Usiamo L’IDE per la ridondanza dell’errore. Quando si programma, gli errori che vengono creati non sono gravi, sono ISOLATI. Dal momento in cui usciamo dall’IDE ed entriamo in un contesto di programmazione reale, da li in poi quello che facciamo ha ridondanza, cioè ciò che fai riverbera nel computer, perché la macchina ha bisogno di elementi che sono sempre uguali, quindi COERENTI. Primo motivo per il quale utilizziamo è perché centralizza gli strumenti di programmazione, il secondo è perché da coerenza funzionale e isola l’errore, quindi evitare degli errori all’interno della macchina difficilmente gestibili. Diminuire incoerenza e chiudere l’errore su alcune pareti solidi (gli errori possibili sono x,y,z e devo lavorare su questi), è uno dei motivi principali per il quale usiamo l’IDE. Il terzo punto quindi è per la coerenza funzionale, ed è la capacità del codice del funzionare nella maniera identica indipendentemente dalla macchina che utilizziamo. Noi programmatori diamo coerenza funzionale. Java deve funzionare sempre alla stessa maniera, ad alto livello, quindi coerenza funzionale. Il contrario è proprio l’incoerenza funzionale.

Tramite Java io sviluppo una volta, e lo porto ovunque (portabilità).

Barra software in alto orizzontale e barra tool verticale (visual studio).

Quindi, un IDE, o ambiente di sviluppo integrato, è un software progettato per la realizzazione di applicazioni che aggrega strumenti di sviluppo comuni in un'unica interfaccia utente grafica.

**Definizione di Coerenza Funzionale e Incoerenza Funzionale**

**Coerenza funzionale** significa che un programma o un sistema si comporta sempre nello stesso modo, in modo prevedibile e stabile, indipendentemente dalla macchina, dall'ambiente o dal contesto in cui viene eseguito, a parità di librerie. Questo è particolarmente importante nello sviluppo software, perché garantisce che il codice funzioni allo stesso modo su computer diversi, con configurazioni diverse, o dopo essere stato modificato.

Un **IDE** (Integrated Development Environment) aiuta a ottenere coerenza funzionale perché:

1. **Centralizza gli strumenti di sviluppo**, come compilatori, debugger e controlli di sintassi.
2. **Isola gli errori**, rendendoli più facili da individuare e correggere prima che si propagano nel sistema reale.
3. **Riduce l'incoerenza**, fornendo un ambiente controllato dove è più difficile che il codice si comporti in modo imprevisto.

L'opposto è l'**incoerenza funzionale**, che si verifica quando il programma si comporta in modo diverso su macchine diverse o in condizioni differenti. Questo rende difficile individuare e correggere gli errori, perché non si ha la certezza che il software funzioni sempre nello stesso modo. Non è un problema avercela, anzi, lo sfruttiamo facendo in modo che gli elementi siano semplicemente controllabili

Introduciamo al concetto basilare e lineare di OOP

E’ un paradigma che deriva dall’inglese Object-oriented programming. Il fulcro di OOP sono le classi. Le classi esistono grazie all’utilizzo contemporaneo delle tre regole: ereditarietà, polimorfismo ed incapsulamento. Sono le tre regole fondamentali da dover sapere benissimo. C’è ne anche una quarta, cioè l’astrazione, MA DANDO una contro-opinione, dicendo che è la regola BASE, la regola madre di tutta l’OOP. quindi non è effettivamente una quarta regola. Queste quattro regole definiscono in maniera lineare che cos’è e come si usa l’oop. Si devono rispettare sempre queste 4 caratteristiche. Quindi è un paradigma di programmazione che permette di definire oggetti software che possono scambiarsi informazioni in maniera paritaria o subparitaria (stesso livello quindi sw con sw, oppure uno è il capo dell’altro). Il sw comunica subparitariamente verso il singolo oggetto. Tutti gli oggetti dello stesso insieme comunicano paritariamente. Ricordarsi che comunque abbiamo l’astrazione nella pratica applicata, quindi non solo come aspetto generico.

OOP, Particolarmente adatta nei contesti in cui si possono definire delle relazioni di interdipendenza tra i concetti da modellare (contenimento, uso, specializzazione), un ambito che più di altri riesce a sfruttare i vantaggi della programmazione ad oggetti è quello delle interfacce grafiche.

Le regole di modellazione (contenimento, uso, specializzazione), è legato sia alle tre regole fondamenti,quindi di base anche solo sfruttando queste regole le logiche dell’uml, posso riprodurre tutti i pezzi grafici necessari tramite la coerenza funzionale.

Quindi, alcuni dei vantaggi della programmazione OOP:

* essa fornisce un supporto naturale alla modellazione software degli  
  oggetti del mondo reale o del modello astratto da riprodurre;
* permette una più facile gestione e manutenzione di progetti di  
  grandi dimensioni;
* l'organizzazione del codice sotto forma di classi favorisce la  
  modularità e il riuso di codice.

La classe è un modello teorico che definisce attributi(caratteristiche funzionali) e comportamenti (metodi) di un oggetto.

Posso avere un nome per una classe con infiniti oggetti.

Se creiamo la classe matita, possiamo creare oggetti proprio basati sulla matita (matita rossa etc.).

Le classi, quindi, definiscono dei tipi di dato e permettono la creazione degli oggetti secondo le caratteristiche definite nella classe stessa. Grazie alle relazioni di ereditarietà, è possibile creare nuove classi a partire da quelle esistenti, estendendole con caratteristiche aggiuntive attributi, ossia variabili e/o costanti che definiscono le caratteristiche o proprietà degli oggetti instanziabili invocando la classe; i valori inizializzati degli attributi sono ottenuti attraverso il cosiddetto costruttore; metodi, ossia procedure che operano sugli attributi.

UML: intro to UML

Ci interessa la logica, la modalità che occupa per integrare poi al contesto di esecuzione, del nostro pensiero. UML (unified modeling language), un linguaggio di modellazione unificato o standard. C’è un paradigma all’interno, ed è utilissimo per la programmazione. Senza uml non si programma niente come software definito. E’ uno standard generale che si ritrova all’interno del contesto di esecuzione. E’ all’interno del sistema diretto che abbiamo, un fatto di adozione comunitaria e non di imposizione. È possibile specificare, visualizzare, progettare, documentare i sistemi, soprattutto in ambito sw. Possiamo fare due cose:

* + rappresentare una struttura statica, come diagrammi di classi, che mostrano relazioni tra componenti del sistema
  + diagrammi dinamici, che illustrano il flusso di interazioni e processi nel sistema.

Da un lato abbiamo le strutture statiche, che descrivono la collaborazione e le relazioni tra i componenti del sistema (come frecce, classi, condizioni); dall'altro lato ci sono gli elementi dinamici, che rappresentano i comportamenti e le sequenze che il sistema deve rispettare in fase di esecuzione, con possibili output e percorsi differenti. Un esempio è un configuratore di sistema, lui ha delle classi preimpostate (statiche), ma il modo in cui lavora dipende dal contesto (quindi in base a quanti virus ci sono, lavorerà x, se no y). Non si contraddicono a vicenda e non sono esclusivi.

UML è un modello di standardizzazione della modellazione dei sw sia a livello statico, sia a livello dinamico. Anche all’interno di un unico UML possiamo trovare caratteristiche del lato opposto. Si preoccupa a dare uno standard univoco per una rappresentazione sw coerente.

L’evoluzione di uml non è rigida, ma il fatto che sia versatile ed efficace serve al fatto che tutta una commissione che è incaricata di vigilare dei comportamenti del sistema OOP si occupa di verificare lo standard, e si occupa di comunicare cambiamenti o aggiornamenti. Ci sono stati diversi cambiamenti di uml ma nessuno ha stravolto la logica funzionale, perché ha bisogno di essere un minimo aggiornato per gli elementi necessari al funzionamento del sistema, ma non ha bisogno di capacità, quindi nel momemnto in cui lavoriamo nel contesto del funzionamento del sistema uml, ci dobbiamo ricordare che non esiste la commissione di uml, o qualcuno che lo aggiorna, ma esistono diversi contesti e persone che si occupano di andare a lavora nei vari sistemi.

Parte tecnica

Immagine che contiene testo, schermata, Carattere, diagramma

Il contenuto generato dall'IA potrebbe non essere corretto.

Ci sono due parti tecniche e tre concettuali: la standardizzazione e la versatilità sono elementi tecnici che non si possono scindere da uml. Il fatto che è versatilè è una tecnica, per fare in modo che sia replicabile, cioè quello che creiamo, strutturiamo, è replicabile. Se tu trovi uml online di un social network, tu puoi creare a tua volta un social network. Saper leggere il codice è importante non si parla solo di codice ma anche di ciò che è intorno (come funzionano oggetti, come collaborano, quali caratteristiche hanno in comune e quali esclusive). Racchiude quindi sempre linee guida chiamate legenda, utili alla descrizione funzionale del progetto.

Elementi fondamentali

UML fornisce regole e logiche per riprodurre software. Possiamo creare anche nuove regole ma abbastanza inutile per quanto riguarda le basi (per il resto va bene).

Le relazioni tra uml e database è soprattutto che in pratica all’interno dell’uml, noi abbiamo bisogno della legenda, logica definita ma soprattutto da noi, rispetto a quello che c’è nel database.

Le classi sono fatte da rettangoli. Ogni colore significa qualcosa di diverso. + public, - privato. Ha senso invertire la logica? Si (nuovi stereotipi), ma non conviene mai per gli elementi fondamentali, perché dobbiamo garantire coerenza.

Struttura con regole condivise che permette di creare sia nuove regole, che utilizzare regole basilari, per andare a lavorare all’interno del contesto, in maniera come sempre COERENTE (coerenza funzionale, logiche ed esecuzioni che siano definite, e che siano ripetibili).

QUINDI:

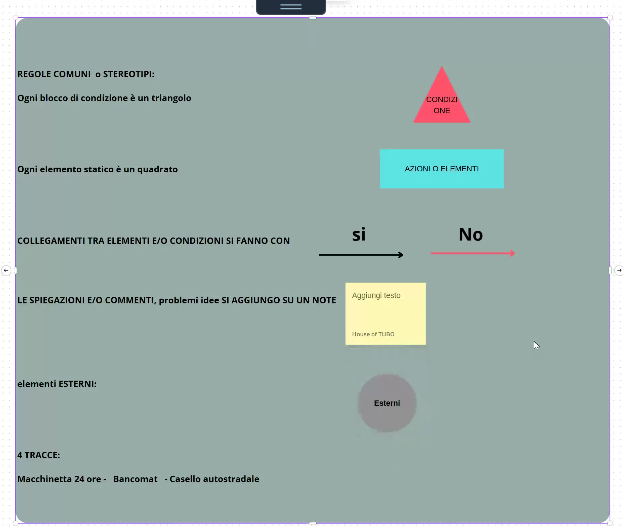
* + classi e oggetti: noi non faremo oggetti, faremo classi, ma dentro uml puoi rappresentare diagrammi con le classi che diagrammi con gli oggetti.
  + Attributi e operazioni: la parte inferiore dei riquadri sono azioni, quelle sopra sono attributi. Dentro uml (sia statici che dinamici), possiamo rappresentare entrambi i contesti.
  + Relazioni: Freccia aperta è ereditarietà. La freccia normale invece sono collegamenti. Includono associazioni, ereditarietà, aggregazioni e composizioni per definire le connessioni tra elementi
  + Interfacce: contratti tra i componenti che fanno capire come hanno una relazione questi elementi. Clausole che legano gli elementi insieme. Un’altra clausola è la condizione, con un triangolo, per dire “si” o “no” e vedere cosa accade.
  + Notazioni grafiche e standardizzate: simbolo della freccia che cambia, se c’è più o meno, il fatto di mettere i due punti per distinguere nome e tipo, etc.

Alla foto precedente, per essere un vero UML manca la legenda. Dobbiamo per forza di cose avercela. Durante gli esercizi useremo una legenda.

Come si collegano gli elementi fra loro? Vediamo un esempio:

Immagine che contiene testo, diagramma, schermata, linea

Il contenuto generato dall'IA potrebbe non essere corretto.



Qui abbiamo tre regole rigide. I triangoli sono condizioni, quadrati azioni da fare, frecce sono l’esecuzione o meno (per errori). Ogni condizione deve avere almeno un si e un no. Gli elementi esterni e le note non le vediamo.