Questo tipo di memoria non viene restituito all'heap al completamento di un metodo, ma viene recuperata soltanto quando il sistema di Garbage Collection di C# stabilisce che non è più necessaria. La dichiarazione dei tipi di riferimento comporta un sovraccarico maggiore, ma questi tipi presentano il vantaggio di essere accessibili da altre classi. BOXING E UNBOXING Boxing è il nome assegnato al processo mediante il quale un tipo di valore viene convertito in un tipo di riferimento. Quando si esegue il boxing di una variabile, viene creata una variabile di riferimento che punta a una nuova copia nell'heap. La variabile di riferimento è un

oggetto e, di conseguenza, può utilizzare tutti i metodi che vengono ereditati da ogni oggetto, ad esempio ToString(). Questa operazione è descritta nel codice riportato di seguito:

```
int i = 67;                                // i is a value type
object o = i;                              // i is boxed
System.Console.WriteLine(i.ToString());  // i is boxed
```

Il processo di unboxing si verifica quando vengono utilizzate classi progettate per l'utilizzo con oggetti: ad esempio mediante una classe ArrayList per archiviare valori integer. Quando un valore integer viene archiviato in una classe ArrayList, viene eseguito il processo di boxing. Quando un valore integer viene recuperato, è necessario eseguirne l'unboxing.

```
System.Collections.ArrayList list =
    new System.Collections.ArrayList();  // list is a reference type
int n = 67;                              // n is a value type
list.Add(n);                             // n is boxed
n = (int)list[0];                        // list[0] is unboxed
```

MODIFICATORI DI VISIBILITA'

- *public* Il tipo o il membro è accessibile da altro codice nello stesso assembly o in un altro assembly che vi fa riferimento.
- *private* Il tipo o il membro è accessibile solo dal codice nella stessa classe o struttura.
- *protected* Il tipo o il membro è accessibile solo dal codice nella stessa classe o struttura o in una classe derivata dalla prima.
- *internal* Il tipo o il membro è accessibile dal codice nello stesso assembly ma non da un altro assembly.
- *protected internal* Il tipo o il membro è accessibile dal codice nello stesso assembly in cui è dichiarato o da una classe derivata in un altro assembly. L'accesso da un altro assembly deve avvenire all'interno di una dichiarazione di classe che deriva dalla classe in cui l'elemento interno protetto viene dichiarato e tramite un'istanza del tipo di classe derivata.

OPERAZIONI SU EVENTO

Un evento per esistere deve incapsulare un delegato. Al di fuori della classe in cui viene dichiarato, l'evento viene visto come un delegato con permessi molto limitati e le operazioni che si possono fare su di lui sono l'agganciarsi, ovvero aggiungere un nuovo delegato o sganciarsi. Per l'aggiunta o l'eliminazione del delegato si usa lista += delegato o lista -= delegato

FRAGILITA RIGIDITA E VISCOSITA

Per fragilità del software si definisce il rischio di far collassare l'intera struttura modificando una piccola parte del progetto, oppure che una singola modifica comporti effetti non considerati nè attesi su altre parti del sistema. I cambiamenti infatti causano comportamenti inattesi su più parti del sistema, spesso addirittura in aree che non hanno relazioni concettuali con quella modificata. Per rigidità si intende invece la necessità di effettuare modifiche su più parti del sistema quando si desidera solo modificarne una. La viscosità invece indica la difficoltà di risolvere un problema senza crearne di altri.

ATTORI

per attore si intende un'entità esterna al sistema che fa a questo una richiesta. E' presente nei casi d'uso

DIAGRAMMA DI SEQUENZA

I diagrammi di sequenza appartengono al modello dinamico ovvero quel modello che desctive il comportamento del sistema durante il suo funzionamento. I diagrammi di squenza evidenziano lo scambio di messaggi e le interazoni tra gli oggetti del sistema e l'ordine in cui i messaggi vengono scambiati tra gli oggetti, ovvero la sequenza delle invocazioni delle operazioni. L' invio di un messaggio o di una richiesta di servizio da un oggetto mittente (cliente) ad un oggetti destinatario (fornitore) corrisponde all'invocazione di un'operazione della classe del destinatario o di una sua superclasse. Ogni messaggio comprende il nome dell'oeperazione invocata, gli eventuali parametri attuali e un eventuale valore restituito.

MODELLO EVOLUTIVO

Tutti i modelli evolutivi propongono un ciclo di sviluppo in cui un prototipo iniziale evolve gradualmente verso il prodotto finito attraverso un certo numero di iterazioni. Il vantaggio fondamentale è che ad ogni iterazione è possibile confrontarsi con gli utenti per quanto riguarda le specifiche e le funzionalità (raffinamento analisi) e rivedere le scelte di progetto (raffinamento del design)

PRE E POST CONDIZIONI

le precondizioni devono essere identiche o meno stringenti, le post condizioni devono essere identiche o più stringenti, gli invarianti di classe devono essere identici o più stringenti.

EVENT-HANDLER

L'EventHandler è il metodo che viene lanciato alla ricezione della notifica. Quando il sender emette la notifica, il delegato lo collega con receiver/handler. Anche l'EventHandler è un delegato

COS'E' UN REQUISITO

Un requisito è la descrizione di un comportamento del sistema (funzionale) o di un vincolo (nn funzionale)sul comportamento del sistema e sul suo sviluppo

EREDITARIETA'

Per rigidità si intende la difficoltà nel cambiare un programma, in quanto ogni cambiamento risulta difficile e costoso in termini di tempo O meglio Per rigidità si intende invece la necessità di effettuare modifiche su più parti del sistema quando si desidera solo modificarne una.

GARBAGE COLLECTOR

Gestisce il ciclo di vita di tutti gli oggetti .NET, distruggendoli quando nn sono più referenziati. A differenza del GC di COM, NON si basa su Reference Counting e ciò lo rende più veloce nell'allocazione, permette di avere riferimenti circolari, MA FA PERDERE LA DISTRUZIONE DETERMINISTICA Il funzionamento è questo: quando il CLR(COMMON LANGUAGE RUNTIME) deve eseguire una newobj, calcola la dimensione in byte dell'oggetto e aggiunge a questo due campi da 32 bit(rispettivamente un puntatore alla tabella dei metodi e un campo SyncBlockIndex), quindi controlla che ci sia spazio sufficiente a partire da NextObjPtr e in caso di spazio insufficiente fa partire il Garbage Collection. Questo verifica se nell'heap esistono oggetti non più utilizzati dall'applicazione basandosi sull'insieme delle sue radici, ovvero dei puntatori che o contengono l'indirizzo di un oggetto di tipo riferimento oppure valgono NULL. Tali radici sono variabili globali e field statici di tipo riferimento. Gli oggetti "vivi" sono quelli raggungibili direttamente o indirettamente dalle radici, mentre gli oggetti garbage sono quelli NON raggiungibili direttamente o indirettamente dalle radici. Quando il GC parte, ipotizza che tutti gli oggetti siano garbage, quindi scandisce le radici e per ognuna marca l'oggetto referenziato e tutti gli oggetti da questo raggiungibili. se nella scansione incontra un oggetto già marcato lo salta. Una volta terminata la scansione delle radici, tutti gli oggetti NON marcati sono non raggiungibili e quindi garbage. Prima di rilasciare la memoria allocata per l'oggetto , il GC invoca il metodo speciale Finalize, durante questo occorre NON accedere ad oggetti esterni IMPORTANTE sapere che il GC compatta la memoria ancora in uso MODIFICANDO tutti i riferimenti agli oggetti spostati. La FINALIZATION è responsabilità del programmatore e NON del GC. il metodo Finalize dovrebbe essere utilizzato SOLO PER RILASCIARE RISORSE UNMANAGED che appartengono all'oggetto su cui si esegue il metodo, e sarebbe bene evitare di accedere ad altri oggetti managed. Per ovviare al problema del rilascio deterministico occorre dispose(), metodo dell'interfaccia IDisposable, automaticamente chiamato se si utilizza lo statement using

Relazione fra distruttore e finalize con il GC

Per la maggior parte degli oggetti creati dalla propria applicazione, si può contare sul supporto di Garbage Collector di .NET Framework, che svolge in modo implicito tutte le necessarie attività di gestione della memoria. Se però si creano oggetti che incapsulano risorse non gestite, sarà necessario rilasciare esplicitamente queste ultime quando si smette di utilizzarle. Il tipo più comune di risorsa non gestita è un oggetto che incapsula una risorsa del sistema operativo, quale un file, una finestra o una connessione di rete. Sebbene Garbage Collector sia in grado di tenere traccia del ciclo di vita di un oggetto che incapsula una risorsa non gestita, esso non ha una specifica conoscenza di come pulire la risorsa. Per tali tipi di oggetti, .NET Framework fornisce il metodo Object.Finalize, che consente a un oggetto di pulire correttamente le risorse non gestite quando la memoria da esso utilizzata viene reclamata dal Garbage Collector. In base all'impostazione predefinita, il metodo Finalize non effettua alcuna operazione. Se si desidera che Garbage Collector esegua operazioni di pulizia sull'oggetto prima di reclamarne la memoria, è necessario eseguire l'override del metodo Finalize nella propria classe. ESEMPIO DI DISTRUTTORE class Car

```
{
    ~Car()  // destructor
    {
        // cleanup statements...
    }
}
```

Il distruttore chiama in modo implicito il metodo Finalize sulla classe di base dell'oggetto. Il codice del distruttore riportato sopra viene quindi convertito implicitamente nel codice seguente:

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

In questo modo, il metodo Finalize viene chiamato in modo ricorsivo per tutte le istanze nella catena di ereditarietà, dalla più derivata alla meno derivata.

Precondizioni, Postcondizioni definizione precisa

Precondizioni: condizioni che devono essere tutte verificate prima che il caso d'uso possa essere eseguito (vincoli sullo stato iniziale del sistema) Postcondizioni: condizioni che devono essere tutte vere quando il caso d'uso termina l'esecuzione di norma con successo

Invariante di classe come funziona e definizione precisa

E' un vincolo di classe (un'espressione logica) che deve essere sempre verificato sia all'inizio che alla fine di TUTTE le operazioni pubbliche della classe, può non essere verificato solo durante l'eseuzione dell'operazione. In caso di subclassing, gli invarianti devono essere identici o più stringenti.

DIAGRAMMI DI STATO

I diagrammi di stato appartengono al modello dinamico del sistema ovvero quel modello che descrive il comportamento del sistma durante il suo funzionamento. I diagrammi di stato evidenziano i possibili stati che un oggetto puo' assumere e gli eventi interni o esterni che attivano transizioni da uno stato all'altro

MODELLO A CASCATA

E' un modello a fasi distinte in cascata tra loro e con retroazione finale: il prodotto di una fase che va in ingresso alla fase successiva è detto semilavorato. I limiti sono dati in dalla sua rigidità , in particolare sia dall'assunzione discutibile dell'immutabilità dell'analisi, cioè che gli utenti siano in grado a priori di esprimere esattamente le loro esigenze e di conseguenza che l'analisi iniziale sia esaustiva e complete, sia dall'immutabilità del progetto , cioè che sia possibile progettare l'intero sistema prima di aver scritto una sola riga di codice

QUALITA' DEL SOFTWARE

Qualità esterne sono quelle percepibili da un osservatore esterno che esamina il prodotto software come se fosse una scatola nera e sono : AFFIDABILITA', FACILITA' D'USO, VELOCITA'. qualità interne, cioè osservabili esaminando la struttura interna del prodotto software come se questo fosse una scatola trasparente: MODULARITA', LEGGIBILITA'...

TIPI DI EREDITA'

di Modello (riflettono la relazione "isa", cioè una sottoclasse è un sottotipo compatibile con tutti i tipi definiti lungo la sua catena ereditaria e consente il POLIMORFISMO per INCLUSIONE) NOTA: se non vale la relazione IsA, usare la composizione. di Software (esprimono relazioni entro il software stesso piuttostoche nel suo modello) di variazione (un caso speciale che può essere sia di modello che di software)

TIPOLOGIE DESIGN PATTERN

Pattern di creazione: risolvono problemi inerenti il processo di creazione di oggetti
Factory,Builder,Prototype,Singleton

Patter strutturali: risolvono problemi inerenti la composizione di classi o di oggetti
Adapter,Bridge,Composite,Decorator,Facade,Flyweight,Proxy

Pattern Comportamentali : risolvono problemi inerenti le modalità di interazione e di distribuzione delle responsabilità tra classi o tra oggetti Iterator,Mediator,Memento,Observer,State,Strategy,Visitor

ANALISI DEI PATTERNS http://msdn.microsoft.com/it-it/library/cc185064.aspx STRUTTURALI: ADAPTER -> (esempio adattamento di un emplyee ad un customer)

Composite, può utilizzare composizione e delega (esempio capitoli e paragrafi. nel momento di stampare, chapter fa stampare ai figli paragrafi) -> Il pattern Composite, di tipo strutturale basato sugli oggetti, consente di creare gerarchie di oggetti aggregando insieme elementi primitivi e compositi direttamente a runtime. Questo pattern può essere applicato per rappresentare strutture ad albero in una logica parte-tutto (part-whole), dove ciascun elemento può a sua volta aggregare insieme altri elementi della stessa specie e dove oggetti singoli e composizioni possono essere trattati in modo uniforme. Uno dei pregi principali nell’applicazione di questo pattern risiede nel fatto di rendere molto agevole l’aggiunta di nuove tipologie di

oggetti componenti. In più, dal momento che i vari elementi vengono trattati in modo uniforme, l'utilizzo della struttura composita nell'ambito del client risulta essere molto semplificata. Ciascun elemento, seppur presentando caratteristiche simili, può essere caratterizzato da comportamenti assai diversi, a seconda dei casi. Il client non conosce la differenza che esiste tra i diversi elementi e, in particolare, tra oggetti primitivi e oggetti compositi, pertanto tratta entrambe le tipologie allo stesso modo, con un significativo miglioramento della leggibilità del codice.

State- http://dellabate.wordpress.com/2012/03/13/gof-patterns-state/ Si tratta di un pattern comportamentale basato su oggetti che viene utilizzato quando il comportamento di un oggetto deve cambiare in base al suo stato. Questo pattern trova applicazione quando abbiamo a che fare con una "Macchina a Stati Finiti" ossia siamo in presenza di un sistema dinamico in cui i valori di ingresso, uscita e stato sono un insieme finito. Solitamente il codice è pieno di istruzioni condizionali che esaminano lo stato degli oggetti, in particolare esaminano il valore di variabili e costanti al fine di prendere delle decisioni. Molto spesso queste decisioni sono molto complesse e dipendono da molti valori, tutto ciò determina la generazione di grossi blocchi IF-ELSE/SWITCH che spesso contengono anche logica di business. Questo comporta seri problemi di comprensione, manutenzione ed evoluzione del codice. L'utilizzo di questo pattern permette di scorporare i grossi blocchi condizionali ed inserirli negli oggetti di stato, in modo da associare il comportamento di un oggetto al suo stato. Un linguaggio ad oggetti ha delle caratteristiche basate sul Paradigma Orientato agli Oggetti che consentono di creare oggetti con un proprio stato ed un proprio comportamento e l'utilizzo di questo pattern consente di sfruttare queste caratteristiche.

FLYWEIGTH Si tratta di un pattern strutturale basato su oggetti che viene utilizzato per ottimizzare l'utilizzo delle risorse ed evitare la presenza di oggetti duplicati, spesso viene associato a richieste di performance in quanto il suo utilizzo può migliorare le prestazioni di una applicazione. In particolare in Java, ogni qual volta che viene utilizzato l'operatore new, la JVM alloca nell'heap uno spazio in memoria di 32 bit ed una eccessiva generazione di oggetti può saturare le risorse del sistema che, anche se elevate, sono pur sempre limitate. Molto spesso la generazione di nuovi oggetti non è motivata da reali esigenze mentre invece è dovuta a superficialità oppure a errata analisi degli impatti prestazionali. Da qui l'esigenza di riutilizzare gli oggetti precedentemente creati ai fini del loro riutilizzo. L'esigenza di adottare questo pattern può nascere da vari fattori: evitare la generazione di un elevato numero di oggetti evitare di creare oggetti che richiedono molta memoria evitare di creare oggetti che richiedono molto impegno computazionale Disponibilità limitate di risorse: di calcolo, di memoria, di spazio ecc Ogni oggetto presenta uno stato e tale stato è costituito da elementi costanti ed elementi variabili. Gli elementi costanti caratterizzano l'oggetto e sono indipendenti dai fattori contingenti mentre gli elementi variabili sono spesso collegati ad essi. Possiamo fare una distinzione tra stato interno o intrinseco e stato esterno o estrinseco: Lo stato interno fa riferimento alle caratteristiche di un oggetto indipendenti dal contesto di utilizzo e che vengono sempre mantenuti. Lo stato esterno fa riferimento alle caratteristiche dell'oggetto dipendenti dal contesto di utilizzo e che possono cambiare senza che le caratteristiche dell'oggetto mutino. Considerando che lo stato interno è destinato a perdurare, possiamo condividerlo in modo da evitare la generazione di un oggetto duplicato con il medesimo stato, a differenza dello stato esterno che non è da condividere in quanto è soggetto a facili cambiamenti. Quindi al fine della implementazione di un oggetto Flyweight occorre prima di tutto definire lo stato interno per poi definire le proprietà della classe che avranno il compito di mantenerne tale stato. A differenza dello stato esterno che non deve essere mantenuto nell'oggetto, pertanto non indicheremo delle proprietà atte a memorizzarle, in quanto tali proprietà non devono essere condivise. ESEMPIO FLYWEIGHT Come esempio pensiamo al caso in cui vogliamo creare una applicazione di word processing Dalle 26 lettere dell'alfabeto è possibile creare un numero elevato di parole. Se per ogni lettera decidessimo di creare un oggetto, ci troveremmo nella situazione in cui andremmo a creare troppi oggetti, allocando troppa memoria. Per evitare

questo inconveniente decidiamo di creare 26 oggetti, ognuno rappresenta una lettera dell'alfabeto (a, b, c ,d ecc) che indica il suo stato interno, mentre invece altre sue caratteristiche (la dimensione, il colore, la posizione ecc) rappresentano il suo stato esterno.

VISITOR- http://dellabate.wordpress.com/2013/04/02/gof-patterns-visitor/ Motivazione 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? Non mi riferisco alla tecnica di decorazione, trattata in un articolo precedente, ma mi riferisco ad un'altra tecnica di "arricchimento" del codice. 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. Nella prima struttura, che definisce lo stato, è presente il metodo accept() che invoca il metodo visit() 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.

OBSERVER - http://dellabate.wordpress.com/2012/03/03/gof-patterns-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 intervenire 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. 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.

MVC- E' un pattern architetturale, utilizza sottoscrizione e notifica. Può utilizzare un altro pattern (Observer), registrando le View come osservatori del Model. Le View possono quindi richiedere gli aggiornamenti al Model in tempo reale. Inoltre la View delega al Controller l'esecuzione dei processi richiesti dall'utente dopo averne catturato gli input, e la scelta delle eventuali schermate da presentare. Permette di suddividere un'applicazione, o anche la sola interfaccia dell'applicazione, in tre parti Modello elaborazione / stato View (o viewport) output Controller input Controller Ha la responsabilità di trasformare le interazioni dell'utente della View in azioni eseguite dal Model. Realizza la corrispondenza tra l'input dell'utente e i processi eseguiti dal Model. Selezionando le schermate della View richieste, il Controller implementa la logica di controllo dell'applicazione. AGGIONRAMENTO DATI In risposta a qualche evento il metodo handleEvent() viene invocato sul Controller. Il Controller esamina lo stato del modello usando il metodo getState() e invoca un metodo di servizio nel modello per cambiare il suo stato. Il modello cambia lo stato e invoca il suo metodo di notifica per notificare il cambiamento di stato a tutte le View registrate; Ogni View registrata viene notificata invocando il suo metodo update(). La View esamina lo stato del modello tramite il metodo getState() e si autoaggiorna

DESIGN PRINCIPLES

L' OOD risulta pieno di principi e tecniche ingegneristiche per gestire le dipendenze tra i moduli: tali principi sono detti "Design principles". Alcuni sono : -SRP - Single responsability principle : "non ci dovrebbe mai essere più di una ragione per modificare una classe, ovvero ogni classe deve avere una unica responsabilità al fine di ridurre l'accoppiamento e diminuire la FRAGILITA'"

-DIP - Dependency inversion Principle "Fai dipendere le cose dalle astrazioni e non dalle concretizzazioni: le dipendenze devono interessare le interfacce o le classi astratte e non classi concrete al fine di ridurre RIGIDITA' , FRAGILITA', IMMOBILITA'"

-ISP - interface segregation principle "I clienti non devono mai dipendere da interfacce che non utilizzano, a tal fine è meglio avere più interfacce specifiche piuttosto che una unica general purpose"

-OCP - Open/Closed Principle "Le entità software (cioè classi, moduli, funzioni..) dovrebbero essere aperte ad estensioni ma chiuse a modifiche: il segreto sta nell'utilizzo di astrazioni ovvero avere dei moduli che utilizzino astrazioni AL FINE DI MIGLIORARE NETTAMENTE LA RIUSABILITA' (ricorda esempio di readKeyboard e read generico)"

-LSP - Liskov Substitution Principle "Le sottoclassi dovrebbero poter sostituire classi base, onorandone i contratti cioè osservando che le precondizioni vincolano il chiamante che dovrà fare in modo che queste siano meno o ugualmente stringenti rispetto alla definizione originaria e che le post condizioni siano identiche o più stringenti rispetto alla definizione originaria" il motivo di questo è che un cliente che invoca il metodo conosce il contratto definito a livello della classe base e quindi non è in grado di soddisfare precondizioni più stringenti o di accettare post condizioni meno stringenti

TIPI DI MANUTENZIONE

CORRETTIVA -> correzione di errori che non sono stati scoperti nelle fasi precedenti. ADATTATIVA -> aumento dei servizi forniti dal sistema in seguito a definizione di nuovi requisiti PERFETTIVA-> miglioramento delle caratteristiche delle unità del sistema

INTERFACCE E CLASSI ASTRATTE

INTRFACCE -Deve descrivere una funzionalità semplice, implementabile da oggetti eterogenei (cioè appartenenti a classi non correlate tra di loro) Ad esempio: • le istanze di tutte le classi che implementano l'interfaccia ICloneable sono clonabili • le istanze di tutte le classi che implementano l'interfaccia IList sono trattate come collezioni . -Può "ereditare" • da 0+ interfacce -Non può essere istanziata -Non può contenere uno stato -Non può contenere attributi e metodi (e proprietà ed eventi) statici (a parte eventuali costanti comuni) -Non contiene alcuna implementazione -Le classi concrete che la implementano: • devono realizzare tutte le funzionalità -Deve essere stabile Se si aggiungesse un metodo a un'interfaccia già in uso, tutte le classi che implementano quell'interfaccia dovrebbero essere modificate -Non può gestire la creazione delle istanze delle classi che la implementano -La creazione deve essere effettuata • dai costruttori delle suddette classi • da (un'istanza di) una classe non correlata, la cui unica funzionalità è la creazione di istanze di altre classi (classe factory)

CLASSE ASTRATTA -Può descrivere una funzionalità anche complessa, comune a un insieme di oggetti omogenei (cioè appartenenti a classi strettamente correlate tra di loro) -Può "ereditare" • da 0+ interfacce • da 0+ classi (astratte e/o concrete) • minimo 1 classe, se esiste una classe radice di sistema • massimo 1 classe, se non è ammessa l'ereditarietà multipla -Può contenere attributi e metodi (e proprietà ed eventi) statici-Le classi concrete che la estendono: • devono realizzare tutte le funzionalità non implementate • possono fornire una realizzazione alternativa a quelle implementate -Può essere modificata Quando si aggiunge un metodo a una classe astratta già in uso, è possibile fornire un'implementazione di default, in modo tale da non dover modificare le sottoclassi -Può gestire la creazione delle istanze delle sue sottoclassi-la creazione può essere effettuata come per l'interefaccia ma anche da un metodo statico della classe atratta (factory)

DIFFERENZA TRA AZIONE E ATTIVITA'

Azione -> computazione atomica (cioè non interrompibile) associata ad una transizione Attività -> computazione NON atomica (interrompibile) associata ad uno stato o ad un insieme di azioni