- ***Lezione 11 (parte due)***

- *Quinto Blocco*

La fase di OOA (Objet Oriented Analisis) definisce cosa un prodotto software deve fare, senza occuparsi del come. Successivamente [secondo modulo] vedremo la fase di OOD (Objet Oriented Desiner).

Queste due fasi devono fornire, ciascuno dal proprio punto di vista, una rappresentazione corretta, completa* e consistente:

- dal punto di vista dei dati → il modello dei dati si occupa degli aspetti statici e strutturali, deve descrivere i dati che dovranno essere gestiti dal software
- dal punto di vista comportamentale → il modello comportamentale si descrive sulle funzioni che dovranno elaborare e processare questi dati
- dal punto di vista dinamico → il modello dinamico si sofferma sugli aspetti di controllo e su come questi dati vengano modificati.

Completa → significa che oltre a considerare tutti i requisiti deve anche rappresentare e specificare il software da ogni punto di vista, quali il modello dei dati, il modello comportamentale e il modello dinamico.

Questi metodi di OOA, definiscono l'insieme di procedure, tecniche, linguaggi e strumenti per un approccio sistematico alla gestione e allo sviluppo della fase di queste attività.

Attività che riceve come input l'insieme dei requisiti utente (contenuti nel documento di analisi dei requisiti) e come output produce l'insieme dei modelli del sistema che definiscono la specifica del prodotto software (e che sono anch'essi contenuti nel documento di analisi dei requisiti)

Questi metodi sfruttano un approccio semi-formale e quindi fanno principalmente uso di notazioni visuali (diagrammi), ma possono essere affiancati da metodi tradizionali per la definizione di requisiti di sistema di tipo testuale (in linguaggio naturale strutturato)

Lo sviluppo dei modelli di OOA non è un processo sequenziale (prima modello dei dati, poi modello comportamentale, infine modello dinamico). La costruzione di questi avviene in parallelo (concorrente), e ciascun modello fornisce informazioni utili per gli altri modelli. I metodi di OOA fanno uso di un approccio iterativo, con aggiunta di dettagli per raffinamenti successivi (iterazioni)

Non esiste uno o tanti metodi, possiamo analizzare i seguenti:

- *Catalysis:* metodo di analisi e progettazione OOAD particolarmente indicato per lo sviluppo di sistemi software a componenti distribuiti.

- *Objectory:* metodo ideato da I. Jacobson, si basa sull'individuazione dei casi d'uso utente (*use case driven*).
- *Shlaer/Mellor:* metodo particolarmente indicato per lo sviluppo di sistemi software real-time.
- *OMT (Object Modeling Technique):* metodo sviluppato da J. Rumbaugh basato su tecniche di modellazione del software iterative. Pone in particolare risalto la fase di OOA.
- *Booch:* metodo basato su tecniche di modellazione del software iterative. Pone in particolare risalto la fase di OOD.
- *Fusion:* metodo sviluppato dalla HP a metà degli anni novanta. Rappresenta il primo tentativo di standardizzazione per lo sviluppo di software orientato agli oggetti. Si basa sulla fusione dei metodi OMT e Booch.

Ogni metodo aveva una sua notazione per la rappresentazione dei modelli del sistema., negli anni si è pensato di tentare di proporre un linguaggio standard. Questo linguaggio standard si chiama UML.

Questo è stato adottato nel 1997 come standard OMG (Object Management Group).

L'origine di UML è legata agli autori di tre metodi visti prima, ovvero:

- *Objectory*
- *OMT*
- *Booch*

Questi autori si sono trovati a lavorare all'interno di un'azienda, chiamata Rational Software. Oggi non esiste più, però è stata un'azienda molto importante. Questi tre autori si sono trovati insieme nei primi anni '90 e hanno combinato i relativi linguaggi di modellazione dei vari metodi e hanno sviluppato questo linguaggio (univoco) di modellazione.

Prima di definirlo come standard è stato necessario seguire una procedura molto complessa. Questo è un processo molto lungo, il tutto nasce con RFP (Request for Propusa) nel quale nasce l'esigenza di creare un linguaggio di modellazione unificato.

Si compone di nove formalismi di base (diagrammi con semantica e notazione data) e di un insieme di estensioni. All'interno di questo standard è stato introdotto un meccanismo standard di estensione, per dare la possibilità di estendere UML mantenendo la compatibilità verso la versione base del linguaggio.

[*N.B:* UML è un linguaggio di descrizione, non è un metodo né definisce un processo]

Negli anni si è pensato di fare un ulteriore tentativo, che prende il nome di *Unified Software Development Process* (in breve *Unified Process*), nel quale si è pensato di standardizzazione di processo di sviluppo di sistemi orientati agli oggetti basato

sull'uso di UML. A volte si fa riferimento a questo con la denominazione di RUP, dove *r* sta per *Rational* poiché è avvenuto anch'esso all'interno di questa azienda.

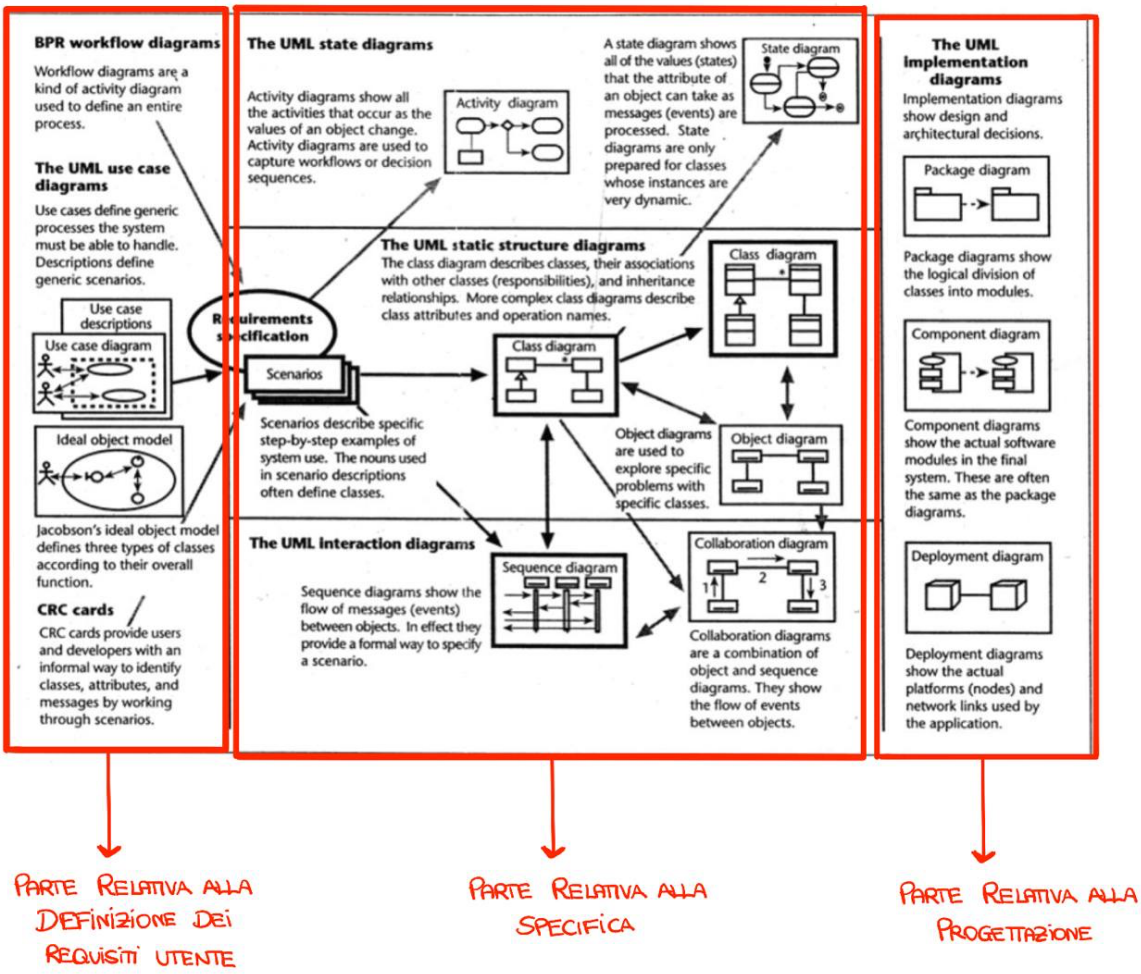


Diagramma UML

I nove formalismi dello UML sono:

1. *Use case diagram* → evidenziano la modalità (caso d'uso) con cui gli utenti (attori) utilizzano il sistema. Possono essere usati come supporto per la definizione dei requisiti utente.
2. *Class diagram* → consentono di rappresentare le classi con le relative proprietà (attributi, operazioni) e le associazioni che le legano.
3. *State diagram* → rappresentano il comportamento dinamico dei singoli oggetti di una classe in termini di stati possibili e transizioni di stato per effetto di eventi.
4. *Activity diagram* → sono particolari state diagram, in cui gli stati rappresentati rappresentano azioni in corso di esecuzione. Sono particolarmente indicati per la produzione di modelli di work-flow.
5. *Sequence diagram* → evidenziano le *interazioni* (messaggi) che oggetti di classi diverse si scambiano nell'ambito di un determinato caso d'uso, ordinate in sequenza

temporale. A differenza dei diagrammi di collaborazione, non evidenziano le relazioni tra oggetti.

6. *Collaboration diagram* → descrivono le *interazioni* (messaggi) tra oggetti diversi, evidenziando le relazioni esistenti tra le singole istanze.
7. *Object diagram* → permettono di rappresentare gli oggetti e le relazioni tra essi nell'ambito di un determinato caso d'uso.
8. *Component diagram* → evidenziano la strutturazione e le dipendenze esistenti tra componenti software.
9. *Deployment diagram* → evidenziano le configurazioni dei nodi elaborativi di un sistema real-time ed i componenti, processi ed oggetti assegnati a tali nodi.

- ***Lezione 12***

Esistono alcuni metodi che partono dal modello comportamentale e altri metodi che partono dal modello dei dati.

Noi ci focalizzeremo sul *Modello dei Dati.* Questo rappresenta da un punto di vista statico e strutturale l'organizzazione logica dei dati da elaborare. Le strutture dati sono definite mediante lo stato degli oggetti, che viene determinato dal valore assegnato ad attributi e associazioni. Il modello dei dati viene specificato mediante il formalismo dei *class diagram* che permette di definire:

- classi
- attributi di ciascuna classe
- operazioni di ciascuna classe
- associazioni tra classi

Il modello dei dati è di fondamentale importanza, visto che, secondo l'approccio ad oggetti, un sistema software è costituito da un insieme di oggetti (classificati) che collaborano

Questo modello di dati è un modello iterativo ed incrementale. Si tratta di un processo creativo, in cui giocano un ruolo importante sia l'esperienza dell'analista che la comprensione del dominio applicativo

Nella fase iniziale di costruzioni del modello dei dati occorre concentrarsi sulle cosiddette *entity classes,* ovvero quelle classi che definiscono il dominio applicativo e che sono rilevanti per lo sviluppo del software. Quello che dovremmo fare per costruire il modello di dati è costruire un *class diagram*, facendo uso della notazione *UML*, nel quale andiamo ad identificare le *classi entity*, specificandole in termini di attributi e associazioni e tralasciando l'identificazione delle operazioni in un secondo momento.

Le *control classes* (che gestiscono la "logica" del sistema) e *boundary classes* (che rappresentano l'interfaccia utente) vengono introdotte successivamente, usando le informazioni del *modello comportamentale*

Le operazioni di ciascuna classe vengono identificate a partire dal *modello comportamentale*, per cui vengono inizialmente trascurate

In particolare vediamo un insieme di tecniche o approcci che ci aiutano in termine di identificazioni delle classi. Questi approcci sono 4, l'ultimo è una combinazione dei precedenti:

- Noun phrase → questo è l'approccio delle frasi nominali (noun phrase). Dove per frase nominale intendiamo una frase in cui il sostantivo prevale sulla parte verbale (sono frasi di tipo assertivo). I sostantivi delle frasi nominali usate per la stesura dei requisiti utente sono considerati classi candidate (candidate classes). La lista delle classi candidate viene suddivisa in tre gruppi:

- *Irrilevante (irrelevant)* → quindi non appartengono al dominio applicativo e quindi possono essere scartate e non diventare una entity classes
- *Rilevante (Relevant)* → quindi una classe che posso identificare come entity classes
- *Fuzzy* → sono quelle classi delle quali non abbiamo sufficienti informazioni per classificarle come rilevanti o irrilevanti, vanno analizzate successivamente.

N.B: Si assume che l'insieme dei requisiti utente sia completo e corretto.

- *Common class patterns* → questo approccio è basato sulla teoria della classificazione, la quale studia il partizionamento in gruppi significativi detti *pattern.* Si hanno un certo insieme di classi predefinite, possiamo vedere un esempio di software per la prenotazione di biglietti aerei:

  - concetti (es. Reservation)
  - eventi (es. Arrival)
  - oerganizzazioni (es. AirCompany)
  - persone (es. Passenger)
  - posti (es. TravelOffice)

  Non è un approccio sistematico, ma può essere utile come guida. A differenza dell'approccio noun phrase, non si concentra sul documento dei requisiti utente. Può causare problemi di interpretazione dei nomi delle classi.

- *Use case driven* → è un approccio derivato da Jacobs. Nel quale i requisiti utente sono definiti con un *use case diagram* (diagramma dei casi d'uso). Per ogni use case sia fornita una descrizione testuale dello scenario di funzionamento, simile all'approccio noun phrase (si considera l'insieme degli use case come insieme dei requisiti utente). Si assume che l'insieme degli use case sia completo e corretto. Approccio function – driven (o problem - driven secondo la terminologia object oriented)

- *CRC* → L'approccio *CRC* (*Class - Responsibility – Collaborators*) è basato su riunioni in cui si fa uso di apposite card. Sono riunioni condotte dal team di analisti software, dove questi usano queste card. Ogni card rappresenta una classe, e contiene tre compartimenti, che identificano:

  - *Il* nome della classe
  - Le responsabilità assegnate alla classe
  - Il nome di altre classi che collaborano con la classe

  Le classi vengono identificate analizzando come gli oggetti collaborano per svolgere le funzioni di sistema. Questo approccio è utile per verifica di classi identificate con altri metodi e per l'identificazione di attributi e operazioni di ciascuna classe.

- Mixed → è una combinazione degli approcci precedenti. Un possibile scenario potrebbe essere il seguente:

1. L'insieme iniziale delle classi viene identificato in base all'esperienza dell'analista, facendosi eventualmente guidare dall'approccio *common class patterns*
2. Altre classi possono essere aggiunte usando sia l'approccio *noun phrase* che l'approccio *use case driven* (se gli *use case diagram* sono disponibili)
3. Infine l'approccio *CRC* può essere usato per verificare l'insieme delle classi identificate

Le seguenti linee guida ci servono per capire quando una classe può essere definita entity

1. Ogni classe deve avere un ben preciso statement of purpose
2. Ogni classe deve prevedere un insieme di istanze (oggetti), le cosiddette singleton classes (per la quali si prevede una singola istanza) non sono di norma classificabili come entity classes
3. Ogni classe deve prevedere un insieme di attributi (non un singolo attributo)
4. Distinguere tra elementi che possono essere modellati come classi o come attributi
5. Ogni classe deve prevedere un insieme di operazioni (anche se inizialmente le operazioni vengono trascurate, i servizi che la classe mette a disposizione sono implicitamente derivabili dallo statement of purpose)

- ***<u>Lezione 13</u>***

Ecco alcune Linee guida per la specifica delle classi, in particolare:

- Nomi di classe
    - Associare ad ogni classe un nome significativo nello specifico dominio applicativo, poiché dal nome devo capire cosa rappresenta.
    - Adottare una convenzione standard per assegnare nomi alle classi, ad esempio: nome singolare, parole multiple devono essere congiunte, con l'iniziale di ciascuna parola in carattere maiuscolo (`es: PostalAddress`)
    - Definire una lunghezza massima per i nomi delle classi (non più di 30 caratteri)

- Attributi e operazioni
    - Considerare inizialmente solo attributi che caratterizzano possibili stati di interesse per gli oggetti
    - Adottare una convenzione standard per assegnare nomi agli attributi, ad esempio: le parole devono essere scritte in carattere minuscolo, separate da un carattere di *underscore* (`es: street_name`)
    - Ritardare l'aggiunta di operazioni fino al momento in cui sia disponibile il modello comportamentale, da cui vanno derivate

- ***<u>Lezione 14</u>***

Come gestiamo le associazioni?

Possiamo ricordare alcune informazioni a carattere implementativo:

- Alcuni *attributi* identificati con le classi rappresentano associazioni (ogni attributo di tipo non primitivo dovrebbe essere modellato come un'associazione alla classe che rappresenta quel tipo di dato)
- Ogni *associazione ternaria* dovrebbe essere rimpiazzata con un *ciclo di associazioni binarie*, per evitare problemi di interpretazione
- Nei cicli di associazioni almeno un'associazione potrebbe essere eliminata e gestita come *associazione derivata*, anche se per problemi di efficienza spesso si introducono associazioni ridondanti

Quindi possiamo dire che un'associazione ha 3 proprietà:

- Per *assegnare nomi alle associazioni* adottare la stessa convenzione usata per gli attributi (le parole devono essere scritte in carattere minuscolo, separate da un carattere di *underscore*)
- Assegnare nomi di ruolo (*rolename*) alle estremità dell'associazione (i *rolename* diventano i nomi degli attributi nella classe all'estremità opposta dell'associazione)
- Determinare la *molteplicità* delle associazioni (ad entrambe le estremità)

UML fornisce tipi particolari di associazioni, come ad esempio le *aggregazioni*:

- Rappresenta una relazione di tipo " *whole – part* " (contenimento) tra una classe composta (*superset class*) e l'insieme di una o più classi componenti (*subset classes*). Questo tipo di relazione (ovvero l'aggregazione) può assumere quattro differenti significati:
    - *ExclusiveOwns*
        - <u>Transitività</u>
        - <u>Asimmetricità</u>
        - <u>Dipendenza di esistenza</u> → quando cancello l'oggetto contenitore cancello tutti gli oggetti contenuti
        - <u>Fixed Property</u> → quando un'istanza di una classe è in relazione ad un'istanza della classe contenitore non può essere messa in relazione con nessun'altra istanza contenitore

    ES: un libro ha dei capitoli, o un capitolo è parte di un libro

- *Owns*
    - Non abbiamo la Fixed Property

  ES: una macchina ha gli pneumatici

- *Has*
    - Non abbiamo dipendenza di esistenza
    - Non abbiamo la Fixed Property

  ES: una facoltà ha dei dipartimenti

- *Member* (la più debole)
    - Nessuna proprietà speciale eccetto l'appartenenza

  ES: La riunione ha un presidente

- ***<u>Lezione 15</u>***

UML ha soltanto due primitive in termini di relazioni di aggregazione (relzioni di contenimento)

- Aggregation

Si rappresenta con un rombo vuoto (◇). Ha una semantica per riferimento. Corrisponde alle aggregazioni di Has e Member

- Composition

Si rappresenta con un rombo pieno (◆). Ha una semantica per valore. Corrisponde alle aggregazioni ExclusiveOwns e Owns.

L'*Ereditarietà*, o *generalizzazione* (poiché in UML non esiste il concetto di ereditarietà, ma esiste una primitiva che si chiama generalizzation).
Usata per rappresentare la condivisione di attributi ed operazioni tra classi. Questa classe generica è detta superclasse, mentre le classi specializzate vengono dette sottoclassi o classi derivate.
Una sottoclasse *eredita* attributi ed operazioni della superclasse.
Abbiamo due caratteristiche per quanto riguarda l'ereditarietà:

- Sostituibilità → un oggetto della sottoclasse può essere utilizzato come valore da assegnare ad una variabile della superclasse.

  `Es: una variabile di tipo Frutta può avere un oggetto di tipo Mela come suo valore`

- Polimorfismo → la stessa operazione può avere differenti implementazioni nelle sottoclassi. Il sistema a Run - Time è in grado di disambiguare automaticamente il tipo di oggetto al quale sta chiedendo di fare qualcosa e ovviamente seguire il metodo corretto.

UML supporta queste associazioni di generalizzazione:

Dal punto di vista della sintassi visuale tutto questo si rappresenta con una freccia che va dalla sottoclasse alla superclasse.

Possiamo identificare relazione di tipo generalizzazione quando leggiamo:

- "può essere" (can – be)

  `Es: lo studente può essere un TeachingAssistant (Assistente didattico)`

- "è un tipo di" (is – a – kind – of)

Es: TeachingAssistant è un tipo di studente

UML supporta il concetto di ereditarietà multipla. Permette ad una classe di ereditare il comportamento di più superclassi.

Una rappresentazione di istanze di classi è il Diagramma degli Oggetti. Questo si chiama così proprio perché rappresenta gli oggetti. Questo è utile perché non può esistere un oggetto senza una classe corrispondente.

Si usa per.

- modellare relazioni complesse tra classi (a scopo esemplificativo)
- illustrare le modifiche ai singoli oggetti durante l'evoluzione del sistema
- illustrare la collaborazione tra oggetti durante l'evoluzione del sistema

Possiamo introdurre il Modello Comportamentale Questo modello si concentra sugli aspetti funzionali del sistema da un punto di vista operativo, evidenziando come gli oggetti collaborano ed interagiscono al fine di offrire i servizi che il sistema mette a disposizione.

Questo modello fa uso di vari (diagrammi) formalismi:

- Use case diagram (per descrivere scenari di funzionamento)
- Activity diagram (per descrivere il flusso di elaborazione)
- Sequence diagram (per descrivere l'interazione tra gli oggetti)
- Collaboration diagram (per descrivere l'interazione tra gli oggetti)

Anche questo modello viene costruito in modo iterativo ed incrementale, usando le informazioni del modello dei dati, che a sua volta fa uso del modello comportamentale per identificare operazioni e classi aggiuntive (control classes e boundary classes).

Analizziamo questi diagrammi:

- Diagramma dei casi d'uso (Use Case Diagram)

  Questo diagramma deve descrivere una funzionalità completa. Deve essere una funzionalità visibile dall'esterno. Deve avere un comportamento ortogonale (ogni use case viene eseguito in modo indipendente dagli altri). Non posso avere un caso d'uso se non ho un attore che lo attiva. Rappresenta una funzionalità che produce un risultato significativo per un attore.

Può essere sviluppato a differenti livelli di astrazione (sia in fase di OOA che OOD).

I casi d'uso li identifichiamo sia a partire dai requisiti utente, sia soffermandosi sugli attori e le loro necessità.

UML ci fornisce una sintassi per la specifica dei casi d'uso:

- Caso d'uso che si rappresenta con un'ellisse
- L'attore che si rappresenta con il simbolo di un omino

Questi elementi sono collegati attraverso relazioni, avremo due casi:

- Quando la freccia parte dall'attore e va al caso d'uso → stiamo indicando un attore responsabile dell'attivazione di quel caso d'uso
- Quando la freccia parte dal caso d'uso e va verso l'attore→ stiamo indicando se l'attore è coinvolto nell'esecuzione di quel caso d'uso (lui non lo attiva, è solo coinvolto)

Esistono relazioni tra due casi d'uso (escluso l'attore)

- Caso d'suo A collegato con una relazione di INCLUDE al caso d'uso B → sto indicando che "A include B", cioè quando un attore attiva il caso d'uso A per poter completare A deve necessariamente attivare e completare B
- Caso d'suo A collegato con una relazione di EXTEND al caso d'uso B → sto indicando che "A estende B", cioè quando un attore attiva il caso d'uso B potrei in alcuni casi (non necessariamente) attivare A, che a sua volta può essere attivato o da un attore differente o dallo stesso attore

Infine abbiamo una relazione tra coppie di attori chiamata Generalizzazione. Se io ho un attore A che viene specializzato da un attore B (freccia da B ad A), significa che B può attivare tutti i casi d'uso che attiva A e tutti i casi d'uso suoi specifici.

- ***Lezione 16***

Il diagramma delle attività, è fornito da UML ed ha un potere espressivo molto elevato, poiché permette di rappresentare il flusso di esecuzione in 2 modalità:

- sia sequenziale
- sia concorrente

È una variante degli state diagram, in cui gli stati (i nodi) rappresentano l'esecuzione di azioni e (gli archi) le transizioni sono attivate dal completamento di tali azioni.

Un vantaggio è che questo diagramma può essere anche in assenza del class diagram.

In presenza del class diagram, ogni attività può essere associata ad una o più operazioni appartenenti ad una o più classi

Usato principalmente in fase di OOD per rappresentare il flusso di esecuzione delle operazioni definite nel class diagram. In fase di OOA, viene usato per rappresentare il flusso delle attività nella esecuzione di un caso d'uso (un caso d'uso può essere associato ad uno o più activity diagram).

UML ci fornisce le seguenti primitive:

- si parte da un nodo iniziale che rappresenta l'evento di inizio, che corrisponde all'attivazione di un caso d'uso.
- Le attività da svolgere, che corrispondono ai nodi, ovvero gli action state si identificano pensando ad uno scenario di funzionamento di un caso d'uso
- questi action state sono collegati mediante transition lines (transizioni) che possono essere controllate da guard conditions (condizioni di guardia), il che significa che io una volta terminata un'attività posso andare su quella successiva percorrendo una transizione se e soltanto se la condizione annotata su quella transizione è vera.

Negli Activity Diagram posso rappresentare:

- Flussi concorrenti → un flusso sequenziale viene specificato in più flussi concorrenti. Questo viene realizzato utilizzando delle barre di sincronizzazione, chiamate barre fork - join (fork dove il flusso si splitta, join dove viene modificato)
- Flussi alternativi → vengono modellati con nodi decisionali di branch (che rappresenta flussi alternativi) e merge diamonds (questi flussi alternativi vengono ricombinati) → (rappresentati con dei rombi)
- Eventi esterni non sono generalmente modellati

I Diagrammi di interazione, vengono messi da UML in due modalità:

- Sequence diagram
    - Descrive lo scambio di messaggi tra oggetti in ordine temporale
    - Usato principalmente in fase di OOA
- Collaboration diagram
    - Descrive lo scambio di messaggi tra oggetti mediante relazioni tra gli oggetti stessi
    - Usato principalmente in fase di OOD

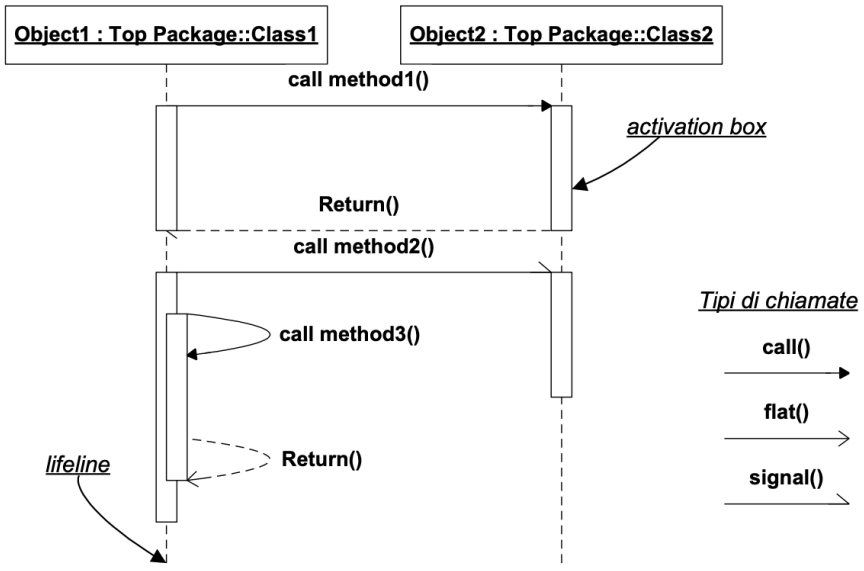
Questi permettono di identificare le operazioni delle classi nel class diagram. Sono rappresentazioni equivalenti e possono essere generati in modo automatico l'uno dall'altro.

Il diagramma di sequenza mostra in modo esplicito le iterazioni tra vari oggetti. Queste iterazioni avvengo attraverso uno scambio di messaggi. Le attività specificate nel diagramma delle attività possono essere mappate come messaggi in un sequence diagram.

Questi messaggi possono essere di due tipi:

- Asincrona → in UML si chiama Signal (segnale). È una richiesta che si basa sul paradigma "Send-no-Replay", l'oggetto mittente continua l'esecuzione dopo aver inviato il messaggio asincrono
- Sincrona → in UML si chiama Call. È una richiesta che si basa sul paradigma "Send-Replay", l'oggetto mittente blocca l'esecuzione dopo aver inviato il messaggio sincrono, in attesa della risposta da parte dell'oggetto destinatario.

Dal punto di vista visuale la notazione è la seguente:



I rettangoli rappresentano gli oggetti che interagiscono. Per ogni oggetto devo dire nome dell'oggetto e classe dal quale quell'oggetto è stato creato.

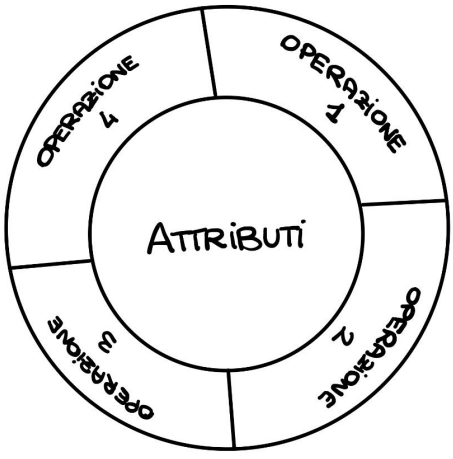
Per ogni oggetto si rappresenta una lifeline (linea tratteggiata) che rappresenta la vita dell'oggetto (scorre dall'alto verso il basso). Qui vengono definiti gli scambi tra messaggi inviati e ricevuti. Queste interazioni tra oggetti vengono rappresentate tramite archi orientati.

Quando l'oggetto durante la sua vita è attivo si usano dei rettangolini vuoti posizionati sulla lifeline, questi si chiamano activation box. [N.B: è una cosa al livello simbolico, infatti UML non ha il concetto di time]

Si parla di Interfaccia Pubblica di Classe. Esiste in Java una tecnica chiamata Information Hiding, la quale si realizza definendo un Interfaccia Pubblica. L'Information Hiding, ha l'idea di implementare le caratteristiche di riusabilità, leggibilità, manutenibilità del codice cercando di far vedere all'esterno soltanto quello che è necessario che gli altri oggetti conoscano. Nel caso del costrutto della classe, quello che ci dice il principio di OOP è che quando si va a definire una classe, gli attributi di questa classe andrebbe nascosta dietro un'interfaccia fornita dalle operazioni. Questi vengono definiti come accessor method che vengono chiamati come getter e setter.

Solitamente una classe viene rappresentata come visto in precedenza, attraverso un rettangolo con 3 "scompartimenti".

Inizialmente esisteva un'altra rappresentazione (che al giorno d'oggi non ne più usata, ma secondo il prof cattura meglio l'essenza dell'Information Hiding):



La "corona" (ovvero il cerchio di operazioni) definisce il concetto di Interfaccia Pubblica di Classe.

Durante la fase di OOA, si determina la signature dell'operazione, che consiste di:

- Nome dell'operazione

- Lista degli argomenti formali
- Tipo di ritorno

Durante la fase di OOD, si definisce l'algoritmo che implementa l'operazione. Una operazione può avere:

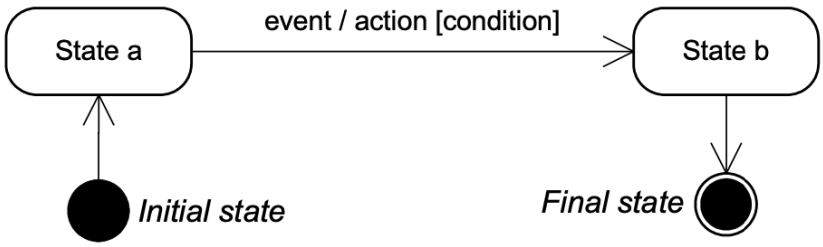
- Instance scope
- Class (static) scope
    - Rappresentata con un carattere $ che precede il nome dell'operazione.
    - Agisce su class object (classi con attributi static)

Per identificare queste operazioni la prima cosa che facciamo è utilizzare tutte le informazioni che ci vengono fornite dal sequence diagram. Si tratta di andare a recuperare per ogni messaggio inviato, il tipo di messaggio, la lista dei parametri, l'eventuale tipo di messaggio di ritorno e aggiungere l'operazione corrispondente alla classe.

Ad ogni classe vanno aggiunte le operazioni che fanno riferimento al criterio CRUD, secondo cui ogni oggetto deve supportare le seguenti operazioni primitive (CRUD operations):

- Create (una nuova istanza)
- Read (lo stato di un oggetto)
- Update (lo stato di un oggetto) • Delete (l'oggetto stesso)

Oltre al modello dei dati e al modello comportamentale serve costruire il modello dinamico. Questo rappresenta il comportamento dinamico degli oggetti di una singola classe, in termini di stati possibili ed eventi e condizioni che originano transizioni di stato (assieme alle eventuali azioni da svolgere a seguito dell'evento verificatosi). Si fa uso del formalismo State Diagrams event / action [condition]



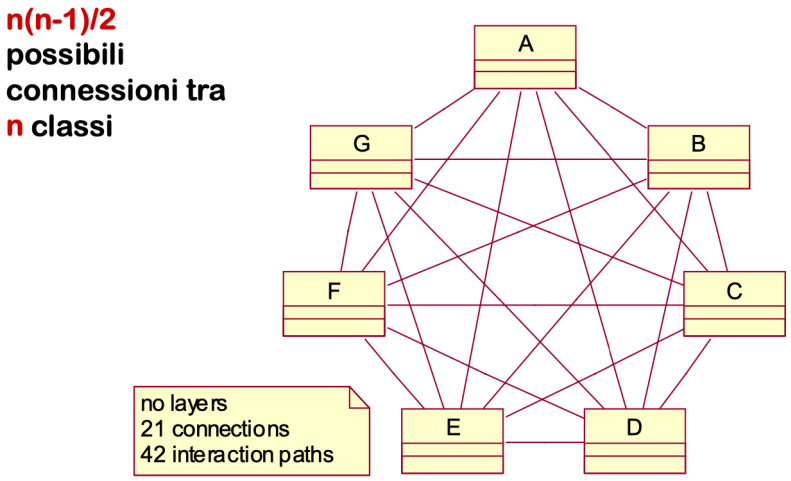
Ogni transizione può essere "etichettata" attraverso → l'evento, la condizione, l'azione

Viene costruito per ogni classe di controllo (per le quali è interessante descrivere il comportamento dinamico)

Usato principalmente per applicazioni scientifiche e real- time (meno frequentemente nello sviluppo di applicazioni gestionali)

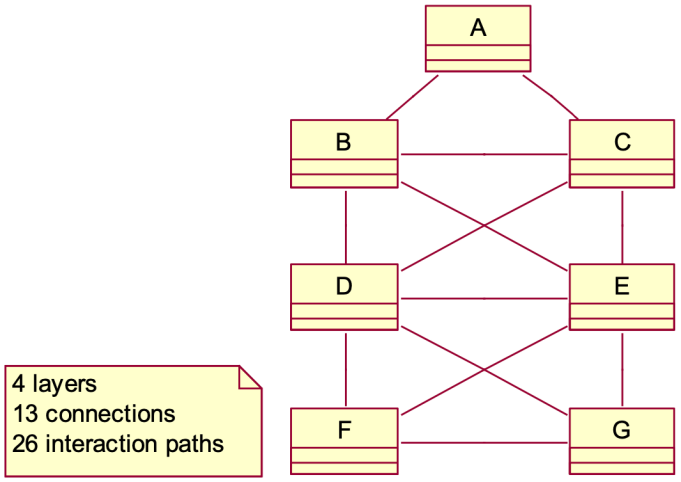
Gestione della complessità nei modelli di OOA. Nella fase di OOA per sistemi software di grandi dimensioni occorre gestire in modo opportuno l'intrinseca complessità dei modelli. Le associazioni tra classi nel modello dei dati formano complesse reti di interconnessione, in cui i cammini di comunicazione crescono in modo esponenziale con l'aggiunta di nuove classi. L'introduzione di gerarchie di classi permette di ridurre tale complessità da esponenziale a polinomiale, grazie all'introduzione di opportuni strati di classi che vincolano la comunicazione tra classi appartenenti allo stesso strato o a strati adiacenti.

- Class diagram non stratificato



Questo è un class diagram interamente connesso, dove ogni classe è associata alle altre classi. Nel nostro caso abbiamo 7 classi e $n\,(\,n - 1\,)\,/\,2$ possibili connessioni, cioè 21.

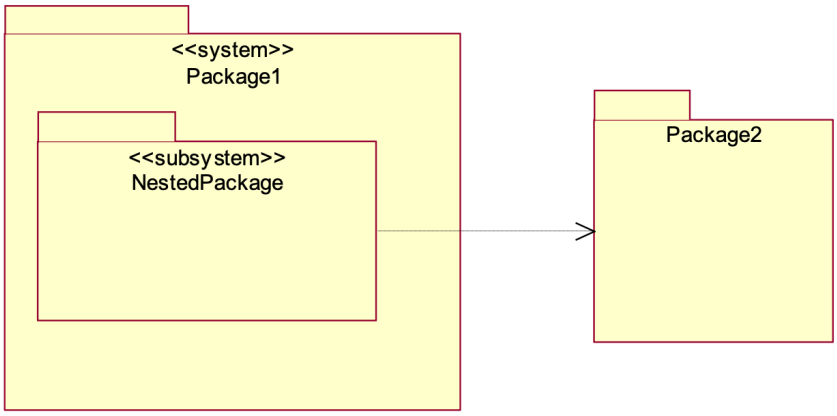
- Class diagram stratificato



Questo è un class diagram stratificato. Qui le classi possono comunicare solo con le classi degli strati adiacenti.

Questi strati vengono definiti attraverso UML Package. L'UML fornisce la nozione di package per rappresentare un gruppo di classi o di altri elementi (ad esempio casi d'uso). I package possono essere annidati (il package esterno ha accesso ad ogni classe contenuta all'interno dei package in esso annidati). Una classe può appartenere ad un solo package, ma può comunicare con classi appartenenti ad altri package. La comunicazione tra classi appartenenti a package differenti viene controllata mediante dichiarazione di visibilità (private, protected, o public) delle classi all'interno dei package.

Per quanto riguarda la rappresentazione grafica della dipendenza tra package avremo:



Si possono specificare due tipi di relazioni tra package:

- Generalization → implica anche dependency
- Dependency → dipendenza di uso, dipendenza di accesso, dipendenza di visibilità

In UML non esiste il concetto di package diagram. I package possono essere creati all'interno di:

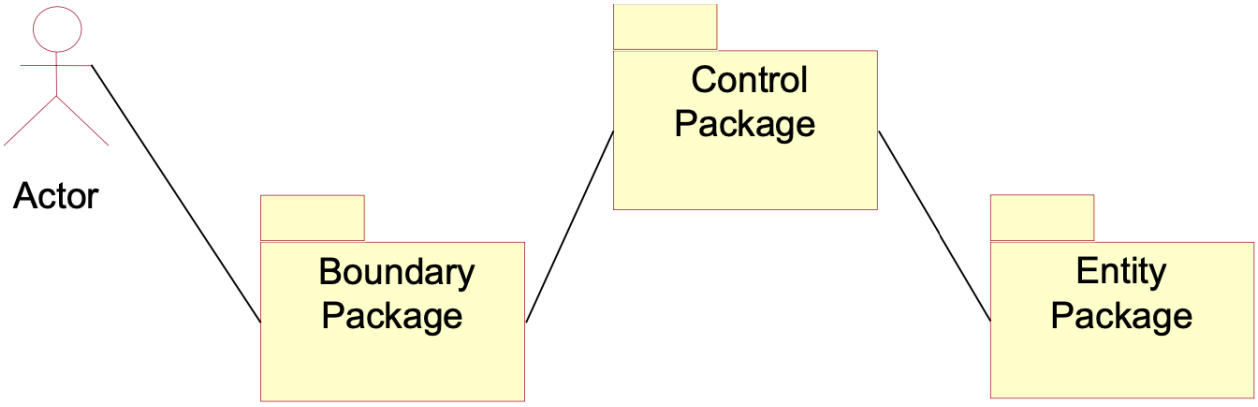
- class diagram
- use case diagram

Per definire quanti sono gli strati e come raggrupparli usiamo l'approccio BCE (Boundary Control Entity):

- Boundary package (BCE) → andremo a inserire le classi di oggetti che gestiscono l'interfaccia tra un attore ed il sistema. Le classi catturano una porzione dello stato del sistema e la presentano all'utente in forma visuale (interfaccia utente)
- Control package (BCE) → raggruppa tutte quelle classi alle quali viene inoltrato l'input dell'utente e devono saper rispondere a questo input. Le classi rappresentano azioni ed attività di uno use case
- Entity package (BCE) → descrive classi i cui oggetti gestiscono l'accesso alle entità fondamentali del sistema. Le classi corrispondono alle strutture dati gestite dal sistema

Questo approccio e simile al paradigma MVC (Model View Controller).

In figura vediamo una gerarchia di package nella quale le richieste degli utenti vanno sugli oggetti boundary, questi interagiscono con gli oggetti di controllo, i quali per poter rispondere possono accedere all'entity package.



N.B: Un oggetto boundary non può comunicare direttamente con un oggetto entity, e viceersa.