Appunti Ingegneria del software

In C++ abbiamo la necessità di avere un punto di accesso globale per quanto riguarda i main. Cioè, se abbiamo più main sparpagliati in varie classi, abbiamo bisogno di un unico main selettore che ci fa scegliere dove far partire e come.

In Java questo non è necessario, ci basta invece far eseguire solamente un file tramite un'unica riga di comando inserendo l’oggetto da cui vogliamo partire.

I **package** sono dei contenitori di più classi. Questi package possono essere contenuti dentro altri package e così via.

Tutti i nomi di classe cominciano con Maiuscole. Stessa cosa per quanto riguarda le parole composte es. LocalDate.

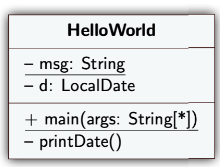
Le classi in Java sono dei file sostanzialmente. Quindi quando importiamo ad esempio java.time.LocalDate, vuol dire che troveremo dentro un package chiamato java, dentro un altro package chiamato time, il file chiamato LocalDate (che è per l’appunto la classe). I package sono intesi a livello fisico come delle cartelle.

Il main è dichiarato **public** perché deve essere accessibile, **static** perché abbiamo bisogno che questo venga caricato in memoria prima di istanziare un oggetto della classe. In fine deve essere contenuto come parametro del main (**String[] args**) perché deve contenere gli eventuali dati da tastiera.

Ciò che tu dichiari statico, che è quindi già caricato in memoria, è quindi già accessibile perché già caricato in memoria. Dunque possiamo dire che è come se fosse **Globale**. Le variabili static public, proprio perché sono Globali, possono essere considerate come un moto per comunicare tra le varie classi.

Una variabile dichiarata come **Final** vuol dire che è immodificabile. (const di c++). Se lo metto davanti ad un metodo, vuol dire che non può effettuare override. Se invece si trova davanti ad una classe, non può essere ereditata.

La notazione UML è il modo di rappresentare una classe. Abbiamo tre sezioni, nella prima ci sta il nome della classe, nella seconda sezione troviamo le variabili che ha e in fine nella parte troviamo la dichiarazione dei metodi.



Esistono dei programmi per passare da UML a Java in modo da dover solamente “riempire” i metodi delle classi.

Un’altra differenza tra C++ e Java sono i puntatori, infatti in Java questi sono stati tolti ai programmatori in modo da evitare la manipolazione diretta della memoria. Un nome indica la locazione di memoria dove risiede un certo dato; dunque, di base il nome della variabile è già un puntatore.

Diciamo che un linguaggio è fortemente tipizzato se dobbiamo sempre assegnare un tipo ad un indicatore e non possiamo assegnare ad una variabile di un tipo ad un tipo non compatibile. (a meno di casting PURAMENTE ESPLICITO).

L’ereditarietà in Java è UNICA, non possiamo avere ereditarietà multipla.

14-03-2024

Caratteristiche peculiari del paradigma OBJORIENTED

Qualunque problema ci viene dato, partiremo sempre dall’insieme di dati di ingresso e cosa questi devono dare in output. Dunque, la cosa da cui dobbiamo partire è sempre la struttura dati.

La nascita delle classi, ci permette di capire che gli algoritmi e le strutture dati (classi) sono un tutt’uno e ciò non viene fatto solo alla vista del programmatore, bensì anche al compilatore. Grazie a ciò noi abbiamo le tre caratteristiche delle programmazione ad oggetti, ossia **Incapsulamento**, **Ereditarietà** e **Polimorfismo.** Queste tre caratteristiche erano da sempre possibili anche nella programmazione strutturata, tuttavia adesso (grazie ai linguaggi di programmazione ad oggetti), ci rendiamo subito conto se questi vengono usati erroneamente.

L’incapsulamento consiste in due concetti: ossia l’integrazione dei metodi che si possono fare con un certo oggetto e la limitazione degli accessi ai valori interni dell’oggetto.

Esempio Stack pubblicato da lui su github. Con l’esempio dello stack, vediamo come se definito in quel modo, una volta incluso Stack.h, siamo sicuri che il codice creato da noi (stack.h) non può andare a creare danno proprio grazie alla caratteristica dell’incapsulamento.

Nello stack versione migliorata non posso avere più stack contemporaneamente.

La final version dello stack è quella ADT (Abstract data type) aggiungiamo un nuovo tipo di dato in modo da poter definire più stack a sé stanti.

L’ADT è un insieme di caratteristiche che descrivono una categoria di oggetti (Classe). Il modello è la rappresentazione di un oggetto (istanza della classe).

Dobbiamo evitare il conflitto tra i nomi e dunque vogliamo che questi siano unici. La funzione destroy potrebbe essere generica per più ADT, se invece decidiamo di chiamarla Stack\_destroy ecc.

Nonostante l’implementazione StackADT, abbiamo ancora problemi perché possiamo ancora entrare dentro dati sensibili dello stack.

In fine quindi, ci rendiamo conto di quanto la classe effettivamente rimuova la maggior parte dei problemi legati alle precedenti versioni. Il fatto che stiamo definendo degli ADT se ne rende conto anche il compilatore e ci evita dei problemi che normalmente sarebbero permessi.

In C++ separiamo l’ADT (file.h) dalla suo modello (file.cpp). In Java questa cosa non la facciamo.

Se avessimo definito StackOverflow dentro la classe Stack, allora lo stackoverflow sarebbe stato richiamabile solo dallo stack e non da altre classi.   
GUARDA BENE LA VERSIONE JAVA.

21/03/2024

JAVA COLLECTIONS FRAMEWORK

Ci sono delle framework di java che ci vengono fornite. Queste sono delle interfacce che Java stesso ci fornisce in modo che noi non dobbiamo scriverle.

L’interfaccia collections si possono distinguere per esempio list, deque, map, set. Ognuno di questi corrispondono a più adt che lo implementano.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente  
L’interfaccia si distingue da un ADT perché di base nell’interfaccia non esistono gli attributi. L’interfaccia ha quindi solo dei metodi (protocollo).

Arraylist vs Vector

Per rappresentare un aggregato di elementi sequenziali abbiamo il dubbio se usare uno o l’altro.  
La prima differenza è che l’array list non è sincronizzato mentre vector è **sincronizzato**. (Vuol dire che se io ho qualcosa che è condiviso tra più thread, se è sincronizzato, se A sta facendo qualcosa su quel pezzo di codice sincronizzato, allora gli mette un lock e quindi il thread B non può modificarlo)

L’array list è più veloce rispetto al vector perché non è sincronizzato.

Sono entrambe iterabili e basate sull’array. Gli array list una volta che finisce lo spazio, riallocano un’altra parte di array grossa la metà dell’array originale, i vector invece riallocano un vettore grande uguale.

Gli array non sono legacy(?) i vector si.

Pattern Iterator for(T b : array)

Grazie al pattern iterator noi siamo in grado di scorrere un aggregato senza esporre la sua struttura, quindi indipendente dalla sua implementazione. Abbiamo un accesso indiretto che viene fatto per mezzo di un terzo oggetto che è l’iteratore che quindi sarà lui ad avere l’accesso privilegiato e non il client. Ogni istanza di Iterator, sarà indipendente dalle altre e quindi può essere diversa. Questi pattern possono visualizzare anche grafi o alberi.

Questo è un vecchio tipo di notazione che poi confluisce in uml. La freccia tratteggiata ha come significato che esiste una dipendenza tra due cose. In questo caso il legame implica che, se io creo concreteaggregate dipende da concreateiterator. La freccia continua dice che un’istanza di una classe verrà usata da un’altra.

Il client può quindi usare Aggregate per creare un oggetto Concreate Aggregate. Quindi Aggregate usa istanze di Concreate Aggregate. È come se dentro una classe A, usassi un oggetto di tipo B. Dunque, A avrebbe bisogno di conoscere esplicitamente B. La freccia che parte da ConcreteAggregate e arriva ad Aggregate indica che i metodi scritti in aggregate sono implementati in concreteaggregate.

Immagine che contiene testo, diagramma, schermata, linea

Descrizione generata automaticamente

Possiamo pensare al nostro codice così:

Immagine che contiene testo, diagramma, schermata, linea

Descrizione generata automaticamente

Cioè noi abbiamo una lista abstract che, a prescindere dalla sua implementazione, può fare determinate cose. Una volta che vogliamo scorrerla in base al tipo, abbiamo bisogno di un iteratore diverso che deve esserci fornito dal tipo di lista che implementiamo a tempo di esecuzione.

02/04/2024

Accoppiamento: è il numero di dipendenze che ha con altre classi. Se la classe A utilizza la classe B e anche la classe C, quando io modifico la classe B devo stare attento appunto ad A che la utilizza. Vogliamo chiaramente che le dipendenze siano il meno possibile perché appunto se la classe B è utilizzata da tante altre classi, allora dovrò andare a riguardare tutte le altre a cascata. Meno dipendenze ho, più facile sarà modificare il mio codice. L’accoppiamento è quindi una misura qualitativa che indica quante dipendenze abbiamo.

Coesione: è un’altra metrica dei programmi scritti ad oggetti ed è interna alla sola classe. Questa va’ ad indicare quanto effettivamente i metodi che sono dentro una classe servono lì dentro. Un metodo può essere Stateful o Stateless, ossia il risultato dipende da altri stati che sono puramente dell’oggetto e non del metodo. Dipendenza dei metodi tra loro (nella stessa classe). Il fatto che esistano queste dipendenze è naturale, al contrario dobbiamo porci delle domande nel momento in cui il metodo non è legato all’oggetto, a maggior ragione se questo metodo ha bisogno di un altro oggetto per funzionare.

Principi di progettazione riusabile

Principio della singola responsabilità: Un metodo deve fare SOLO UNA COSA.

Programmiamo in base alle interfacce e non in base alle implementazioni: Vogliamo scrivere del codice indipendente dalla sua implementazione. Per fare ciò possiamo usare classi astratte ecc. esempio dell’orologio

Favorire sempre la composizione all’ereditarietà: Vedi differenza tra composizione e ereditarietà. La differenza sta nel fatto che la composizione non crea estensioni di tipo.

Usare sempre tipi parametrici: Questo è utile per creare codici generici.

Document Editor

Problema di design. Come facciamo a scrivere codice per una struttura eterogenea per far sì di permettere una navigazione facile di tutti i suoi elementi?  
Un composite ci permette di definire una struttura aggregata che può contenere o tipi semplici o tipi complessi. Per navigarlo possiamo utilizzare il pattern iterator.

04/04/2024

Design pattern Singleton

Il nostro intento tramite questo design pattern è quello che una classe possa essere istanziata solo una volta o comunque voglio limitarne le istanze.

Questo può essere utile se vogliamo un NullObject a cui tutti puntano quando andiamo a fare riferimento in un albero, in modo da non avere tante classi istanziate. Stessa cosa anche un log che noi vogliamo unico.

Grazie a questo pattern possiamo creare una “variabile globale” come un oggetto senza doverlo dichiarare come statico.

L’idea è quella di rendere il costruttore della classe privato in modo da non permettere l’istanziazione della classe a piacere. Visto che il costruttore è privato, per istanziare la classe, utilizziamo un altro metodo che andremo a definire noi dentro la classe stessa. Questo metodo, che chiameremo **getinstance,** per essere funzionante dovrà essere statico in modo che venga subito creato in memoria all’inizio dell’esecuzione del programma. Al suo interno conterremo il riferimento alla sua istanza, se questa è null, allora noi faremo la new Oggetto (invocandone il costruttore da dentro la classe stessa), altrimenti ritorneremo il puntatore all’istanza. Oltre al metodo statico, avremo anche un altro attributo statico che conterrà a tutti gli effetti l’oggetto.

VEDI ESEMPIO SINGLETON FIB E LOGS.

Considerazioni sul Singleton

Togliamo al client il controllo dell’istanziazione della classe, l’oggetto è un’istanza normalissima della classe e dunque non l’abbiamo neanche dovuta rendere statica. Per renderla Multiton, ci basta usare anziché una sola variabile, una lista che contiene tutti gli oggetti. Naturalmente così facendo dobbiamo creare un metodo anche per eliminare una istanza dalla lista.

CREA UN BST CON NULLNODE E SINGLETON E CHE RENDILA ITERATOR

Object Oriented Design

La parte fondamentale della programmazione ad oggetti è il riuso del codice. Di fatto quando noi andiamo a creare delle classi che durino per sempre e che quindi siano modificabili nella maniera più semplice possibile. Queste classi devono essere specifiche per il nostro problema ma non troppo, perché altrimenti il riuso del codice diventa difficile.

Le sottoclassi possono aiutare al riuso del codice tramite la funzione del partizionamento. Usare un’interfaccia anziché una classe astratta ci permette di eliminare il fattore “gerarchia” e dunque non abbiamo bisogno di creare una subclass quindi non c’è alcun legame con una ipotetica classe padre. Infatti le interfacce possono essere utilizzate da tutte le classi e non solo da una specifica gerarchia, possono quindi avere senso in più categorie.

09/04/2024

UML: freccia tratteggiata A dipende da B. Freccia normale A usa B. Freccia con rombo, indica un’istanza di A come ciclo di vita dipende da un’istanza di B (A non può funzionare senza un’istanza di B).

Factory Method

In un algoritmo so che devo istanziare un oggetto di servizio e quando farlo, ma non so quale oggetto mi serve precisamente.

Primo principio: programmiamo usando interfacce: so cosa devo fare e che devo farlo in un certo momento ma non ancora.

Incapsulo la creazione dentro un metodo dà come valore di ritorno l’oggetto che ci serve. Il mio algoritmo può essere scritto solamente conoscendo l’interfaccia dell’oggetto

Immagine che contiene testo, diagramma, linea, Carattere

Descrizione generata automaticamente

Scriviamo la classe application che ha come attributi per esempio createdocument, newdocument e opendocument. Questi sono molto legati al tipo di documento che dobbiamo utilizzare; tuttavia, noi vogliamo che questo codice sia il più generico possibile, dunque quello che possiamo fare è utilizzare delle interfacce o comunque delle classi astratte, in modo che successivamente ogni sua sottoclasse implementi da sé le sue funzionalità. Tutto ciò perché application potrebbe utilizzare più documenti (Pdf, word, excel, ecc.)

Schema del pattern:

product è l’interfaccia comune degli oggetti creati da factory method()

concreteproduct è l’implementazione del product

creator dichiara il factorymethod(), quest’ultimo ritorna un oggetto di tipo product e usa l’oggetto product nel suo coidce

concretecreatore implementa il factorymethod() e ne fa ovverride, sceglie quale concreteproduct istanziare

Immagine che contiene testo, diagramma, linea, schermata

Descrizione generata automaticamente

Il primo client del factorymethod è il creator stesso.

Il factory method non è per forza chiamato solo dai creator, bensì altri clients possono trovare i factory method utili, specialmente nel caso di gerarchie di classi parallele. Speller e biglietti

Immagine che contiene testo, diagramma, linea, Piano

Descrizione generata automaticamente

Questo design pattern è stato utilizzato per creare il pattern iterator.

Possiamo implementare anche il factoryMethod anche come statico perché così un solo creator può anche creare tutti i concretecreator

11/04/2024

Abstract Factory

L’intento di questo pattern è quello di fornire un’interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificarne le loro classi concrete. La differenza dal factory method è che gli N prodotti con cui voglio lavorare sono tutti legati tra loro e non possono essere cambiati in maniera indipendente. Dunque, se io voglio modificare un prodotto, non voglio modificare solo il suo factory method ma anche modificare il creator a tutti gli effetti. Abbiamo un Creator di creator.

Immagine che contiene testo, diagramma, Piano, linea

Descrizione generata automaticamente

In questo caso è come se creassimo una Factory di Factory. Vediamo un esempio.

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

Come possiamo vedere abbiamo un Widget-Factory, che crea a sua volta delle factory di widget per magari il Mac per Windows ecc. Questo può servire perché, se vogliamo ad esempio creare un Linux Menu, non possiamo farlo singolarmente, bensì dovremmo creare anche tutte le altre cose che ci stanno attorno, cioè la window e il button.

Abbiamo altre varianti di implementazione oltre a quella classica.

Immagine che contiene testo, Carattere, schermata, documento

Descrizione generata automaticamente

Questa versione ha come differenza il fatto che nell’abstract factory abbiamo dei riferimenti alle cose che vogliamo implementare e che poi dovranno essere a tutti gli effetti loro a creare l’oggetto singolarmente senza che noi dobbiamo necessariamente andarli a ricreare..

Adapter

Questo pattern non è di tipo creazionale, tuttavia implementa alcune caratteristiche che lo rendono simile. Supponiamo di avere due codici di cui uno un oggetto e un altro un client. Questo client si aspetta un determinato tipo di oggetto che non è quello che già abbiamo; tuttavia, tramite l’adapter andiamo a fare una specie di conversione per renderlo simile a quello che il client si aspetta.

Immagine che contiene testo, Carattere, diagramma, linea

Descrizione generata automaticamente

Rispetto al factory method, qui noi tramite l’interfaccia target, andiamo ad utilizzare degli oggetti che vengono adattati. Di questo pattern esistono due versioni, Class Adapter e Object Adapter.

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

Tendiamo al voler utilizzare l’object adapter per via della sua flessibilità, ma un’altra differenza sta nel fatto che abbiamo da una parte un legame statico (versione della classe) e dall’altra un legame di tipo dinamico (oggetto).

Prototype

Questo pattern è di tipo creazionale ed ha uno schema simile a quello del Factory, tuttavia, cambia il fatto che noi il metodo per crearsi non lo mettiamo in oggetti a parte, bensì dentro l’oggetto stesso. Il prototipo è un’istanza di un certo tipo che usiamo come riferimento per creare istanze dello stesso tipo. Quindi vogliamo aggiungere un metodo clona alla classe, così che l’istanza si “sdoppi” in due.

16/04/2024

Facade

Questo pattern si pone l’obbiettivo di fornire un’interfaccia unificata al posto di un insieme di interfacce in un sottosistema. Questo non è un pattern creazionale ma un pattern Strutturale. Definire un’interfaccia di alto livello che rende il sottosistema facile e comodo da usare.

Immagine che contiene linea, testo, diagramma, schermata

Descrizione generata automaticamente

Permetto al sistema di essere visibile come un’unica entità anche se al suo interno può essere composto da più sottosistemi. Per fare ciò andiamo ad introdurre un’altra classe Facade che si frappone tra il client e sottosistemi. Andiamo quindi a ridurre il numero di dipendenze tra il client e i vari sottosistemi, introducendo un’unica dipendenza tra il client e la classe facade.

Immagine che contiene diagramma, linea, schizzo, Disegno tecnico

Descrizione generata automaticamente

Il facade ha come funzione quella di rimbalzare le richieste del client, all’oggetto coinvolto. In questo pattern non devo andare ad aggiungere funzionalità che prima non c’erano, posso tuttavia aggiungere dei piccoli accorgimenti ai metodi che già prima c’erano senza andarne a modificare il vero e proprio significato. Ad esempio, tramite il facade posso andare a verificare delle condizioni che prima non andavo a controllare.

Applicando questo pattern, nascondo al client l’implementazione del sottosistema, e faccio a tutti gli effetti un re factoring. Posso applicare questo pattern nel momento in cui ho tanti blocchi di “sottosistemi” per facilitarne l’uso. Un’altra cosa da notare è che, una volta applicato, se cambiamo una classe del sottosistema, mi basta ricompilare il sottosistema fino al facade, quindi non ricompilo il client. Possiamo considerare quindi questo come una facciata che implementa varie funzionalità.

Questa classe non previene l’accesso diretto quando occorre.

A livello di implementazione questo pattern non deve aggiungere delle funzionalità ma può rielaborare e comporre funzionalità esistenti. Se voglio limitare l’accesso diretto dalle classi posso creare la classe dentro l’altra classe facade.

Immagine che contiene testo, Carattere, ricevuta, schermata

Descrizione generata automaticamente

La logica di questo esempio è che il client andrà ad utilizzare la classe registro come facade, senza questo, dovrebbe andare ad utilizzare tutte le sottoclassi assieme in maniera specifica.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Template

Questo è un pattern di tipo comportamentale. Rientra nella categoria di soluzioni che danno la possibilità di modificare il comportamento del mio sistema. Immaginiamo di creare un **Framework** che quindi è del codice che noi possiamo utilizzare per supportare più tipi.

Voglio fissare per esempio prima ciò che voglio fare in un algoritmo ad alto livello e do la possibilità di implementare i passi dell’algoritmo in un secondo momento. Quindi alcuni passi dell’algoritmo li potremmo lasciare astratti.

Immagine che contiene testo, schermata, Carattere, algebra

Descrizione generata automaticamente

In questo caso qui possiamo modificare i modi in cui implementiamo i singoli passi; tuttavia, qui non possiamo modificare l’algoritmo OpenDocument. Questo metodo è un metodo Template perché è implementato ma non è ancora eseguibile perché manca la vera e propria implementazione.

Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente

L’intento qui è quello di implementare un algoritmo standard del quale però abbiamo dei pezzi che devono necessariamente essere implementati dalla sottoclasse che lo implementa. Un pattern simile è il Factory Method che però differisce per intento, infatti il factory method istanzia oggetti, mentre invece qui cambia i metodi. Offro quindi l’algoritmo che però necessità di implementare alcune particolari funzioni senza modificare lo scheletro dell’algoritmo. Tornando all’esempio di prima, l’algoritmo Open Document rimarrà sempre quello, ciò che cambierà saranno i metodi CanNotOpenDocument, AddDocument ecc. Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Abbiamo quindi una differenza con il polimorfismo, perché noi non andiamo a cambiare per intero il templateMethod, andiamo a modificare solo i vari step primitivi. Il templateMethod DEVE NECESSARIAMENTE RIMANERE QUELLO.

Immagine che contiene testo, Carattere, schermata, ricevuta

Descrizione generata automaticamente

Da Java 8 in poi possiamo avere nelle interfacce dei metodi template non modificabili, mettendo la parola chiave default.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Questo pattern lo implementiamo nel momento in cui voglio che alcune parti dell’implementazioni siano fisse e non modificabili.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Un caso particolare è l’applicazione di una lazy inizialization.

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

Qui una volta assegnato il valore 10, non potremmo più modificarlo a meno che non mettiamo un parametro nel metodo getField. Applicando però il pattern, la situazione cambia.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Strategy

Questo pattern serve a definire una famiglia di algoritmi correlati, incapsularli singolarmente e renderli intercambiabili. Utilizziamo algoritmi che hanno degli elementi interni che potrebbero essere intercambiati in un modo piuttosto che in un altro, di fatto cambiando la strategia. A parità di algoritmo, individuiamo un momento decisionale che può andare a cambiare la strategia. (esempio ordinare tramite selection sort cambiando un < con un >, l’algoritmo sarà lo stesso ma cambiamo l’ordninamento tra ascendente e discendente)

Tramite questo pattern non vogliamo decide una strategia a tempo di compilazione, bensì lo vogliamo fare a run time quindi a tempo di esecuzione.

Immagine che contiene testo, ricevuta, diagramma, linea

Descrizione generata automaticamente

La differenza dal Template è che la parte che deve essere modificata è solamente una e tutto il resto è identico. Chi usa l’algoritmo sceglierà la strategia specifica. Il client istanzia strategy e il contex. Il contex userà la strategy che gli verrà passata dal client. Non cambia l’algoritmo ma ne modifica l’effetto.

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

Scegliamo il layout a sinistra

Se utilizzassimo in questo caso il TemplateMethod, dovremmo sottoclassare 20 diversi layout e per ogni layout dovremo fare un’ulteriore sottoclasse per ogni component che sono circa 40, arrivando a derivare circa 800 classi.

Un altro esempio di utilizzo potrebbe stare nel modo di ordinamento. Posso ad esempio ordinare una matrice per colonna nome, o per colonna cognome ecc.

Una cosa implementata da Java è l’interfaccia Comparable.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Rendiamo una classe “comparabile”. In questo caso lo facciamo per mezzo del ranking e vuol dire che ciò che verrà ritornato è il risultato di questa comparazione che risulterà un numero negativo se il primo è minore del secondo, positivo se il primo è maggiore e 0 se sono uguali.

Oltre il comparable, troviamo il comparator. Il comparator la possiamo implementare se vogliamo una classe che si occupa solo del confronto.

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

Quindi andiamo a guardare il codice. Supponiamo di avere una lista di Player che vogliamo ordinare.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

In questo caso avremo il Contex che sarebbe quindi l’algoritmo di ordinamento mentre la strategy da scegliere sarebbe il campo per cui fare l’ordinamento.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Ci sono vari modi per utilizzare il comparator, per mezzo di notazione lambda op er Comparator.comparing ecc.

Questo pattern lo applichiamo quando vogliamo delle versioni diverse dell’algoritmo. Ci potremmo rendere conto di doverlo utilizzare nel momento in cui ci troviamo degli switch. Un’altra cosa che posso fare è applicare il pattern per trasformare più classi in una classe sola. O ad esempio se vogliamo che il client possa decidere da sé che venga fatta una certa elaborazione dei dati senza che lui possa andare ad accedere in modo diretto ai dati.

Posso definire anche le varie strategy dentro il context stesso in modo che questa possa accedere anche agli attributi privati della outer classe senza problemi.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente.ì

Per implementare il pattern la prima cosa da fare è definire le interfacce delle strategie e del contesto e come queste devono interagire. Il contesto può passare i dati alla strategia come parametro o la strategia ha accesso privilegiato al contesto.

Posso inoltre implementare le strategie come parametri template dei contesti.

Come conseguenze, il client deve conoscere la strategia da utilizzare mentre il contesto non deve necessariamente saperlo.

Template vs Strategy

Entrambi i pattern sono comportamentali. Tuttavia, il Template si occupa di creare un algoritmo scheletro che avrà alcune parti modificabili e altre no, mentre lo strategy ha un algoritmo completo per il qualche cambia solamente un dettaglio critico per la sua implemetazione. Il template viene occupato sottoclassando, mentre invece lo strategy viene applicato scegliendo in un client a tempo di esecuzione la strategia.

State

Questo pattern assume che abbiamo un algoritmo in un determinato contesto che deve cambiare comportamento nel momento in cui abbiamo un determinato stato del contesto. Dunque non è più il client a decidere il comportamento dell’algoritmo, bensì è il contesto stesso.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Per il client l’effetto finale potrebbe anche essere completamente diverso rispetto a uno stato. Manteniamo l’oggetto esterno ma ne cambiamo l’interno. Potremmo anche non avere una parte in comune tra i vari stati. Possiamo applicare questo pattern nel momento in cui abbiamo dei metodi che sono profondamente legati allo stato del contesto

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

Un esempio dell’applicazione di questo pattern può essere questo che tuttavia va’ a mancare un punto fondamentale del pattern. Infatti, in base allo stato che abbiamo di libro decidiamo quale modo usare per stampare. Però, questo stato lo sta decidendo di fatto il client ma la corretta applicazione di questo pattern va’ a controllare uno stato dell’oggetto che per la maggior parte delle volte è privato e non dipende in alcun modo dal client.

07/05/2024

Gli strumenti per gestire la complessità sono l’astrazione, la decomposizione e la gerarchia. Astrarre vuol dire semplificare in modo da poter ridurre la complessità, un esempio è il raggruppamento di alcune caratteristiche dentro una categoria come può essere una classe. Infatti, così facendo, raggruppando solo alcune caratteristiche ne nascondiamo alcune che non ci servono. L’astrazione è la creazione di buon modello. Un buon modello rappresenta solo le caratteristiche che servono a noi (di fatto semplificando) in quel momento. Quindi quando dobbiamo fare una modellazione, dobbiamo anche chiederci a cosa serve quel modello.

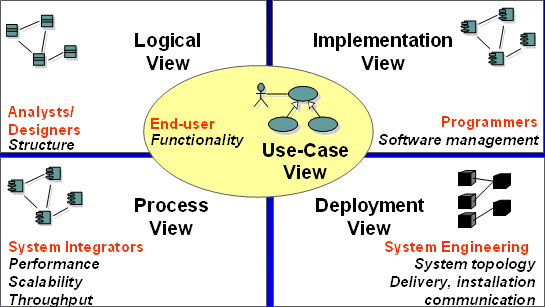
Un modello è un’astrazione che descrive il sottoinsieme di un sistema che può essere complesso

Una vista è una descrizione parziale che prende solo una caratteristica del modello.

Una notazione è il modo in cui noi rappresentiamo una vista.

Le informazioni di un modello e quelle di una vista si sovrappongono.

UML è uno strumento per effettuare una modellazione. Complessivamente UML usa 5 categorie di diagrammi per il quale ognuno fornisce una vista specifica del sistema piuttosto che un'altra.



La process view è una rappresentazione dinamica mentre le altre sono statiche. In questa view non si parla di classi ma di classi ATTIVE. La logical view rappresenta le relazioni logiche tra le varie classi.

Quello che noi conosciamo è un approccio all’ingegneria del software basato sui modelli. Il modello è quindi in genere ognuno di questi diagrammi sommati perché capiamo di cosa stiamo parlando solo quando li vediamo tutti dato che ognuno ne descrive una parte.

Tutta la fase di design è un lavoro che noi facciamo PRIMA di andare ad effettuare coding. Noi andiamo prima di tutto a generare la struttura del nostro programma che il suo codice anche perché è molto più semplice ragionare sulle strutture piuttosto che su 40milioni di linee di codice.

Una story card racconta il modo in cui un sistema deve funzionare.

Distinguiamo vari tipi di modelli quelli **funzionali oggetti** () **e dinamici.**

Rappresenta le funzionalità che un sistema deve avere (pagina 271). Inoltre, questi pallini che ci sono non devono essere necessariamente a livello implementativo, ma saranno anche astratti. Questi casi sono molto simili alle story card (scenari).Dopo aver individuato le funzionalità, dobbiamo trovare la struttura di questi oggetto. Andiamo ad effettuare un’analisi lessicale di ciò che il cliente dice. Partendo dallo use case questo lo guardiamo come una specie di flusso di eventi che racconta in un certo senso ciò che succede. Andiamo a cercare i nomi e i verbi e capiamo cosa sono oggetti, ciò che sono funzioni o metodi ecc., sorgenti di dati o ciò a cui dobbiamo mandarli e ciò che invece sta fuori. Analizzando questo testo avremo una prima idea della nostra struttura.

Dopo aver dato una prima struttura, possiamo costruire un diagramma di classi.

Model base software enginer

1 formuliamo una serie di scenari con l’utente (user story)

2 estraiamo da queste gli use case

3 analizziamo il flusso di eventi

4 generiamo il diagramma di classi.

Questa parte si trova nell’analisi dei requisiti.

Modellazione dinamica

Possiamo usare anche qui i diagrammi UML ma anche gli activity diagram, quelli di sequenza e quelli di stati (macchine a stati finiti). L’activity diagram è qualcosa che possiamo rappresentare come una sequenza di azioni, in questi diagrammi possiamo anche inserire ad esempio le sincronizzazioni, i bivi e le varie “corsie” (simile al diagramma di sequenza ma è diverso perché nelle corsie ci focussiamo sull’attività in genere.). Un atro diagramma è il Collaboration Diagram dove tutte le interazioni tra due oggetti sono tutti corrispondenti ad un solo arco, mettiamo in evidenze quindi Chi collabora con chi.

Il diagramma di stati è mirato al mostrare un solo adt come va a mostrarsi. Le transizioni di stato possono non solo avere ciò che li fa scattare ma anche una sorta di guardia tra [] che magari fanno accadere quella cosa solo se è rispettata una condizione e una / che indica ciò che dopo il trigger lo fa scattare.

Gli stati composti possono essere degli stati che consistono in vari sottostati sequenziali o concorrenti.

Parliamo della decomposizione. Ragioniamo quindi su quello cha abbiamo prodotto prima dalla fase di astrazione. Utilizziamo due tecniche, la decomposizione **funzionale** e quella **OBJ-oriented**. Le story card servono per raccontare le funzionalità e non per dare le entità.

La decomposizione funzionale ci dice di ragionare in maniera sequenziale senza usare oggetti. Quindi abbiamo un goal da raggiungere e lo scomponiamo in varie funzionalità che ci permettono poi di raggiungere quel goal. Individuiamo una sorta di scomposizione in moduli che rappresentano funzionalità. Scomporre le funzionalità va a spalmarsi tra più strati che andranno a comporre la funzionalità che serve e ogni strato può essere assegnato a degli oggetti che li eseguono.

La decomposizione ad oggetti è completamente differente da quello funzionale. Qui partiamo dalle entità per capire come comporre il sistema.

Noi non utilizziamo o una o l’altra per creare un sistema, bensì le utilizziamo entrambi perché sono entrambe importanti allo stesso modo.

Quando si arriva al momento in cui le funzionalità sono sensatamente correlate ad un metodo di un oggetto, ci fermiamo con la scomposizione funzionale.

Identificazione di una classe

È importante capire prima di tutto a cosa serve la classe. Se dobbiamo modellare una valigia, implementeremmo apri, chiudi ecc. Tuttavia, noi se la utilizzassimo per sederci sarebbe ancora una valigia o una sedia? Di fatto, noi creiamo una classe non per rispecchiare i suoi attributi, bensì per i metodi che usa.

L’ultima tecnica è la **gerarchia** che ci permetta di dare struttura al nostro sistema. La gerarchia comprende utilizzo, aggregazione, composizione e ereditarietà.

Da queste andiamo a stabilire alcune architetture per organizzare il nostro software.

14/05/2024

L’architettura è tutto ciò che definisce il nostro sistema. Possiamo dire che sia composta dai nostri oggetti e come questi si relazionano tra loro.

Quali sono i componenti? Quali interfacce usano? Come si relazionano tra loro?

Dobbiamo separare il nostro software in unità che in generale chiamiamo Moduli (package che rappresentano un sottosistema del sistema complessivo). Il principio che ci guida è minimizzare l’accoppiamento e massimizzare la coesione interna.

Se ho fatto correttamente l’identificazione delle entità e delle funzionalità, questi saranno oggetti di livello più “basso”. E da qui ci accorgiamo subito se abbiamo un oggetto di livello più alto che magari dovrà essere esplosa in più classi di implementazione.

Di fatto nell’identificazione abbiamo effettuato un **partizionamento.**

Ci sono anche altri principi che possiamo seguire se abbiamo effettuato correttamente la divisione in moduli. Il primo è **reuse/release equivalence principle**. Quando noi creiamo un sistema lo facciamo tramite dei tracking system. La granularità del riuso è la granularità della release. La dimensione della porzione nuova del codice dovrebbe essere equivalente alla porzione di codice di un modulo.

È sbagliato fare un incremento delle funzionalità toccando più package usandoli male. Un package deve contenere un solo tipo di cose ben specifiche.

Un altro principio è il **common reuse principle.** Le classi dentro un package devono poter essere utilizzate assieme. Cioè, se noi dobbiamo utilizzare una delle classi in un package, devono essere riutilizzate tutte assieme. Anche all’interno di un modulo, quando riuso il modulo ho bisogno di TUTTO il modulo. Le classi all’interno di un modulo hanno un’alta coesione tra loro

Questo si traduce nel principio **Common closure principle.** Quando noi modifichiamo una parte del nostro codice, noi andiamo a toccare solo un modulo, quindi la modifica sarà circoscritta solo in quel modulo, senza andare a toccare a cascata tutto il resto. E, visto il principio di prima della coesione, se modifico una parte del package, allora modifico tutte le classi all’interno del package. Il nostro package dovrà comportarsi come se fosse un grande oggetto.

Infine, dobbiamo essere bravi a gestire le dipendenze del nostro sistema. **Acyclic dependencies principle.** Quando dobbiamo andare a descrivere le dipendenze del nostro sistema, queste dovranno andare a formare un grafo **ACICLICO** dato che altrimenti rischieremmo che fare un cambio, vorrebbe dire cambiare tutte le classi nel ciclo. Se il componente A usa il componente B e viceversa, questo comporta un grosso problema. Se ci fosse un ciclo, non sapremmo da dove partire per capire se il nostro sistema funziona.

Per risolvere il problema dobbiamo rompere il ciclo e possiamo applicare il **dependency inversion principle.** Per applicare questa tecnica, utilizziamo il pattern adapter.

A questo punto, il sistema sarà organizzato in livelli o strati. La gerarchia è data dalla relazione di uso. Nella rappresentazione a layer, più siamo in alto più abbiamo un alto livello di astrazione. Nella nostra architettura dovremmo avere dipendenze solo su “layer” diversi. Perché, se siamo allo stesso livello d’astrazione e non siamo uniti, allora vuol dire che siamo separati e abbiamo funzionalità diverse. Inoltre, tecnicamente non dovremmo avere dipendenze DIRETTE saltando interi livelli

La differenza tra un programma e un tool è che un tool ha delle funzionalità per altri programmi e non per gli utenti.

Un altro stile architetturale è quello Pipes & filters. Questi sistemi usano in pratica dei connettori che vanno a formare una pipeline dove abbiamo che l’output di una parte è l’input di un altro. Inoltre, i connettori ci possono dare la possibilità di splittare e unire gli output. Questi connettori rappresentano un flusso di dati (oltre che delle dipendenze).

I costi di manutenzione sono spesso più grandi di quelli di sviluppo.

Possiamo misurare questa complessità. Abbiamo complessità dei dati (spaziale), quella di elaborazione (computazionale e temporale). Per misurare la complessità di un software utilizziamo delle metriche dove la più importante è quella di McCabe che serve a misurare la complessità di un algoritmo (metodo).

Altre metriche comuni sono LOC o SLOC (numero di righe di codice), o NCNB (numero di righe di codice non bianche). Il numero di commenti rispetto al numero di codice dovrebbe essere all’incirca il 30%.

Guardiamo delle metriche OBJ-Oriented. WMC, DIT, NOC, CBO, RFC, LCOM.

WMC indichiamo la complessità di una classe in base alla somma delle complessità dei vari metodi.

Immagine che contiene testo, Carattere, bianco, algebra

Descrizione generata automaticamente

La forma più semplice di questa metrica è il numero di metodi per classi.

Se valutiamo il numero di metodi di una sottoclasse, dobbiamo tenere conto solo di quelli suoi en on di quelli del padre. Questo numero dovrebbe essere tenuto sotto controllo all’incirca sotto i 35 ecc.

DIT è la profondità dell’albero di tipo (sottoclassi). Se una classe ha molti livelli di profondità (dista molto dalla sua radice), allora è molto specializzata e quindi difficilmente verrà riutilizzata. Questo numero viene di solito tenuto sotto 5.

NOC indica il numero di figli. Quante classi derivo dalla mia classe. Avere tante sottoclassi vuol dire che stiamo riutilizzando molto la classe (positivo), tuttavia vuol dire che la classe base dobbiamo testarla molto di più dato che dovrà essere testata da tutti i figli. Un’altra cosa è che, se abbiamo tanti figli, allora potrebbe voler dire che la classe base non sia stata progetta bene.

CBO accoppiamento tra oggetti. Andiamo a vedere il suo FAN IN (classi che uso) e FAN OUT (classi che mi usano). Come CBO = FAN OUT – FAN IN. Una classe intermediaria, avrà un CBO molto basso. Tendenzialmente utilizziamo un CBO > 14.

RFC è una metrica dinamica che va’ a vedere quanto le implementazioni sono legate tra loro. Quanti metodi di altri oggetti vengono invocati a run time. Un grande RFC comporta: testing e debugging più complicato, maggiore complessità dell’oggetto e difficoltà di comprensione dei metodi. RFC = NLM (numero di metodi della mia classe) + NRM (numero di chiamate di metodi di classi esterne).

LCOM lack of cohesion in methods1. C ha M1…Mm metodi. Definisco con Ik numero di istanze di variabili del metodo Mk. Definisco P come l’insieme di tutte le coppie che non si intersecano e Q come l’insieme di tutte le coppie di attributi che vengono utilizzate assieme.

Se |P| > |Q| allora LCOM = |P| - |Q| altrimenti LCOM = 0. Se |P| cresce vuol dire che ho molta coesione.

21/05/2024

Testing

I **black box test** sono basati su input e output (sono a scatola chiusa e non vediamo cosa c’è dentro). Dunque, ci basiamo solo su quello che vede in un certo senso l’utente.

Il **white box test** è un test che è basato sulla conoscenza del codice.

**Validazione** controlla che il nostro box fa quello che ci si aspetta e quello che si aspetta soprattutto l’utente.

**Verifica** della correttezza ci dice se a livello interno i meccanismi sono corretti.

In entrambi i casi stiamo parlando di fare test.

Abbiamo vari livelli di testing.

**Immagine che contiene testo, schermata, Carattere, ricevuta

Descrizione generata automaticamente**

L’unico test white box è quello Unit test. Possiamo distinguere due spazi da esplorare quello di **input** e quello dei **comportamenti.** Il test funzionale deve essere scelto in maniera tale da testare alcuni particolari (non nel complesso) comportamenti del sistemi. Questi test sono comodi se sono molto automatizzati.

Dobbiamo scegliere i test in modo che siano pochi ma che diano la possibilità di sentire che il sistema è stato testato a sufficienza.

Nel black box testing gli input vengono scelti in maniera tale da testare il sistema secondo le specifiche date. Per quanto riguarda il white box testing gli input vengono scelti in base al codice. Fa parte del black box testing il **beta testing.**

Verifica formale corrisponde alla verifica matematica e logica per dimostrare che formalmente il nostro sistema funziona. Per fare queste verifiche sono nati dei tool del tipo Proof assistant e Model Checker.

(Su Java facciamo la verifica del bytecode per evitare intrusioni interne e per fare una seconda verifica)

Stabiliamo un criterio per scegliere i campioni da provare. Grazie a ciò possiamo rendere il nostro sistema sistematico. Il criterio di **Copertura** ci fa limitare il numero di test che abbiamo al numero di righe che abbiamo (un test per riga ad esempio).

L’environment corrisponde a tutto ciò che sta attorno al nostro sistema. Per fare dei test al meglio dobbiamo garantire che tutto l’environment sia **stabile.** Ciò viene fatto “simulando” l’environment tramite dei dati che noi sappiamo essere al 100% funzionali (non ci appoggiamo ad un codice di una funzione ma ci affidiamo ad un “fantoccio” che dà sempre risposte corrette).

SI parla di Drivers e Stubs. Al posto di stubs si utilizza spesso il termine “mock objects”. Tramite questo sistema riusciamo anche a capire se la nostra classe una volta immessa nel sistema, interagisce correttamente con tutte le classi. Questa parte è fondamentale per lo unit test, dove andiamo a testare solamente una classe singola. Una volta immessa dicevamo, abbiamo bisogno di effettuare un test di **integrazione**. E’ più semplice fare dei test dal basso verso l’alto.

Per automatizzare lo unit test abbiamo vari tools. Un esempio può essere JUnit e TestNG per lanciare i test. Inoltre, per generare dei test in maniera automatica abbiamo Randoop, EvoSuite, Ttuples.

23/05/2024

Tramite assertJ possiamo inserire più clausole nello stesso rigo anziché scriverle separate.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Un altro package che possiamo utilizzare con JavaU è MockObjects (oggetti finti). Simuliamo quindi il comportamento delle classi con cui la classe che testo lavora. Tutto questo viene fatto per assicurarci che il test è eseguito sula classe che è isolata.

Tipologie di test doubles

**Stubs** Rispondono solo a particolari tipi di chiamate e dà risposte “preconfezionate”.

**Spies** è uno stubs che conserva però le informazioni e quali invocazioni sono state ricevute. In particolare questo è una primitiva dei **MockObj**

**Mock objects** sono programmati per sapere quante volte e in che ordine sono stati chiamati i suoi metodi durante il test.

Immagine che contiene linea, diagramma, schermata, Rettangolo

Descrizione generata automaticamente

Stub testa come agisce la classe. Mock testa che la classe funziona correttamente in relazione con le altre classi.

Il test black box è sempre una garanzia dato che ragioniamo sugli input e gli output e non sul codice che può cambiare.

I test black box potrebbero mancare degli errori che magari non testano determinate funzionalità.

Per il WBT abbiamo o uno **statement coverege** (copre tutte le righe) o un **branch coverege** (copre tutti i percorsi). Questi sono i fondamentali ma ne abbiamo alcuni anche più complicati com **AP Du Pairs ecc.**

L’equivalence partitions consiste nel prendere un dato rappresentativo per un range di valori. Questo discorso lo facciamo per tutti i parametri e proviamo ogni categoria di dato. (per gli int magari prendiamo 0 un valore negativo e un positivo, per gli enum li usaimo tutti ecc.). Non dobbiamo considerare solo i parametri di una classe, dobbiamo considerare anche gli attributi della classe stessa o le variabili globali, quindi in generale dobbiamo provare tutto l’ambiente che circonda. A questo punto otteniamo il Test Coverage Items. Cioè vogliamo vedere tutti i goal che potrebbe raggiungere un input.

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Ad ognuno di questi diamo un EPN.

Immagine che contiene testo, schermata, Carattere, ricevuta

Descrizione generata automaticamente