Progetto Software

Processo di Design del Software (1)

Il Modello Waterfall

Il modello Waterfall è uno dei modelli di sviluppo software più tradizionali e strutturati. Questo modello segue un approccio sequenziale e lineare, dove ogni fase del processo di sviluppo deve essere completata prima di passare alla successiva.
Le fasi tipiche del modello Waterfall sono:

1. **Requisiti**: In questa fase, vengono raccolti e documentati tutti i requisiti del sistema. Gli stakeholder definiscono ciò che il software deve fare. Vengono identificati i KPIs, che possono essere utilizzati per monitorare vari aspetti del progetto e garantire che i requisiti e gli obiettivi siano raggiunti.
2. **Progettazione**: In base ai requisiti raccolti, viene progettata l'architettura del sistema. Questo include sia la progettazione dell'architettura software che quella dei dettagli di implementazione.
3. **Implementazione**: Durante questa fase, i programmatori scrivono il codice necessario per implementare il design del software.
4. **Verifica (Testing)**: Una volta che il software è stato implementato, viene rigorosamente testato per assicurarsi che funzioni correttamente e che rispetti i requisiti specificati.
5. **Integrazione e Manutenzione**: Dopo il testing, il software viene distribuito e integrato nel contesto operativo. Successivamente, si procede alla manutenzione, che può includere la correzione di bug e l'aggiornamento del software per adattarlo a nuovi requisiti o ambienti.

Customer

Il Customer è l'entità che ha diretto interesse nello sviluppo finale di un progetto. Infatti quando un progetto viene istanziato viene istanziato specificatamente per

1. Progetti interni di ricerca (R&D)
2. Prodotti per l'ampio mercato (esempio MS Windows)
3. Prodotti Time-to-market agreed, cioè un prodotto che deve essere sviluppato e testato entro una quantità di tempo specificata tra Azienda e Cliente-

Qualità del prodotto

La qualità del prodotto si riferisce alla sua funzionalità, maggiori sono le sue funzionalità maggiore sarà il suo costo di mercato.

Qualità del processo

La qualità del processo si riferisce a come il software è stato sviluppato relativamente ad un specifico dominio, uno specifico team.

Valutare la qualità e la funzionalità di un prodotto

La qualità e la funzionalità di un prodotto si basa su diversi fattori, fra questi:

1. La correttezza: la correttezza di un prodotto dipende strettamente dalle sue funzionalità che sono negoziate direttamente con il cliente
2. Facilità d'uso: la facilità d'uso ha un rapporto stretto con l'UX - User Experience (ciò che vedo è comprensibile?)
3. Performance: la performance di un prodotto è un fattore molto importante, permette di valutare se sotto diversi carichi di lavoro risulta efficiente, scalabile
4. Dependability: il fattore affidabilità è importante, un cliente non comprerà mai un prodotto non sicuro

Affidabilità di un prodotto
La "dependability" (affidabilità) di un prodotto è un concetto ampio che si riferisce alla capacità del prodotto di funzionare correttamente e in modo consistente nel tempo, senza guasti, sotto condizioni specificate. Nel contesto dei sistemi informatici e software, la dependability è cruciale perché influisce direttamente sulla soddisfazione dell'utente, sulla sicurezza e sull'efficacia complessiva del sistema. Quando si dice che per specifici domini la dependability può fare "overlap" (sovrapposizione), si sta indicando che le diverse componenti della dependability (affidabilità, disponibilità, sicurezza, ecc.) possono influenzarsi e intersecarsi tra loro. In altre parole, i vari aspetti della dependability non sono completamente separati e indipendenti, ma spesso si sovrappongono e si interrelaziona.

*Un **IFIP Working Group** è un gruppo di lavoro creato sotto l'egida dell'**International Federation for Information Processing (IFIP)**, un'organizzazione internazionale dedicata alla promozione e allo sviluppo delle scienze e tecnologie dell'informazione.*

Ad ogni modo l'IFIP ha identificato 3 elementi principali per valutare la dependability: attributi, minacce e means (che voglia dire Burgio non lo so… lo scopriremo)

Proprietà non funzionali
Le proprietà non funzionali di un sistema sono caratteristiche che definiscono come un sistema opera piuttosto che ciò che il sistema fa e sono la verificabilità, riusabilità, portabilità, interoperabilità e mantenimento

Come iniziare un progetto
Per poter iniziare un progetto abbiamo bisogno di sistematizzare i processi, tipicamente lo facciamo con il divide et impera, splittando i problemi in più sottoproblemi. In modo particolare:

Il ciclo di vita di un processo - Modello Waterfall
Architettura di un sistema - MicroServizi
Architettura di sistema interno - MVC o MVVM
Test e Deploy - CI / CD

Ancor di più nel particolare per sistematizzare il processo di creazione potremmo utilizzare due approcci, top down (visualizzazione completa del progetto che viene poi diviso in più componenti) o bottom up (si sviluppano i componenti e poi si integrano nel sistema)

Costi
Il costo di un progetto e di tutte le fasi che lo riguardano è importante.
In breve, la manutenibilità di un progetto, componente principale, prende una grande fetta dei costi. In più aggiungiamo, sviluppatori, supporto, HR, elettricità, aumento delle performance, bug fixing…

Problemi principali
I sistemi diventano sempre più complessi, il software deve sempre essere aggiornato, tecnologie obsolete non possono essere sostituite ed il "time-to-market" è una politica sempre più applicata fra le aziende

Modellare il processo
Modellare il processo di progettazione significa strutturare l'intero flusso di sviluppo. Abbiamo diverse possibilità per strutturare il flusso di sviluppo che dipende dal tipo di tecnologie che vogliamo usare, il tempo a disposizione, se ci sono procedure standard per campi specifici.

Esempio di un pattern
1. Specifiche

2. Design
3. Implementazione SW e HW
4. Integrazione
5. Test
6. Deploy (Distribuzione)
7. Maintenance / Aftermarket (AM)
8.

Andiamo ad esaminare ogni punto

1. **Requisiti e specifiche**

Il processo inizia coinvolgendo gli impegni e sommergendo di domande riguardanti cosa dovrebbe fare il sistema (funzionale) e come interagisce con l'utente (non funzionale), oltre a considerare l'ambiente o l'ecosistema (hardware vs. software). È essenziale verificare se esistono basi di codice legacy, processi legacy o fornitori legacy, e cercare di immaginare possibili problemi per la manutenzione e possibili sviluppi futuri per il mercato post-vendita. L'esperienza annuale in questo settore e nei domini applicativi specifici è molto utile, e spesso sono coinvolte persone specializzate come analisti di business, ingegneri delle prestazioni e ingegneri dei requisiti.
Il risultato di questo processo è un insieme di documenti che coprono chiaramente tutti questi argomenti e identificano indicatori misurabili (KPIs) per valutare se il sistema sta funzionando correttamente. Questo influenza direttamente la fase di test e definisce il valore del sistema per il cliente. In aggiunta, si deve firmare un accordo con il cliente, stabilendo il prezzo del nostro servizio o software e specificando chiaramente cosa faremo e cosa no, oltre a cosa dovrà fare il cliente. Ad esempio, chi sarà responsabile della manutenzione dell'infrastruttura server e chi pagherà le eventuali licenze annuali. In questa fase, il cliente sarà ancora collaborativo, ma inizierà a diventare ansioso perché desidera vedere il prodotto completato. È importante non affrettarsi, cercando comunque di essere il più rapidi possibile. Ricorda che, in Italia, il 95% delle aziende sono PMI/SMEs e i Project Manager spesso hanno una limitata esperienza nella progettazione software. I clienti potrebbero pensare di conoscere qualcuno che potrebbe fare il lavoro al posto nostro. Questa fase è cruciale perché qui stiamo facendo promesse. Un tipico scenario è che, dopo 6-12 mesi, il cliente possa dire: "Il sistema non funziona a meno che non facciate anche questo e quest'altro. Se il sistema non funziona, ovviamente non vi pagherò".

2. **Design**

Per fare uno sketch di come possa essere un sistema funzionante dobbiamo iniziare dal principio - scegliere linguaggio appropriato, framework, hardware, definire i protocolli, definire i metodi di comunicazione tra team, git ad esempio.

3. **Implementazione**

Dopo aver creato un buon design, bisogna implementare tutte le funzionalità.

- CRUD - create, read, update, e delete di dati
- consultare possibili HTTPs
- Renderlo visibile via web

4. **Integrazione**

Il processo di riunione delle diverse parti del nostro software, chiamate componenti/moduli/servizi a seconda dell'architettura, è ottimizzato dai principi del divide-et-impera. Seguendo adeguati pattern architetturali durante la fase di progettazione, possiamo utilizzare strumenti e metodologie specifiche. Ad esempio, i requisiti e le specifiche indicano la necessità di un server web con molti endpoint, in grado di funzionare con altri servizi esistenti, sfruttando le licenze per MS Azure e AWS. Il nostro flusso di lavoro prevede molte operazioni di lettura/scrittura senza modificare i dati. Per la

progettazione, seguiamo il pattern architetturale dei micro-servizi, distribuiamo su Azure FAAS e AWS Lambdas utilizzando c#/dotNet e Python, e adottiamo MongoDB o, in generale, database noSQL.

5. **Testing**

Verifichiamo che le specifiche e i requisiti sia funzionali che non siano stati implementati.

6. **Deploy**

L'integrazione continua e la consegna continua (CI/CD) utilizzano metodologie e strumenti ben noti che semplificano il processo partendo dal framework che vogliamo utilizzare, spesso integrandosi con la fase di testing. L'ideale sarebbe un sistema in cui basta un clic perché il framework faccia tutto, ma non è sempre così. Spesso i progetti sono più piccoli e non richiedono la complessità di un framework completo.

7. **Manutenzione**

Il processo di modifica o estensione del software dopo la prima versione, concordata con il cliente, include sempre il debug per un periodo di 6, 12 o 24 mesi, a seconda del contratto. Potreste anche concordare con il cliente la vendita di Man-Month per lavorare su funzionalità aggiuntive, spesso da definire successivamente. Non copriremo questo aspetto, poiché è molto specifico per ogni cliente o progetto. Tuttavia, posso darti alcune regole d'oro: il cliente spesso non sa cosa vuole, nemmeno nella fase di progettazione; l'appetito viene mangiando, quindi mostrategli il pieno potenziale del vostro software per stimolare il loro interesse; questo non riguarda solo il vostro software, ma anche la mentalità del cliente.

Requisiti e Specifiche (2)

Dalle Specifiche ai Requisiti

Le specifiche sono un contratto tra noi e il cliente. Inizialmente definiamo cosa deve fare il sistema e deriviamo un insieme di requisiti funzionali. Per ogni fase del flusso di sviluppo, creiamo requisiti più dettagliati e tecnici, comprendendo specifiche e requisiti di sistema, moduli e componenti, dai quali derivano i test.
Questi accordi sono tra progettisti e sviluppatori, il cliente è coinvolto solo se ha vincoli o competenze specifiche.

Cosa vs Come

È importante capire che le specifiche definiscono cosa il sistema deve fare (requisiti funzionali), mentre l'implementazione descrive come il sistema lo fa (requisiti di sistema/modulo/componente), solitamente con un approccio top-down. Ogni fase del processo di sviluppo definisce un contratto tra le sue sottoparti, come protocolli middleware nella fase di progettazione del sistema o interfacce Java nei sottocomponenti.

Ad esempio, per l'autenticazione degli utenti, i requisiti indicano che gli utenti devono autenticarsi nel sistema, e l'implementazione specifica come inseriscono i dati in un form web e cliccano su "submit", con dettagli che diventano sempre più specifici.

Livello più alto: Requisiti di sistema

I requisiti di alto livello descrivono le funzionalità del sistema e includono requisiti funzionali (come il sistema reagisce agli input e produce output) e non funzionali (prestazioni, affidabilità, ecc.). I requisiti specifici del dominio possono essere impliciti e non sempre indicati dal cliente, come nei sistemi in tempo reale che considerano il tempo di esecuzione peggiore anziché quello medio.

FURPS Model
I requisiti sono classificati secondo il modello FURPS: Funzionalità, Usabilità, Affidabilità, Prestazioni e Supportabilità.
Comunicare i requisiti è la parte più difficile della costruzione di un sistema software, poiché spesso neanche i clienti sanno esattamente cosa vogliono.
Le specifiche devono essere chiare, non ambigue, coerenti e complete, e devono essere costruite incrementando la collaborazione con il cliente.

Comunicare i requisiti
Per comunicare i requisiti, utilizziamo formalismi come macchine a stati finiti, diagrammi di casi d'uso e diagrammi E/R, che possono variare in base al dominio applicativo.
Le specifiche possono essere:

- operative (cosa deve fare il sistema),
- descrittive (come è fatto il sistema)
- tecniche (come è implementato il sistema), spesso a causa di applicazioni esistenti o codice legacy.
-

Il livello di prontezza tecnologica (TRL)
Il livello di prontezza tecnologica (TRL) è una scala da 1 a 9 che stima la maturità di una tecnologia, sviluppata dalla NASA negli anni '70 per le missioni spaziali.

Valutazione del TRL del Software

TRL	Utilizzo NASA	Unione Europea
1	Principi di base osservati e riportati	Principi di base osservati
2	Concetto tecnologico e/o applicazione formulata	Concetto tecnologico formulato
3	Funzione critica analitica ed esperimentale e/o proof-of-concept caratteristico	Proof-of-concept sperimentale
4	Validazione del componente e/o breadboard in ambiente di laboratorio	Tecnologia validata in laboratorio
5	Validazione del componente e/o breadboard in ambiente rilevante	Tecnologia validata in ambiente rilevante (ambiente industrialmente rilevante nel caso delle tecnologie abilitanti chiave)
6	Dimostrazione del modello o prototipo del sistema/sottosistema in un ambiente rilevante (a terra o nello spazio)	Tecnologia dimostrata in ambiente rilevante (ambiente industrialmente rilevante nel caso delle tecnologie abilitanti chiave)
7	Dimostrazione del prototipo del sistema in un ambiente spaziale	Dimostrazione del prototipo del sistema in ambiente operativo
8	Sistema effettivo completato e "qualificato per il volo" attraverso test e dimostrazioni (a terra o nello spazio)	Sistema completo e qualificato
9	Sistema effettivo "provato in volo" attraverso operazioni di missione di successo	Sistema effettivo provato in ambiente operativo (produzione competitiva nel caso delle tecnologie abilitanti chiave; o nello spazio)

Specifica dei Requisiti Software (SRS)
Lo standard IEEE 830 del 1998 per le specifiche dei requisiti software si utilizza solo per definire i requisiti e non si occupa dell'implementazione o della qualità del codice.

La SRS deve essere:

- Corretta, in modo da modellare correttamente il comportamento del sistema.

- Non ambigua, utilizzando la notazione e la terminologia corrette e un dizionario dei dati (DD) per evitare ambiguità nel linguaggio.
- Completa, coprendo tutti i requisiti, sia funzionali che non funzionali, e definendo una risposta per ogni possibile stato o input del sistema.
- Coerente, evitando conflitti tra i requisiti, spesso derivanti da contribuenti che non comunicano tra loro. Definire un programma e un DD è il primo passo per assicurare la coerenza.
- Ordinata, organizzando i requisiti per priorità e livello di astrazione, assegnando un identificatore e una priorità, e descrivendo i requisiti in dettaglio.
- Verificabile, assegnando metriche e indicatori chiave di prestazione (KPI) ai requisiti, ad esempio "tempo di risposta inferiore a 7 secondi nel 90% dei casi".
- Modificabile, dato che i requisiti possono cambiare, il documento deve essere ben organizzato e i requisiti non devono essere ridondanti.
- Tracciabile, per capire da dove provengono i requisiti. I requisiti di alto livello (ad esempio, funzionali) devono essere approvati dal cliente, mentre i requisiti tecnici sono generalmente concordati tra il team di sviluppo.

In sintesi, la SRS è uno strumento essenziale per definire chiaramente e organizzare i requisiti di un progetto software, assicurando che siano compl**eti, coerenti e verificabili, e che possano essere modificati e tracciati nel corso del progetto.**

Struttura di una Specifica dei Requisiti Software (SRS)

La struttura di una SRS include una serie di sezioni principali. Inizialmente, c'è un'introduzione che copre lo scopo del documento, il campo di applicazione, le definizioni, gli acronimi, le abbreviazioni, le referenze e una panoramica della struttura del documento. Successivamente, c'è una descrizione generale che fornisce una prospettiva del prodotto, le funzioni del prodotto, le caratteristiche degli utenti, i vincoli generali, le assunzioni e le dipendenze, e la distribuzione dei requisiti. La terza parte è dedicata ai requisiti, comprendendo l'interfaccia di sistema e front end, i requisiti funzionali e non funzionali. Infine, ci sono eventuali appendici e un indice.

1. Introduzione

Nell'introduzione, si chiariscono gli obiettivi del documento e il pubblico di riferimento, che può essere composto da persone tecniche o non tecniche. Viene anche indicato il nome del prodotto, gli obiettivi, i benefici, ciò che si intende risolvere e ciò che non si intende risolvere al momento, nonché le possibili future evoluzioni. Si elencano definizioni, acronimi e abbreviazioni e si forniscono le referenze, che possono essere inserite anche in appendice. Infine, si offre una panoramica della struttura del documento, spesso suddivisa in "as-is" e "to-be" in caso di migrazioni.

2. Descrizione Generale

La descrizione generale comprende le principali caratteristiche del sistema e le regole a cui deve aderire.

Si descrive la prospettiva del prodotto, inclusi eventuali prodotti simili, le interfacce di sistema, utente, hardware e software, l'interfaccia di comunicazione, i requisiti di memoria, l'inizializzazione, il backup, il recupero, l'installazione e la configurazione. Si delineano le funzioni del prodotto, le caratteristiche degli utenti e le competenze necessarie per utilizzare il sistema. Vengono identificati i vincoli generali, le assunzioni e le dipendenze esistenti, e si specifica la distribuzione dei requisiti, indicando chi si occuperà di determinati aspetti.

3. Requisiti (Funzionali e Non Funzionali)

La sezione sui requisiti affina quanto descritto nella parte 2.1 e 2.2, specificando i formati di input e output e i protocolli. Ogni requisito funzionale ha una tabella o una riga dedicata che specifica il contesto, la funzionalità e le entità/attori coinvolti, l'input e l'output, la descrizione dei dati, le operazioni da eseguire, le azioni in caso di errori e gli output previsti.

Esempio

USR_568	User account	Update birth date
Input	User ID, the new birth date	
Description	User wants to update the birth date with a new one; the system updates it, and, in case user does not exist, it creates a new user with empty fields	
Error messages	In case birth date is a future date, or it is empty, an error with a meaningful message is shown. The error is shown in user language, as described in requirement LOCAL_45	
Output	Information is updated in the persistent storage; all views that contain the datum are automatically updated, or get the fresh new datum when manually updated.	

I requisiti non funzionali includono aspetti come le prestazioni, il database e lo storage, i vincoli generali, gli attributi del sistema e altri requisiti specifici.

Requisiti di Prestazione

I requisiti di prestazione stabiliscono quanti accessi concorrenti il sistema deve supportare, quanti utenti devono essere gestiti e quanti flussi di lavoro asincroni devono essere operativi contemporaneamente.

Database e Storage

Per quanto riguarda il database e lo storage, si verificano eventuali vincoli tecnici esistenti.

Vincoli Generali

I vincoli generali includono l'adesione agli standard, le limitazioni hardware o del sistema operativo e l'uso di linguaggi specifici.

Attributi del Sistema

Gli attributi del sistema comprendono vari aspetti come affidabilità, accessibilità, sicurezza, manutenibilità e portabilità. Ad esempio, per l'affidabilità, si considera il tempo di inattività e i guasti tollerabili. Per l'accessibilità, si valuta il tempo di backup e di recupero in caso di guasti. La sicurezza, la manutenibilità e la portabilità sono anche elementi chiave.

Altri Requisiti

Altri requisiti includono le linee di sviluppo, staging e produzione, le pipeline CI/CD e le possibili versioni pubbliche. Vengono inoltre definite le regole di visibilità e accesso per i gruppi di utenti e la scalabilità del sistema in termini di prestazioni, numero di utenti e costi associati.

Appendice e Indice

Tutto ciò che non è essenziale viene inserito in appendice, come la struttura e il protocollo di AWS (usato da Burgio e Hypert Lab), se pertinente. Le referenze sono tipicamente incluse in questa sezione. L'indice comprende l'elenco delle sezioni e dei termini utilizzati nel documento.

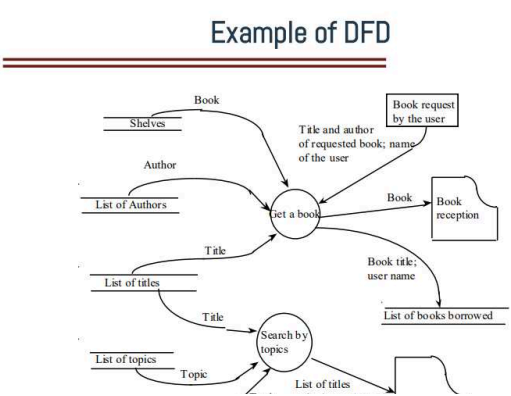
Documentazione, Note e Strumenti (3)

In genere, distinguiamo tra diversi tipi di diagrammi.

I diagrammi operativi includono i diagrammi di flusso dei dati, UML, modelli come le macchine a stati finiti e le reti di Petri. I diagrammi descrittivi/strutturali sono quelli basati sul modello Entità-Relazione, che si ispira all'analisi e progettazione di entità nei database. I diagrammi strutturali, che sono standard UML, comprendono casi d'uso, notazioni per classi, oggetti, pacchetti e componenti, derivati dalla programmazione orientata agli oggetti. Infine, i diagrammi comportamentali iincludono i diagrammi di sequenza, di stato e di attività. Tuttavia, non possiamo spiegare tutto in questo ordine: iniziamo dalle specifiche, poi dalla progettazione del sistema e infine dall'implementazione. UML ha presentazioni dedicate per questi argomenti.

I Diagrammi di Flusso dei Dati (DFD)

I diagrammi di flusso dei dati descrivono le funzionalità (nodi) e gli archi di dati, sia in ingresso che in uscita. Vengono mostrati in bianco e nero, ma si consiglia di utilizzare forme e colori per migliorare la comunicazione. Ogni funzionalità può avere un colore diverso e le linee possono essere tratteggiate o più spesse a seconda delle necessità.



I Diagrammi di Flusso dei Dati (DFD) non sono standardizzati.

Vantaggi: sono estremamente semplici e utilizzati da tutti.

Svantaggi: sono informali e non standardizzati. Di solito, utilizzo una variante con simboli aggiuntivi. Inoltre, non sono operativi, quindi non possono specificare flussi di controllo come condizioni (se, oppure, switch, ecc.).

Macchine a Stati Finiti / Automata

Modellare sistemi con stato: un esempio

Ad esempio, un ascensore reagisce a diversi eventi:

- Di solito è in stato di inattività.
- Se premi il pulsante, la porta si apre.
- Se selezioni un piano, le porte si chiudono.
- Poi, raggiunge il piano (con controllo della velocità).
- Infine, apre la porta, che si richiude dopo X secondi.

Questo comportamento è controllato da una macchina a stati finiti.

Problema esempio delle macchine a stati finiti:

- Identificare sequenze pari di una (anche vuota), seguite da una o più o nessuna b, e terminate con c.

Data un alfabeto che modella un insieme di input e regole di validazione per generare una sequenza (o parole), definiamo una macchina a stati finiti (FSA) come segue:

- **S**: un insieme non vuoto di stati.
- **s0** ∈ **S**: stato iniziale.
- **Sf** ⊆ **S**: insieme degli stati finali.
- **t**: funzione di transizione degli stati, t: S x V → S, dove V è l'insieme degli input.

In sintesi, una macchina a stati finiti è definita da un insieme di stati, uno stato iniziale, un insieme di stati finali e una funzione che descrive come si passa da uno stato all'altro in base agli input ricevuti.Fino ad ora, abbiamo visto macchine che possono riconoscere una parola di un linguaggio. Tuttavia, è anche possibile che una macchina produca un output.

La macchina di Mealy è un tipo di macchina a stati finiti che genera un output quando attraversa un arco. In altre parole, l'output è prodotto in base all'input e allo stato corrente della macchina. Questo tipo di macchina è definito da un insieme finito di simboli di input, un insieme finito di simboli di output, un insieme finito di stati, una funzione di transizione degli stati che mappa l'input e lo stato corrente all stato successivo, e una funzione di output che mappa l'input e lo stato corrente all'output.

D'altra parte, la macchina di Moore produce un output basato solo sullo stato in cui si trova, indipendentemente dall'input ricevuto. Anche in questo caso, la macchina è definita da un insieme finito di simboli di input e output, un insieme finito di stati, una funzione di transizione degli stati che mappa l'input e lo stato corrente allo stato successivo, e una funzione di output che mappa solo lo stato corrente all'output.

Qual'è la differenza?

Matematicamente, le macchine di Mealy e Moore sono equivalenti: una può essere trasformata nell'altra. Tuttavia, ci sono delle differenze pratiche:

La macchina di Mealy può produrre output diversi per input e transizioni differenti, il che può richiedere meno stati, perché se l'output dipende dagli input, si può aggiungere un arco alla macchina.Al contrario, la macchina di Moore tende a mantenere l'output stabile per ogni stato, il che può comportare la necessità di più stati se l'output dipende dall'input e non solo dallo stato.

Automatizzare il processo di produzione degli automi

Alla fine, è necessario solo modellare la grammatica e poi creare gli automi. Esistono diversi strumenti per supportare il design, come Matlab Stateflow e UML. Inoltre, ci sono interpreti di grammatica che aiutano a scrivere il codice per le macchine a stati finiti, come GNU Bison di FSF, incluso in GCC, e YACC (Yet Another Compiler-Compiler).

Per una macchina a stati finiti, definiamo:

- **I**: insieme finito di simboli di input.
- **O**: insieme finito di simboli di output.
- **S**: insieme finito di stati.
- **mfn**: funzione della macchina che mappa input e stati agli output.
- **sfn**: funzione di stato che mappa input e stati ai nuovi stati.

Reti di Petri - definizione

Le reti di Petri modellano il comportamento delle applicazioni tramite un grafo bipartito diretto. Le transizioni, rappresentate da barre, sono attivate da eventi, mentre i posti, rappresentati da cerchi, indicano le condizioni. Gli archi collegano solo posti a transizioni (o viceversa) e specificano quali posti sono condizioni preliminari o successive per gli eventi. Ogni posto raccoglie token (puntini) che possono attivare un evento. Se più eventi vengono attivati nella stessa rete, quale scatta per primo è non deterministico. Le reti di Petri sono utili per modellare sistemi distribuiti e sistemi dinamici a eventi discreti.

Reti di Petri - formalismo

Una rete di Petri è formalmente definita da una quadrupla (S, T, W, M0), dove:

- **S** e **T** sono insiemi disgiunti.
- Nessun arco può collegare due stati o due transizioni tra loro.

Come funzionano:

- L'attivazione di una transizione **t** in una marcatura **M** consuma **W(s, t)** token da ciascuno dei posti di input e produce **W(t, s)** token in ciascuno dei posti di output.
- Una transizione è abilitata (può scattare) in **M** se ci sono abbastanza token nei suoi posti di input affinché il consumo sia possibile, cioè se e solo se per ogni **s**, **M(s) ≥ W(s, t)**.

Documentazione, Modello Unified Language (UML) (4)

Che cos'è UML?

UML (Unified Modeling Language) è nato nel 1994, è stato standardizzato nel 1998 e la versione ufficiale 2.0 è stata rilasciata nel 2005. È diventato un linguaggio di progettazione di fatto, basato su una notazione semi-standard che descrive in modo meta descrittivo le entità di un sistema software. Utilizza notazioni grafiche e supporta il principio del "divide et impera".UML è utile perché può modellare diversi livelli di astrazione e fasi di sviluppo, dalle specifiche fino alle singole classi. Funziona sia per approcci top-down che bottom-up ed è indipendente dal linguaggio di programmazione.

UML si suddivide in tre macro-aree:

1. **Entità** (struttura): classi, interfacce, comportamenti (macchine a stati finiti, interazioni con gli utenti), raggruppamenti e pacchettizzazione, notazioni e informazioni generali.

2. **Relazioni**: associazione, dipendenza, generalizzazione e implementazione.

3. **Diagrammi**: offrono diverse prospettive dello stesso oggetto/entità, permettendo di "vedere le cose" sotto una luce diversa.

I diagrammi standard di UML includono:

- Diagrammi strutturali: casi d'uso, notazioni per classi/oggetti/pacchetti/componenti (derivanti dalla programmazione orientata agli oggetti), distribuzione/componenti.

- Diagrammi comportamentali: diagrammi di sequenza, di stato e di attività.

Modellazione dei casi d'uso

La modellazione dei casi d'uso descrive l'interazione tra il sistema e altri attori, come utenti o altri sistemi esterni. Può essere rappresentata sia come un'immagine che come una tabella (o entrambe). È importante notare che i casi d'uso non sono requisiti del sistema, ma rappresentano il comportamento atteso e sono utili anche per i test funzionali e la verifica.

L'obiettivo è identificare chiaramente i confini del sistema (sia comportamentali che "fisici"), gli attori o macro-entità e gli scenari dei casi d'uso. La notazione grafica per ogni elemento può includere una breve descrizione, mantenendo il giusto livello di astrazione.

Il processo inizia con l'identificazione degli attori e la modellazione delle loro interazioni con il sistema. Gli attori sono esterni al sistema e, anche se possiamo modellarli internamente, non sono sotto il nostro controllo. Successivamente, si definiscono diversi scenari, che rappresentano istanze singole di un caso d'uso, descrivendo la sequenza di eventi che si verificano in uno scenario specifico. Gli scenari principali rappresentano situazioni in cui tutto funziona correttamente, mentre gli scenari secondari riguardano estensioni o errori.

Gli scenari devono definire le condizioni operative, che possono essere standardizzate, come nel settore automobilistico, specificando quando e in che condizioni il sistema non funziona. In genere, si specificano le precondizioni e le post-condizioni (cioè, i cambiamenti di stato), le garanzie da fornire e assumere (affidabilità, QoS, ecc.), e i trigger per gli eventi.

Le relazioni tra gli attori e i casi d'uso includono:

- **Associazione** tra attori e casi d'uso.
- **Generalizzazione** tra attori e casi d'uso, aggiungendo caratteristiche al genitore.
- **Inclusione** di casi d'uso (<<includes>>).
- **Estensione** di casi d'uso (<<extends>>).

La generalizzazione tra attori permette ai figli di partecipare a tutti i casi d'uso del genitore e aggiungerne di nuovi. La generalizzazione tra i casi d'uso consente di ridefinire o specializzare i passaggi e gli eventi esistenti. Le estensioni sono utilizzate per esprimere dipendenze, come ad esempio, prima si piazza la scommessa e poi si lanciano i dadi; teoricamente, il giocatore vede solo la funzionalità "lancia i dadi".

Come identificare i casi d'uso (e gli scenari)?

Esistono due approcci per identificare i casi d'uso: basato sugli attori e basato sui processi. Nel primo, si modella l'interazione per ogni attore, mentre nel secondo si identificano gli attori per ogni interazione.

I diagrammi dei casi d'uso sono il punto di partenza per i progettisti di sistemi, offrendo una buona approssimazione della dimensione e complessità del sistema. Possono anche essere utilizzati per scrivere guide utente. Questo processo è iterativo.

Diagrammi di sequenza

Questi diagrammi si concentrano sugli attori e sui dati che scambiano con il sistema, descrivendo l'interazione tramite messaggi, oggetti, ecc. Illustrano la cronologia di uno scenario. Gli oggetti, rappresentati come rettangoli, sono entità nel sistema (attori, moduli, classi, database) e la loro esistenza è rappresentata da una "lifeline". Il focus di controllo viene rappresentato come un rettangolo sulla lifeline, mostrando quando un oggetto blocca l'interazione in modo sincrono. Gli stimoli rappresentano chiamate o invocazioni e possono includere descrizioni brevi.

Messaggi e stimoli

I messaggi astratti rappresentano o attivano azioni come chiamate, risposte, invio di segnali, creazione o distruzione di oggetti. Possono essere aggregati in sequenze per modellare interazioni complesse o cambiamenti di stato. I tipi di messaggi includono costruttori, distruttori, lettura/consulta, aggiornamento, collaborazione e iterazione.

Entità con stato

I diagrammi di stato rappresentano il ciclo di vita di un'entità, solitamente classi/oggetti, tramite eventi, azioni, stati, transizioni e guardie. Gli stati sono definiti come un insieme coerente e significativo di attributi di un'entità che influenzano il suo comportamento. Le transizioni tra stati sono innescate da eventi e possono includere condizioni di guardia che devono essere soddisfatte. Le attività all'interno degli stati possono essere complesse, mentre le azioni sono più piccole e atomiche e alterano lo stato dell'oggetto.

Azioni

Le azioni in un diagramma di stato si suddividono in diverse categorie:

- **Entry**: eseguite non appena l'oggetto entra in un determinato stato (entry/nome azione).
- **Exit**: eseguite quando l'oggetto esce da uno stato a causa di una transizione (exit/nome azione).
- **Do**: eseguite mentre l'oggetto si trova in un certo stato (do/nome azione).

- **Include**: invocano una «submacchina», cioè un altro diagramma di stato.
- **Event**: azioni che si verificano in risposta a un evento o trigger.

Sequenza delle azioni

Quando un evento attiva una transizione:

1. Le attività in esecuzione vengono interrotte (preferibilmente in modo ordinato).
2. Si esegue l'azione di uscita dello stato precedente.
3. Si esegue l'azione associata all'evento.
4. Si esegue l'azione di ingresso del nuovo stato.
5. Si eseguono le azioni "Do" del nuovo stato.

Diagrammi di attività

Modellano il comportamento di qualsiasi entità nel sistema per ogni stato. Sono diagrammi di flusso "orientati agli oggetti" che includono dipendenze temporali e spaziali, e forniscono maggiori dettagli sulle attività (e sottoattività) che si verificano in ogni stato sotto determinati input. Gli stati di azione rappresentano attività atomiche che:

- Non possono essere divise.
- Non possono essere interrotte (preempted).
- Nel modello, avvengono istantaneamente.

Stati speciali

- **Stati di inizio/fine**.

Transizioni

- Avvengono quando uno stato di azione termina.

Diramazione/Fusione

- La diramazione si basa su condizioni, catturando la semantica IF-ELSE (rombo: uno-a-molti).
- La fusione raccoglie più percorsi in uno solo (rombo: molti-a-uno).

Importante

- Non è necessario modellare l'intera applicazione.
- Scegliere il giusto livello di astrazione aiuta a evitare complicazioni superflue.

Fork/Join

- Modella azioni concorrenti, implementate con thread/processi paralleli. Il punto di join rappresenta la sincronizzazione di tali processi.

Stati di sottoattività

- Raggruppano stati di attività in macro-funzionalità. Non sono atomici, possono essere interrotti e hanno un tempo di esecuzione non nullo.

Swimlanes

- Raggruppano le attività in aree, partizionando il diagramma di attività. Queste aree possono rappresentare casi d'uso, classi/oggetti, componenti, unità aziendali o ruoli.

Dal diagramma di attività ai diagrammi di sequenza

- I diagrammi di attività identificano scenari per i quali possiamo (e dobbiamo) scrivere un diagramma di sequenza. Le aree possono essere utilizzate per modellare scenari/casi d'uso.

System Architecture and Design (5)

Progettazione del sistema

In questa fase si trasformano le specifiche del cliente in specifiche tecnologiche comprensibili per glii sviluppatori. L'output di questa fase è l'architettura del sistema, che prevede l'identificazione di una serie di moduli, ognuno con una funzionalità specifica, e la descrizione delle loro interazioni (contratti, prototipi, interfacce OOP, endpoint web, header C/C++).

Ingredienti

1. **Decomposizione**:
    - Suddividere il sistema in sottosistemi indipendenti.
    - Suddividere ulteriormente in moduli e sottomoduli, ciascuno con un servizio specifico.
    - Componenti: unità di implementazione base (es. librerie Java).
2. **Identificazione e assegnazione del controllo**:
    - Chi fa cosa? Identificare componenti attivi e passivi, e i processi/thread coinvolti.
3. **Moduli**:
    - Raggruppano funzionalità correlate.
    - È necessario identificare chiaramente le interfacce verso altri moduli/mondo esterno e le sotto modulazioni.
4. **Relazioni tra moduli**:
    - I moduli espongono servizi utilizzati da altri moduli.
    - I moduli possono dipendere da altri moduli per seguire un diagramma di sequenza per un caso d'uso specifico.

Strategie di partizionamento

- **Top-down**: dalle specifiche ai servizi, ai moduli, ai componenti.
- **Bottom-up**: centrato su strutture dati/funzionalità, utile se si dispone già di un framework o di una base di codice.
- **Misto**: spesso si mescolano le due strategie.

Architettura client-server Tipica dei sistemi distribuiti, composta da:

- Server che offrono servizi generici.
- Client che utilizzano questi servizi.
- Una rete di comunicazione per le richieste e risposte.

Pro e contro del client-server

- Pro: facilita la distribuzione dei dati e delle responsabilità, scalabilità.

- Contro: richiede risorse e può creare ridondanza, dipendenza dai server (necessità di un servizio di naming).

Architettura multi-tier La maggior parte dei sistemi scalabili implementa questo modello (Java Enterprise Edition, dotNet, ecc.).

Design del controllo Si tratta di definire "chi fa cosa" e "chi esegue i casi d'uso". Può essere centralizzato (sincrono) o decentralizzato (asincrono).

1. **Controllo centralizzato (sincrono)**:
    - Un unico sistema serve tutte le richieste (es. un server web).
    - Comunicazione sincrona (es. chiamate di funzione).
    - Pro e contro: punto di accesso e di fallimento unici.
2. **Controllo basato su eventi (asincrono)**:
    - Ogni modulo funziona indipendentemente, comunicazione asincrona.
    - Pro e contro: sistema distribuito complesso ma con interazione più robusta tra i moduli.

Esempi noti:

- Modelli di trasmissione in sistemi altamente paralleli.
- Modelli basati su interruzioni nei computer.
- Infrastrutture di micro-servizi moderne con broker di messaggi (MQTT, COAP).

Specifiche per il singolo modulo

Dopo aver definito il modello architetturale, è necessario redigere le specifiche per ciascun modulo. Ecco i principi fondamentali:

1. **Indipendenza e Bassa Dipendenza**:
    - Ogni modulo dovrebbe essere il più indipendente possibile dagli altri (basso accoppiamento).
    - Minimizzare la conoscenza tra sviluppatori riguardo agli altri moduli.
    - Servizi altamente dipendenti devono appartenere allo stesso modulo (alta coesione). Ad esempio, la funzionalità di "aggiornare età" e la memorizzazione dei dati utente.
2. **Definizione del Contratto del Modulo**:
    - Definire chiaramente l'interfaccia del modulo, indicando:
        - Le funzionalità esposte (es. "Aggiorna età", "Elimina utente").
        - Modalità di esposizione (funzioni da invocare, servizi da chiamare).
        - Parametri di input-output (numero e tipo di parametri).
3. **Utilizzo delle Notazioni UML**:
    - UML viene utilizzato sia per la fase di analisi sia per quella di progettazione e implementazione, mantenendo coerenza nei vari livelli di astrazione.

Modello MVC (Model-View-Control)

Strategia di partizionamento per componenti/moduli software:

- **Model**:
    - Rappresenta lo stato dell'applicazione, la sua rappresentazione, memorizzazione e comunicazione (Data Transfer Objects).
- **View**:
    - Visualizza il Model, essenzialmente rappresenta l'interfaccia utente.

- **Control**:
    - Contiene la logica applicativa e modifica il Model, direttamente ereditato dai diagrammi comportamentali.

Regole Generali per MVC

- Model, View e Control devono essere separati almeno in file diversi, spesso in pacchetti/componenti/librerie differenti.

Esempio di MVC in Java EE

- **Model**:
    - Stato del modello memorizzato in componenti implementati come JavaBeans.
    - JavaBeans devono:
        - Implementare `java.io.Serializable`.
        - Avere un costruttore pubblico senza argomenti.
        - Avere proprietà/campi privati con metodi getter e setter pubblici.
- **Integrazione delle Parti di MVC**:
    - La View accede al Model tramite getters.
    - Il Control modifica il Model tramite setters e accede ai suoi dati tramite getters.
    - View e Control sono disaccoppiati; il Control agisce come codice di supporto per una View, iniettando dati elaborati e attivando modifiche al Model in risposta all'interazione utente.

Vantaggi e Svantaggi di MVC

- **Pro**:
    - L'isolamento tra componenti migliora la modularità e la riusabilità.
    - Permette di passare facilmente da una vista web (es. JSP) a un'app mobile scritta con un'altra tecnologia.
- **Contro**:
    - L'architettura è più complessa, con più file e componenti, ma questo non rappresenta un problema significativo.

cosa è il MVVM

MVVM, che sta per **Model-View-ViewModel**, è un pattern architetturale utilizzato nello sviluppo software, particolarmente nelle applicazioni con interfacce utente ricche. È una variante del pattern Model-View-Controller (MVC) e si usa soprattutto in applicazioni di tipo desktop e mobile, nonché nello sviluppo di app web con framework come Angular o React.

Componenti del MVVM

1. **Model**:
    - Rappresenta i dati dell'applicazione e la logica di business. Include anche le regole di validazione e i servizi di accesso ai dati. È indipendente dalla UI e non ha alcuna consapevolezza della parte visuale dell'applicazione.
2. **View**:
    - È la parte dell'interfaccia utente dell'applicazione. Include tutto ciò che riguarda la presentazione delle informazioni all'utente e la gestione dell'input dell'utente. La View osserva il ViewModel per i cambiamenti dei dati e reagisce di conseguenza.
3. **ViewModel**:

- Funziona come intermediario tra la View e il Model. Contiene la logica di presentazione e si occupa della trasformazione dei dati del Model in un formato che la View possa facilmente presentare. Inoltre, il ViewModel implementa il binding bidirezionale, permettendo l'aggiornamento automatico dei dati tra la View e il Model.

Vantaggi del MVVM

- **Separazione delle Preoccupazioni**: Come in MVC, MVVM separa chiaramente le diverse responsabilità, facilitando la manutenzione e la scalabilità del codice.
- **Binding Dati Bidirezionale**: Permette la sincronizzazione automatica dei dati tra la View e il Model tramite il ViewModel, riducendo il codice boilerplate.
- **Testabilità**: Poiché la logica di presentazione è separata dalla View, il ViewModel può essere testato in isolamento, facilitando i test unitari.

Utilizzo Comune

MVVM è comunemente utilizzato nello sviluppo di applicazioni con Windows Presentation Foundation (WPF), Silverlight, Xamarin e framework JavaScript come Angular e Vue.js. Questi framework e librerie spesso offrono un supporto nativo per il data binding, che è un elemento chiave del pattern MVVM.

UML for Code Design (6)

Fornire astrazioni con UML

UML (Unified Modeling Language) è utilizzato per progettare come dovrebbe apparire il codice. A questo livello, ciò che conta è principalmente la rappresentazione dei dati. In precedenza, l'attenzione era concentrata su entità come parti, unità o componenti del sistema, modellando il loro comportamento con diagrammi di sequenza in base a diversi casi d'uso. Ora, bisogna passare a un livello di astrazione inferiore, osservando le entità.

ER Diagrammi (Entity-Relationship):

- Utilizzati per modellare database, descrivendo i dati creati e richiesti dai processi aziendali e le relazioni tra i vari componenti del sistema.

Z Diagrammi:

- Linguaggio di specifica formale che identifica chiaramente cosa fa un sistema informatico e come lo fa, basato su concetti matematici. È formale ma complesso da usare.

Diagrammi di Classi:

- Descrivono chiaramente classi/interfacce e le loro relazioni tramite nodi e archi, modellando il comportamento statico del sistema (ossia, le classi) definibile in fase di compilazione.

MVVM – Model View ViewModel:

- Una strategia di partizionamento per componenti/moduli software:
  - **Model:** Rappresenta lo stato dell'applicazione.
  - **View:** Come viene mostrato il Model, essenzialmente le interfacce utente.

- **ViewModel:** Logiche applicative che modificano il Model, derivando dai diagrammi comportamentali.

Concetti Chiave in UML

1. **Astrazione e Dettaglio:** UML permette di rappresentare il sistema a diversi livelli di dettaglio, facilitando la comprensione e la progettazione.
2. **Associazioni:** Linee solide che collegano classi, indicano le relazioni (es. composizione o aggregazione).
3. **Generalizzazione/Specializzazione:** Modella le relazioni padre-figlio (ereditarietà) tra le classi.
4. **Ereditarietà e Metodi:** Le proprietà nelle super-classi sono ereditate dalle sotto-classi, con possibilità di sovrascrittura.
5. **Modelli di Dipendenza:** Relazioni dove un elemento necessita di un altro per funzionare, modellate con linee tratteggiate e stereotipi.

Esempi di Notazioni UML:

- **Enum:** Stereotipo <<enumeration>>
- **Classi attive:** Bordi verticali doppi
- **Classi astratte:** Notazioni varie, indicano classi parzialmente implementate con metodi astratti.

Diagrammi degli Oggetti in UML:

- **Definizione:** Gli oggetti in UML sono istanze di classi che, durante l'esecuzione (run-time), memorizzano lo stato delle entità atomiche del nostro modello o rappresentazione.
  - **Proprietà:** Non memorizzano proprietà statiche, ma i dati e i metodi per accedere a tali dati.
  - **Notazione:** Simile ai diagrammi delle classi, con possibili notazioni specifiche del linguaggio.
- **Relazioni tra Oggetti:** Le relazioni tra oggetti sono simili a quelle tra classi e possono includere associazioni, aggregazioni e composizioni.

Diagrammi dei Pacchetti:

- **Definizione di Pacchetto:**
  - **In Java:** I pacchetti raggruppano entità (classi) per strutturare il codice e seguire il principio di divide et impera.
  - **Scelte di Design:** Decidere come raggruppare le classi (per funzionalità o altri criteri) è una scelta di design. Java impone una corrispondenza tra pacchetti e cartelle, mentre C# offre maggiore libertà, ma con più responsabilità.
- **Dipendenze tra Pacchetti:**
  - **Usage:** Indica una relazione client-server tra pacchetti, con lo stereotipo <<uses>> (predefinito).
  - **Import:** Utilizza lo stereotipo <<imports>>, dove il namespace del pacchetto fornitore diventa parte del namespace del pacchetto client.
  - **Access:** Con lo stereotipo <<accesses>>, gli elementi del pacchetto client possono accedere agli elementi del pacchetto fornitore (es. friend classes).

Visibilità tra Pacchetti:

- **Visibilità:** Determinata dai seguenti indicatori:

- **- (Privato):** Solo all'interno del pacchetto.
- **+ (Pubblico):** Accessibile da qualsiasi parte.
- **~ (Pacchetto):** Visibile all'interno del pacchetto.
- **# (Protetto):** Visibile solo ai pacchetti figli.

In sintesi, i diagrammi degli oggetti mostrano le istanze delle classi e il loro stato dinamico, mentre i diagrammi dei pacchetti aiutano a strutturare e gestire le dipendenze tra le diverse parti del sistema.

Solid and Clean (7)

Programmazione SOLID: SOLID è un insieme di cinque principi fondamentali per la progettazione orientata agli oggetti (Object Oriented Design) che aiutano a trasformare un programmatore in un architetto software.

1. **Principio di Responsabilità Singola (Single Responsibility Principle):**
   - **Definizione:** Ogni entità software dovrebbe avere una sola ragione per cambiare. Ogni classe dovrebbe avere una sola responsabilità o scopo.
   - **Vantaggi:** Facilita la comprensione e la manutenzione del codice, rendendo più chiaro dove e come apportare modifiche.
   - **Svantaggi:** Aumenta il numero di classi e il lavoro necessario per gestirle.
2. **Principio di Apertura/Chiusura (Open/Closed Principle):**
   - **Definizione:** Le entità dovrebbero essere aperte per estensioni ma chiuse per modifiche. Dovresti aggiungere nuove funzionalità tramite il polimorfismo piuttosto che modificare il codice esistente.
   - **Vantaggi:** Riduce il numero di bug e le problematiche nel codice.
   - **Svantaggi:** N/A
3. **Principio di Sostituzione di Liskov (Liskov Substitution Principle):**
   - **Definizione:** Dovresti poter sostituire qualsiasi classe base con una delle sue classi derivate senza modificare il comportamento del programma.
   - **Vantaggi:** Il codice è scalabile e riduce la necessità di modifiche quando vengono apportati cambiamenti.
   - **Svantaggi:** Richiede una riflessione accurata prima di implementare l'ereditarietà.
4. **Principio di Segregazione delle Interfacce (Interface Segregation Principle):**
   - **Definizione:** È meglio avere molte interfacce specifiche per i client piuttosto che una grande interfaccia generica. Le interfacce dovrebbero essere il set minimo di comportamenti necessari.
   - **Vantaggi:** Riduce le dipendenze nel codice e semplifica la manutenzione.
   - **Svantaggi:** Nessuno significativo.
5. **Principio di Inversione delle Dipendenze (Dependency Inversion Principle):**
   - **Definizione:** Il tuo progetto non dovrebbe dipendere da dettagli concreti, ma da astrazioni. Dovresti creare wrapper intorno alle tue dipendenze.
   - **Vantaggi:** Isola i componenti del codice e riflette meglio l'analisi/modello del business.
   - **Svantaggi:** Richiede uno sforzo di programmazione aggiuntivo.

In sintesi:

- **Principio di Responsabilità Singola:** Ogni classe ha un solo compito.
- **Principio di Apertura/Chiusura:** Aggiungi funzionalità senza modificare il codice esistente.
- **Principio di Sostituzione di Liskov:** Le classi derivate devono essere intercambiabili con le classi base.
- **Principio di Segregazione delle Interfacce:** Usa interfacce specifiche e minimali.

- **Principio di Inversione delle Dipendenze:** Dipendenza da astrazioni, non da implementazioni concrete.

Questi principi aiutano a creare codice più modulare, manutenibile e scalabile.

Design Patterns(8)

Cosa offrono i design pattern:

- **Vocabolario comune:** Forniscono un linguaggio standardizzato per discutere e risolvere problemi di design.
- **Soluzioni a problemi complessi:** Aiutano a risolvere problemi noti prima che si presentino.
- **Fondamenti solidi per le scelte di design:** Offrono basi per motivare e giustificare le decisioni di progettazione.

Cosa non offrono i design pattern:

- **Soluzioni esatte:** Ogni progetto è unico e richiede soluzioni personalizzate.
- **Soluzioni complete per tutti i problemi di design:** Non risolvono ogni problema di programmazione, ma aiutano a gestire problemi comuni.

Cosa obbligano a fare i design pattern:

- **Trovare oggetti appropriati per modellare il dominio:** Aiutano nella decomposizione del problema.
- **Determinare la granularità degli oggetti:** Per esempio, pattern di creazione come il Factory.
- **Definire chiaramente interfacce e classi:** Implementare e definire le relazioni tra di esse.
- **Implementare codice riutilizzabile:** Decidere tra ereditarietà o composizione/aggregazione e delega.

Errori comuni nei design pattern:

- Dichiarare esplicitamente le classi degli oggetti.
- Chiamare direttamente i metodi per implementare operazioni di alto livello.
- Dipendere fortemente da piattaforme hardware o software specifiche.
- Dipendere dagli interni di un'altra classe.
- Dipendere da algoritmi implementati personalmente.
- Accoppiamento stretto tra componenti, entità, o classi.
- Usare sempre sottoclassi per estendere funzionalità o specializzare comportamenti.

Principio di Inversione delle Dipendenze:

- **Definizione:** Il progetto non dovrebbe dipendere da dettagli concreti, ma da astrazioni.
- **Utilizzo:** Creare wrapper intorno alle dipendenze per mantenere l'isolamento e riflettere il modello di business.

Viaggio del software fino ad ora:

- **Applicazione:** Artefatti software autonomi con poche dipendenze. Tipici per progetti piccoli.
- **Toolkit:** Applicazioni che utilizzano runtime e librerie per funzionalità di base. Adatte per progetti più grandi.
- **Framework:** Strutture di classi che costituiscono l'architettura di un'applicazione. Rilevante per applicazioni complesse e specifiche di dominio.

Differenze tra Application, Toolkit e Framework:

- **Application:** Software autonomo, dipendente solo da piattaforme hardware/software.
- **Toolkit:** Software che utilizza librerie e runtime, adatto per progetti più grandi e complessi.
- **Framework:** Offre un'architettura predefinita e richiede di seguire le sue convenzioni.

Tipi di design pattern:

1. **Creational Patterns:**
    - **Factory:** Definisce un'interfaccia per la creazione di oggetti, lasciando alle sottoclassi la scelta della classe da istanziare.
    - **Singleton:** Garantisce che ci sia solo un'istanza di una classe nel sistema.
    - **Builder, Prototype:** Altri pattern di creazione per specifiche necessità.
2. **Structural Patterns:**
    - **Adapter, Bridge, Composite, Façade, Proxy, Decorator, FlyWeight:** Modelli per gestire le relazioni tra classi e oggetti.
3. **Behavioral Patterns:**
    - **Chain of Responsibility, Command, Iterator, Interpreter, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor:** Gestiscono algoritmi e responsabilità tra oggetti.

Struttura tipica di un design pattern:

1. **Nome e scopo:** Descrive il nome e lo scopo del pattern.
2. **Motivazione:** Perché utilizzare il pattern.
3. **Applicabilità:** Dove applicare e dove non applicare il pattern.
4. **Esempi e frammenti di codice:** Implementazione del pattern con esempi pratici.
5. **Effetti collaterali:** Impatti e potenziali problemi derivanti dall'uso del pattern.

Esempio pratico: Testing con Factory Pattern:

- **Problema:** Necessità di testare un'applicazione con un database senza eseguire un database reale.
- **Soluzione:** Usare un pattern di creazione come Factory per gestire l'istanza del database e facilitare i test unitari e di integrazione.

Factory Pattern e Abstract Factory

Abstract Factory Pattern:

- **Scopo:** Definire un'interfaccia per la creazione di oggetti correlati, senza specificare le classi concrete.
- **Motivazione:** Le classi che offrono varianti sono spesso correlate tra loro.
- **Applicabilità:** Quando il sistema utilizza più "famiglie" di oggetti e deve rimanere indipendente dall'implementazione concreta dei servizi.
- **Conseguenze:** Permette di cambiare rapidamente la "famiglia" di servizi utilizzati, ma aggiungere nuove classi richiede modifiche all'interfaccia della fabbrica.
- **Note:** Le fabbriche sono spesso Singleton. Questo pattern non è limitato alla OOP e può essere applicato anche a librerie runtime.

Adapter Pattern:

- **Scopo:** Convertire l'interfaccia di una classe in un'altra interfaccia richiesta dal client.
- **Motivazione:** Permette di utilizzare un'interfaccia che altrimenti non sarebbe compatibile con l'applicazione client.
- **Applicabilità:** Quando ci sono problemi di compatibilità tra due oggetti, perché il client utilizza un'interfaccia non dichiarata dall'oggetto sorgente.
- **Esempio:** Un controller di motore che usa un protocollo PWM e un sistema di guida che usa un protocollo diverso.
- **Conseguenze:** Il lavoro di un adapter dipende dalla differenza tra le interfacce da adattare. È possibile raggruppare adattatori in famiglie e applicare il principio di Responsabilità Unica anche alle interfacce.

Programmazione Embedded:

- **Caratteristiche Speciali:** Circuiti e sistemi specifici per scopi particolari, vincoli di tempo reale, gestione esplicita della memoria, e limitate risorse computazionali.
- **Codifica:** Tipicamente in C, C++ o linguaggi più ridotti, evitando OOP per motivi di performance e utilizzando strutture e funzioni.
- **Abstraction Layer e Adapter:** Utilizzati per astrarre e adattare interfacce hardware specifiche.

Hardware Proxy e Hardware Adapter:

- **Hardware Proxy Pattern:**
  - **Scopo:** Rappresentare un dispositivo hardware con una struttura e primitive specifiche che forniscono accesso a esso.
  - **Motivazione:** Evitare che le modifiche hardware influenzino il codice direttamente.
  - **Applicabilità:** Quando l'hardware non ha una rappresentazione standard.
- **Hardware Adapter Pattern:**
  - **Scopo:** Adattare l'interfaccia hardware specifica al formato richiesto dall'applicazione.
  - **Motivazione:** Differenze nei formati di dati tra interfacce hardware.
  - **Applicabilità:** Quando è necessario adattare strutture di dati dell'applicazione all'hardware.
  - **Conseguenze:** Richiede gestione della concorrenza e comunicazione basata su interruzioni. La conversione di formato può introdurre ritardi.

Code Smells e Anti-patterns:

- **Code Smells:** Indicatori che qualcosa potrebbe non funzionare correttamente nel codice. Spesso derivano da scelte di design errate o dalla violazione dei principi SOLID.
- **Anti-pattern:** Processi, strutture o modelli che, sebbene inizialmente sembrino appropriati, hanno conseguenze negative più gravi. Una soluzione documentata e ripetibile esiste per affrontare il problema.
- **Esempi di Smell Comuni:**
  - **Bloaters:** Codice, metodi e classi che sono diventati eccessivamente grandi e difficili da gestire.
  - **OO Abuse/Misuse:** Uso errato dei principi OOP.
  - **Changes Preventers:** Modifiche che richiedono modifiche in molteplici luoghi.
  - **Dispensables:** Codice non necessario come codice morto o duplicato.
  - **Couplers:** Dipendenze eccessive tra classi.

Conclusione:

- **La buona pratica:** Utilizzare design patterns per evitare problemi comuni e mantenere il codice manutenibile e scalabile. Implementare le migliori pratiche di codifica e progettazione per evitare e risolvere i code smells e gli anti-patterns.

Programming DotNET (9)

Cosa è .NET:

.NET è un framework che offre un ambiente integrato per lo sviluppo di applicazioni per internet, desktop e mobile, ed è orientato agli oggetti. Si distingue per essere indipendente dalla piattaforma e insensibile al linguaggio di programmazione, supportando quindi diversi linguaggi di coding. È progettato per essere portabile, con DotNet Core che gira su GNU/Linux e offre strumenti da shell. Inoltre, .NET è un framework gestito, il che significa che si occupa automaticamente della gestione della memoria e di altre risorse.

Architettura di .NET:

L'architettura di .NET è multi-linguaggio e cross-platform, basata su un ambiente simile alla JVM chiamato Common Language Runtime (CLR). Questo ambiente gestisce l'esecuzione del codice e si occupa della compilazione Just-in-Time (JIT), che trasforma il codice intermedio (CIL) in codice nativo per l'architettura specifica al momento dell'esecuzione. I componenti .NET sono organizzati in assembly, che sono pacchetti di codice e risorse.

Ecosistema .NET:

Nel contesto dell'ecosistema .NET, l'interoperabilità tra linguaggi è facilitata dalla Common Language Specification (CLS) e dal CLR. Esistono compilatori specifici per vari linguaggi come C#, VB e Managed C++. La piattaforma offre una vasta gamma di framework e librerie, tra cui WebAPI, WCF, ASP.NET, Windows Forms, MAUI e Framework Class Libraries (FCL). Per garantire la scalabilità, .NET utilizza pattern di design e richiede l'adozione dei principi SOLID.

Common Language Runtime (CLR):

Il CLR è un runtime comune per tutti i linguaggi .NET e fornisce un sistema di tipi comune, metadata condivisi e un processo di compilazione altamente ottimizzato. Gestisce la memoria tramite allocazione e raccolta dei rifiuti, facilitando anche la comunicazione tra codice gestito e non gestito.

Problemi con la Conformità C++:

C++ è considerato un linguaggio non gestito e non supporta direttamente le funzionalità offerte da .NET. È tradizionalmente basato su OS e librerie runtime, ma Microsoft ha introdotto Managed C++ per integrare C++ nel framework .NET, modificando la sintassi del linguaggio con nuove parole chiave.

Compilazione JIT e CLR:

La compilazione Just-in-Time (JIT) consente ai linguaggi .NET di essere compilati in Common Intermediate Language (CIL) e poi convertiti in codice macchina nativo per specifiche architetture. Questo processo è altamente ottimizzato e offre numerosi vantaggi, tra cui il supporto per servizi per

sviluppatori, interoperabilità tra codice gestito e non gestito, gestione della memoria, e accesso ai metadati.

Base Class Library (BCL):

La Base Class Library (BCL) di .NET è simile al namespace System di Java e rappresenta il sottogruppo "core" della Framework Class Library (FCL) utilizzato da tutte le applicazioni .NET. Fornisce funzionalità essenziali come input/output, threading, database, gestione del testo, interfaccia grafica, console, socket e web.

Confronto tra Java e .NET:

A differenza di Java, che è più aperto e ha una forte presenza nel settore open-source, .NET è principalmente proprietario, sebbene alcune componenti siano open-source. Mentre Java ha avuto difficoltà a stabilirsi nel mercato desktop, .NET è diventato una scelta predominante per le applicazioni Windows grazie alla sua integrazione con Visual Studio e il fatto che Windows include .NET ma non Java JRE.

Applicazioni Server:

.NET è altamente integrato con l'infrastruttura di Windows Server, offrendo vantaggi come la sicurezza integrata, debugging e distribuzione. Nel confronto tra Java EE e ASP.NET, entrambi sono utilizzati nei siti web di alto profilo, ma .NET è la scelta preferita per le applicazioni su Windows.

Funzionalità Avanzate di .NET:

.NET è fortemente tipizzato e offre funzionalità avanzate come la gestione delle proprietà, threading, e programmazione asincrona. La programmazione orientata agli oggetti in .NET si avvale di attributi e funzionalità di programmazione funzionale come le chiusure e i delegati. Il supporto per LINQ (Language-Integrated Query) e l'uso di interfacce fluente migliorano la leggibilità del codice, anche se possono complicare il debug e la modifica del codice.

In sintesi, .NET è un framework robusto e versatile, adatto a diverse piattaforme e linguaggi di programmazione, con un forte supporto per la gestione della memoria e la scalabilità del codice.

----- DOMANDE ---

SOLID e **Clean Architecture** non sono la stessa cosa, ma sono concetti complementari nel mondo della progettazione del software. Vediamo le differenze principali:

1. SOLID: Principi di progettazione orientata agli oggetti

SOLID è un insieme di cinque principi di progettazione che si applicano principalmente al design di classi e oggetti all'interno di un sistema software. Questi principi aiutano a creare codice che è più facile da mantenere, estendere e testare. Ecco i principi SOLID:

- **S**: **Single Responsibility Principle (SRP)** — Ogni classe o modulo dovrebbe avere una sola responsabilità o ragione per cambiare.

- **O**: **Open/Closed Principle (OCP)** — Le entità software (classi, moduli, funzioni) dovrebbero essere aperte all'estensione ma chiuse alla modifica.
- **L**: **Liskov Substitution Principle (LSP)** — Gli oggetti di una sottoclasse dovrebbero poter sostituire quelli della loro classe base senza alterare la correttezza del programma.
- **I**: **Interface Segregation Principle (ISP)** — I client non dovrebbero essere costretti a dipendere da interfacce che non utilizzano. Le interfacce dovrebbero essere specifiche per il cliente.
- **D**: **Dependency Inversion Principle (DIP)** — I moduli di alto livello non dovrebbero dipendere da quelli di basso livello. Entrambi dovrebbero dipendere da astrazioni.

I principi SOLID sono focalizzati a livello di **design del codice** e migliorano la qualità e la flessibilità di un singolo modulo o classe.

2. Clean Architecture: Struttura di alto livello di un sistema

L'**Architettura Clean** è una **struttura architetturale** che riguarda l'intera applicazione o sistema. Si concentra sulla separazione dei livelli logici e delle dipendenze tra le componenti, con l'obiettivo di mantenere la logica di business indipendente da dettagli implementativi come database, framework, interfaccia utente e così via.

L'architettura clean si applica alla struttura globale dell'applicazione e descrive **come separare i diversi strati di un sistema** (logica di business, casi d'uso, interfaccia utente, database, ecc.). In pratica, definisce come le diverse componenti devono interagire tra loro e in che modo le dipendenze dovrebbero essere gestite (in particolare, puntando sempre verso l'interno, come spiegato prima).

Confronto e Complementarietà

- **SOLID**: Principi che si applicano al **design a livello di classe** e alle interfacce, migliorando la qualità del singolo modulo o della singola classe.
- **Clean Architecture**: Principi che riguardano l'**organizzazione e la struttura** dell'intera applicazione a un livello più alto, organizzando i moduli e le dipendenze tra essi.

Come si completano:

Puoi utilizzare i principi **SOLID** per scrivere codice migliore a livello di singola classe o modulo all'interno di un'applicazione che segue l'**Architettura Clean**. In altre parole:

- **SOLID** migliora la qualità del codice interno di ciascun livello dell'architettura.
- **Clean Architecture** definisce come i livelli devono interagire tra loro e come organizzare l'intera applicazione.

Ad esempio, puoi avere un'architettura clean dove ogni singolo componente o strato segue i principi SOLID per garantire una buona modularità e manutenibilità.

In sintesi, **SOLID** e **Clean Architecture** non sono la stessa cosa, ma lavorano insieme per creare sistemi ben progettati e facili da mantenere.

L'Architettura Clean, introdotta da Robert C. Martin (spesso chiamato "Uncle Bob") nel suo libro *"Clean Architecture: A Craftsman's Guide to Software Structure and Design"*, è un approccio alla progettazione del software che si concentra sulla separazione delle responsabilità e sull'indipendenza delle componenti. Questo tipo di architettura mira a rendere il codice più flessibile, manutenibile e testabile, riducendo al minimo le dipendenze tra i vari livelli di un sistema.

Ecco i concetti chiave dell'Architettura Clean:

1. Separazione delle responsabilità

L'idea principale è di separare il codice in livelli distinti, dove ogni livello ha compiti specifici e ben definiti. Questi livelli non devono interferire tra loro e ciascuno può evolvere indipendentemente.

2. Indipendenza dalle tecnologie

I dettagli tecnologici, come framework, database o interfacce utente, devono essere separati dalla logica di business. Questo permette di cambiare facilmente tecnologie senza impattare la parte logica principale del sistema.

3. Struttura a strati (o livelli)

L'architettura viene spesso rappresentata come una serie di cerchi concentrici. Ogni cerchio interno rappresenta un livello più fondamentale del sistema e quelli esterni rappresentano dettagli tecnologici o implementazioni specifiche. Più nello specifico:

- Entità: rappresentano la logica di business fondamentale e le regole del dominio.
- Casi d'uso: contengono le regole che gestiscono specifici flussi di lavoro del sistema.
- Adattatori: collegano l'interno (logica) del sistema con l'esterno (come l'interfaccia utente o il database).
- Framework e driver esterni: sono componenti specifici della tecnologia, come database, interfacce web, ecc.

4. Dipendenze verso l'interno

Una regola chiave dell'Architettura Clean è che le dipendenze devono puntare verso l'interno, ovvero i livelli esterni (più specifici e tecnologici) dipendono da quelli interni (più astratti e logici). Questo protegge la logica di business da cambiamenti nelle tecnologie.

5. Testabilità

Poiché la logica di business è separata dai dettagli tecnologici, diventa molto più semplice testare il sistema, poiché i test possono concentrarsi sui vari casi d'uso senza dover coinvolgere le componenti esterne come il database o il front-end.

6. Principio dell'inversione delle dipendenze

Questo principio, derivato dalla Programmazione Orientata agli Oggetti, suggerisce che i moduli di alto livello (che definiscono la logica) non dovrebbero dipendere da quelli di basso livello (che implementano dettagli tecnici), ma entrambi dovrebbero dipendere da astrazioni. Le interfacce vengono spesso utilizzate per isolare le dipendenze.

Vantaggi dell'Architettura Clean:

- Facilità di manutenzione: grazie alla chiara separazione delle responsabilità, il codice è più facile da aggiornare e modificare.
- Scalabilità: consente di estendere o modificare facilmente funzionalità senza impattare l'intero sistema.
- Testabilità: semplifica i test unitari e di integrazione.
- Indipendenza tecnologica: il sistema può evolvere o migrare a nuove tecnologie senza riscrivere la logica di base.

Svantaggi:

- Curva di apprendimento: richiede una certa comprensione per essere implementata correttamente, soprattutto per progetti piccoli o con team inesperti.
- Overhead iniziale: nei piccoli progetti, potrebbe essere considerata eccessivamente complessa.

L'architettura clean è pensata per durare nel tempo, permettendo al sistema di adattarsi ai cambiamenti tecnologici e di business senza dover essere completamente riscritto.