- ***Lezione 25***

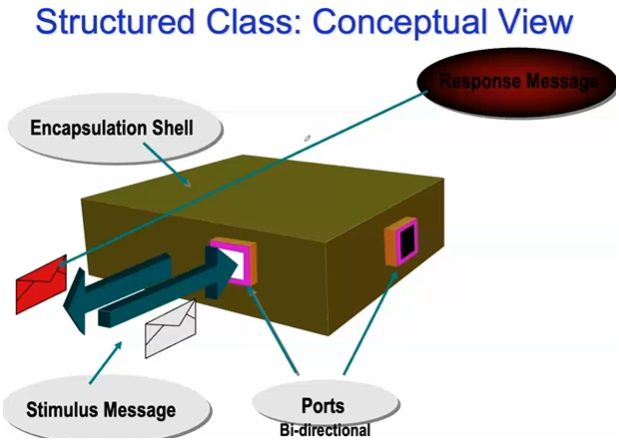
La Legge di Demeter (anche nota come “don’t talk to strangers” in quanto si basa sull’idea di non “comunicare” con oggetti non noti) è una legge che aiuta a mantenere basso l’accoppiamento inte-classi migliorando la manutenibilità del codice.
Essa afferma che un metodo può inviare messaggi (cioè invocare metodi) solo ai seguenti oggetti:

1. L’oggetto del metodo stesso (un metodo deve poter invocare i metodi su se stesso, es. usando this in Java e C++)
2. Oggetti passati come argomenti nel metodo (perché essendo l’oggetto formalmente presente nei parametri dell’operazione e quindi necessario per la sua esecuzione è un oggetto noto, si conosce la classe dalla quale questo oggetto è creato)
3. Oggetti appartenenti a uno degli attributi dell’oggetto corrente (ossia se tra gli attributi dell’oggetto che ha invocato me metodo vi sono altri oggetti io metodo posso invocarne i metodi, tuttavia in questo caso vale la strong law, ossia non posso invocare i metodi degli oggetti che sono a loro volta loro attributi (annidati))
4. Un oggetto creato dal metodo
5. Un oggetto che fa riferimento a una variabile globale

Come detto la Legge di Demeter favorisce quindi manutenibilità ma anche comprensione del codice, infatti per ad esempio ridurre il numero di righe si potrebbe pensare di fare invocazioni a partire dal mio metodo ad un oggetto noto a per poi invocare altri metodi non noti a.b().c().d(), tuttavia ciò non favorisce la comprensibilità e quindi la manutenibilità del codice in quanto un programmatore diverso potrebbe non capire il senso di tale scelta.
Ora quello che ci resta per completare questa parte di progettazione OO è capire come utilizzare UML per produrre i diagrammi di implementazione, che ancora non abbiamo considerato. Essi permettono di definire in modo preciso come gli oggetti che progettiamo vengono tradotti in componenti eseguibili.

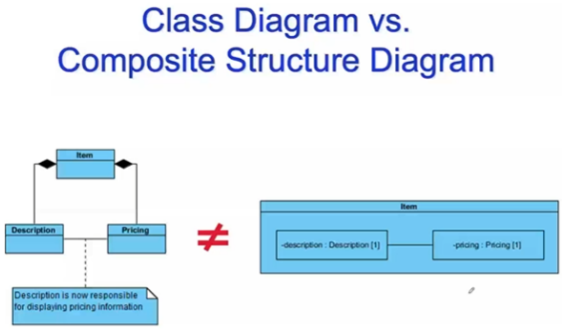
Come abbiamo già accennato una delle novità più interessanti di UML 2 è l’introduzione delle structured classes. Mentre in UML 1 la classe è intesa come semplice aggregato di dati e operazioni, in UML 2 si mantiene lo stesso simbolo (forma rettangolare) ma diventa “classe strutturata” in quanto contiene elementi detti roles o parti che formano la sua struttura e ne descrivono il comportamento.
Questo meccanismo è gerarchico in quanto un ruolo/parte di una classe strutturata può essere a sua volta classe strutturata, ciò permette di gestire la complessità in modo stratificato lavorando a diversi livelli di astrazione (un po’ come avevamo fatto con i DFD data flow diagram che permettevano di rappresentare il comportamento del nostro software a diversi livelli di astrazione).
Ogni ruolo rappresenta un elemento partecipante nella realizzazione della struttura interna della classe, e questi ruoli sono interconnessi attraverso il concetto di connettore (come il ruolo anche il connettore rappresenta quindi una primitiva in UML 2, che permette appunto di descrivere come queste parti interagiscono tra loro)
Esiste una differenza tra ruoli e parti anche se molto sottile: i ruoli seguono una semantica “per riferimento” mentre le parti “per valore” (quindi i ruoli rappresentano degli attributi

presenti nella classe per riferiemento mentre le parti sono proprio contenute nella classe strutturata) (simile al ragionamento fatto introducendo i costrutti per aggregation e composition: la prima semantica per riferimento perché l'oggetto contenuto non è fisicamente contenuto ma utilizzato attraverso riferimento esterno, mentre nella composition l'oggetto è proprio fisicamente contenuto -> existence dependency etc…)
Le structured classes sono molto utili in fase di progettazione. Mentre in fase di analisi dei requisiti ci focalizziamo sul COSA identificando le classi che definiscono gli elementi fondamentali e dichiarandone le operazioni solo con la loro firma e le relative associazioni, in fase di progettazione dobbiamo definire la struttura interna della classe, vedendo ad es. ciascuna operazione COME viene realizzata.
Con UML 1 dovremmo limitarci a progettare ogni operazione dal punto di vista algoritmico, mentre con UML 2 possiamo delegare a dei ruoli interni la realizzazione di queste operazioni.
In questo senso si utilizza la classe strutturata per definire i building blocks di un'applicazione, nascondendo i dettagli implementativi.
In particolare vi è un incapsulamento molto stretto del comportamento, per cui ogni interazione tra elementi di classi diverse deve avvenire attraverso un meccanismo basato su messaggi (qui si torna al ragionamento fatto sul Coupling: il livello migliore è Data/Stamp dove l'interazione tra moduli avviene tramite scambio di messaggi)
Si vede quindi concettualmente la classe strutturata come una scatola nera che mette a disposizione servizi (operazioni elencate nell'interfaccia pubblica della classe) nascondendo l'implementazione delle operazioni agli utilizzatori della classe.
Per utilizzare i metodi messi a disposizione dalla classe si utilizzano delle "porte", attraverso le quali inviare messaggi e riceverne eventualmente indietro.
Inoltre all'interno della classe strutturata possono esservi altre classi strutturate, altre classi semplici etc…



Internamente poi deve essere definito come viene realizzata la classe strutturata sia in termini strutturali che comportamentali. Per la parte comportamentale si utilizzano spesso macchine a stati finiti (come la classe evolve durante il suo funzionamento transitando da uno stato iniziale a uno finale).
In questo senso quindi ogni classe può essere vista come un elemento di progettazione autonomo, molto utile in fase di progettazione perché quindi posso suddividere il team per lavorare su classi diverse partendo dall'interfaccia definita in fase di analisi dei requisiti, utile anche a livello di testing (si testa a livello di unità Unit-Testing fino alla verifica del funzionamento dell'applicazione quando queste unità sono integrate, integration testing).

Ma che differenza c'è nell'utilizzo dell'approccio structured classes (in termini di ruoli e parti) rispetto ad un approccio che faccia uso dei costrutti di aggregation e composition? Vediamolo con un esempio pratico.

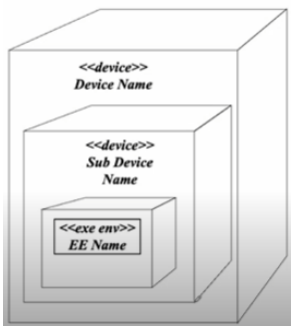


Si vuole rappresentare il legame tra la classe item e le classi descrizione e prezzo.
Con il class diagram e composition: si rappresenta come item sia costituito da due oggetti interni, uno che descrive l'elemento e l'altro che fornisce informazioni sul prezzo → se cancello l'oggetto item cancello anche gli oggetti description e pricing in esso contenuti.
Riguardo la classe strutturata invece metto nella classe item due parti: una che è istanza di description e una che è istanza di pricing.
Perché sono diversi? Il significato in sé è lo stesso, però la classe strutturata mi fornisce un'informazione ancora più precisa in particolare legata all'associazione tra Description e Pricing: con la classe strutturata sto dicendo che anche l'associazione fa parte della classe strutturata (linea tra description e pricing), mentre con il class diagram non è chiaro come viene gestita (se cancello l'oggetto item l'associazione resta o no?)
Quindi la classe strutturata rappresenta un meccanismo migliore per rappresentare informazioni tipiche di sistemi software di grandi dimensioni, dove gli elementi vengono rappresentati in modo gerarchico. Si garantisce una precisione e quindi pulizia di progettazione maggiore rispetto all'uso di UML 1.
Il passaggio che ci manca e che non era possibile con UML 1 è: una volta progettata in dettaglio la mia applicazione via classi strutturate, communication diagrams etc… come si traducono le classi dal punto di vista di componenti fisiche che poi saranno effettivamente mandate in esecuzione su uno dei nodi di esecuzione?
Come passare dall'insieme di classi strutturate all'eseguibile che contiene l'applicazione che funziona grazie a quelle classi?
In UML 1 il concetto di "componente" descrive un componente fisico, e mancava totalmente il passaggio da classi a livello di progettazione a eseguibile.
Con UML 2 invece, dato che le classi sono descritte in dettaglio tramite classi strutturate e dato che il concetto di "componente" ha subito un cambiamento sostanziale (da descrivere un singolo componente fisico è passato a descrivere un elemento a livello di progettazione) è possibile descrivere il passaggio da progettazione a software effettivo da eseguire.
Questo passaggio viene fatto anzitutto descrivendo la piattaforma di esecuzione sottostante.
La configurazione di piattaforma definisce come le funzionalità del software possono essere distribuite sui vari nodi fisici dove viene eseguito il sistema software distribuito.
Questo viene ottenuto attraverso due passi:

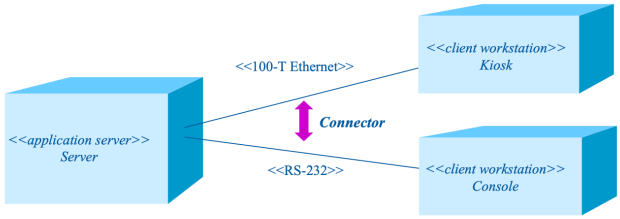
- si definisce la piattaforma hardware sottostante attraverso il Deployment Diagram in UML

- si descrive come sui nodi fisici di esecuzione vengono allocati i componenti software ricavati durante la fase di progettazione.

Questi componenti software non si chiamano più “components” come in UML 1, ma artefatti.
Per quel che riguarda il deployment diagram ogni nodo di esecuzione è rappresentato da un parallelogramma, esistono due tipi di nodi: e può rappresentare o una risorsa computazionale in grado di eseguire il software oppure dei “support device” che non hanno capacità di elaborazione (es. switch di rete).
Le connessioni sono rappresentate mediante archi che collegano i nodi, anche in questo caso ampia libertà sul significato: meccanismo di comunicazione, mezzo fisico o protocollo software.
Come detto quindi un nodo rappresenta una risorsa computazionale che ha tipicamente capacità di elaborazione e memoria:
Ne esistono due tipologie principali: Nodi Device se rappresentano proprio la risorsa fisica con capacità di processing o Execution environment se rappresenta particolari piattaforme di esecuzione allocate nel nodo.



Rigurado i Connector invece rappresenta una connessione tra due nodi delle tipologie citate sopra. Nell’esempio qui sotto a sinistra l’application server rappresenta un execution environment, collegato a una workstation (chiosco) via ethernet (mezzo fisico) mentre console (terminale) collegato direttamente al server attraverso connessione seriale di tipo RS-232.

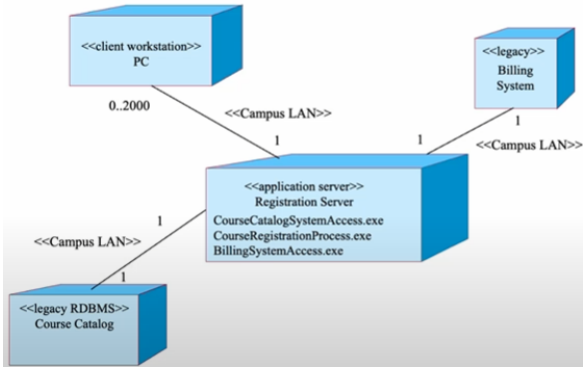


Essendovi nel deployment diagram grande libertà di rappresentazione, è fondamentale come si vede dall’immagine utilizzare gli stereotipi (es. <<100 T Ethernet>>) che permettono di annotare sia nodi che connettori descrivendo le informazioni necessarie a comprenderne il significato.

Quello che ci manca è capire come collegare il Registration Server a quanto ho progettato in fase di progettazione, ossia come inserire le classi strutturate e i vari meccanismi affinché l’applicazione possa funzionare? (questo mancava in UML 1)

Ciò fa riferimento al Process-to-Node Allocation, ossia come assegnare i vari processi ai dispositivi hardware in esecuzione. Per farlo si tiene conto di vari aspetti:

- Pattern di Distribuzione (il carico deve essere distribuito adeguatamente di modo da evitare colli di bottiglia)
- Si vuole trovare un'allocazione che minimizzi i tempi di risposta e aumenti throughput
- Minimizzazione del traffico attraverso la rete (si vorrebbe che processi comunicanti spesso tra loro siano sullo stesso nodo o su nodi vicini per minimizzare traffico)
- In base alla capacità dei nodi (CPU, RAM, spazio)
- In base alla larghezza di banda del mezzo di comunicazione
- In base all'availability dei nodi e connessioni (se connessioni instabili ne devo tener conto per l'allocazione dei processi)
- In base ai Rerouting Requirements (se un nodo fallisce o una connessione si interrompe, il sistema deve poter riallocare i processi)



Si inseriscono quindi i processi nei vari nodi di esecuzione (es. il processo che permette la registrazione e l'accesso al catalogo corsi e al sistema di billing garantendo interoperabilità)
Ma cosa metto nei vari processi rispetto a quanto realizzato in fase di progettazione?
Di questi aspetti se ne occupa l'attività di Deployment.
Rappresenta ciò che mi permette di assegnare/mappare gli artefatti software sui nodi fisici durante l'esecuzione.
L'artefatto software rappresenta quindi l'entità su cui può essere fatto il deployment verso il nodo fisico, e questi artefatti modellano le entità fisiche che in UML 1 erano rappresentate attraverso il costrutto componente (quindi componente rimpiazzata in UML 2 da artifact).
Tra gli artefatti si hanno quindi file, eseguibili, tabelle database, pagine web etc…

Ma quanto definito in fase di progettazione in quali artefatti finisce?
UML 2 permette di definire quest'informazione tramite il concetto di Manifestazione.
La Manifestazione è una relazione tra un elemento di un modello e il relativo artifact che implementa il modello.

Questi legami di deployment possono anche essere arricchiti di informazioni per specificarne meglio il significato attraverso il Deployment Specification.

Con ciò abbiamo concluso la parte legata alla fase di progettazione Object Oriented.