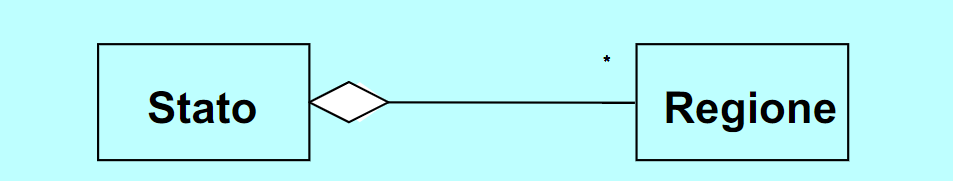
**LEZIONE 1 8/03**

**Definizione di Software** -> Programmi per computer con la relativa documentazione, quale ad esempio requisiti, modelli di progetto, e manuali utente. Non solo programmi, ma l’insieme degli ‘artefatti’ che lo compongono, prodotti durante il suo sviluppo. Questo è importante perché questi processi sono stati creati per essere utilizzati da tante persone, che poi vogliono capire come è strutturata l’ architettura.

**La definizione** **IEEE(Institute of Electrical and Electronic Engineers)** – insieme di programmi, procedure, regole, e ogni altra documentazione relativa al funzionamento di un sistema di elaborazione dati.

**Differenza tra software amatoriale e software professionale**

**Software amatoriale** si scrive per il proprio utilizzo, non sarà utilizzato da nessun altro, non ha bisogno di essere documentato, non ha bisogno di essere prodotto secondo un ciclo di vita X, lo fa in maniera rudimentale, non c’è nessuno obbiettivo di qualità specifico da raggiungere.

**Software professionale** , ci devono lavorare più persone in processo di sviluppo e quindi è un software che sarà utilizzato da queste persone e sarà modificato da altre persone ancora, quindi deve essere documentato. I software professionali devono quindi possedere delle caratteristiche di qualità. Dovrete fornire informazioni aggiuntive (sull’uso e sul progetto), oltre al codice.

**Software generici** sistemi autonomi, prodotti da una organizzazione, per svariati clienti Es. software per dispositivi mobili, per PC (database, office automation). Sistemi per gestire biblioteche, magazzini, contabilità...

**Software personalizzati** sistemi commissionati ad un produttore software da uno specifico cliente

Categorie di software

•**Commerciale**(es. Windows OS, Ms Office, ...)

•**Shareware**(ver. prova + registrazione ad 1 mese)

•**Freeware**(distribuito in modo gratuito, con o senza codice sorgente)

–Open Source (es. Linux)

–Non Open Source (Java)

**•Public Domain**(non ci sono più diritti)

Il Software può essere di diversi tipi:

–Software di sistema/ Applicativo/ di Produttività/ ApplicazioniWeb/ Gaming...

–Stand alone/ Embedded/ a supportodi processi(di produzioneindustrialeo aziendale)

–Generico/ Personalizzato...

Servono approcci diversificati per produrlo

Il software può essere ottenuto in vari modi:

–sviluppando ex-novo / configurando sistemi software generici(es. ERP) / riusando software pre-esistente/ model based engineering ...

I costi del Software spesso dominano i costi complessivi dei sistemi informatici. I costi del software per PC sono spesso maggiori dei costi dell’hardware stesso. Una parte rilevante dei costi del software sono spesi per la sua manutenzione/ evoluzione. È addirittura più costoso manutenere il software piuttosto che svilupparlo, soprattutto per sistemi “legacy”.

Un software di qualità dovrebbe fornire le **funzionalità** e le **prestazioni** richieste essendo contemporaneamente manutenibile, fidato, efficiente ed accettabile.

•**Manutenibilità** –Il Software deve essere in grado di evolvere per soddisfare le nuove richieste dei clienti;

•**Fidatezza** (Dependability) –La fidatezza include l’affidabilità, la safety (incolumità) e la sicurezza (protezione e riservatezza);

•**Efficienza** –Il software non dovrebbe sprecare le risorse del sistema (memoria, CPU);

•**Accettabilità** –Il Software deve essere accettato dai suoi utenti e pertanto deve essere comprensibile, usabile e compatibile con altri sistemi.

L’ ingegneria del software è una disciplina ingegneristica che si occupa di tutti gli aspetti relativi allo sviluppo del sofware.

•Gli ingegneri del software dovrebbero adottare:

–un approccio sistematico e organizzato per il loro lavoro (ossia un PROCESSO)

–usando metodi, strumenti e tecniche appropriate

–variabili a seconda del problema da risolvere, dei vincoli di sviluppo, e delle risorse disponibili.

•per ottenere

–software di qualità,

–nel rispetto dei tempi di consegna previsti

–e con costi contenuti (cost-effective).

Un **processo software** è un insieme di attività aventi per obiettivo lo sviluppo o l’evoluzione di un sistema software.

•Ogni processo software deve includere le seguenti attività fondamentali:

**–Specifica** –definizione di ciò che il sistema dovrà fare e dei vincoli di progettazione

**–Sviluppo** –progettazione e programmazione

**–Convalida** –si verifica che il software sia esattamente ciò che il cliente richiede

**–Evoluzione** –si modifica il software per adeguarlo a requisiti dell’utente e del mercato che cambiano.

Un **modello di processo software** è una descrizione semplificata del processo software, osservato da un determinato punto di vista.

•Diverse possibili descrizioni:

**–Workflow model** –descrive la sequenza di attività;

**–Data-flow model** –evidenzia le trasformazioni dei dati operate dalle attività del processo;

**–Role/action model** –I ruoli delle persone coinvolte nel processo e le relative responsabilità.

•Generici modelli di processo

–Waterfall;

–Iterative development;

–Component-based software engineering.

I **metodi** sono approcci strutturati per sviluppare software di qualità, a costi contenuti. (Es. SADT, JSD, OOA, OOD, etc..),forniscono indicazioni relativamente a:

•Modelli di sistema –Descrizioni dei modelli grafici da sviluppare;

•Regole –Vincoli che i modelli devono rispettare;

•Raccomandazioni –Consigli che determinano la corretta procedura di progettazione;

•Guida ai Processi –Descrizione delle attività di processo e della relativa organizzazione

I **CASE** sono sistemi software usati per aiutare le attività di processi software (es. analisi, modellazione, debugging, testing),sono spesso usati a supporto di specifici metodi.

•**Upper-CASE** –Supportano le attività alte del processo, quali la specifica dei requisiti e la progettazione;

•**Lower-CASE**–Supportano le attività successive, quali la programmazione, il debugging e il testing.

**LEZIONE 2 9/03**

Un **modello del ciclo di vita** del software(CVS)–o Software Life Cycle , SLC-è una caratterizzazione **descrittiva** o **prescrittiva** di come un sistema software viene o dovrebbe essere sviluppato.

Gli obiettivi sono:

-determinare l'ordine delle attività nello sviluppo e nell' evoluzione del software

–stabilire criteri di transizione per progredire da uno stadio di lavorazione al successivi.

**Code and Fix:** Approccio “primitivo” alla produzione del software che consiste nello scrivere codice e successivamente aggiustarlo per correggere errori, migliorare e/o aggiungere funzionalità.

Inadeguato per lo sviluppo odierno del software:

•Le grandi dimensioni dei sistemi rendono difficile la gestione non strutturata della complessità

•Difficile aggiustare/modificare/ristrutturare il codice

•Interpretazione sbagliate dei requisiti utente

•Processo non prevedibile (tempi/costi), qualità non misurabile

Questi problemi portarono alla necessità dei modelli di processo predicibili e controllabili, dunque strutturati. Avere un buon processo è fondamentale. La qualità del processo di sviluppo influenza: La qualità dei prodotti finali, i tempi per portare il prodotto sul mercato, i costi, le prestazioni su diversi progetti.

**Processo** : un insieme di attività concentrate nel tempo finalizzate alla realizzazione di un particolare output.

**Processo Software**: Un insieme strutturato di attività necessarie per lo sviluppo di un sistema software.

**Modello di Processo Software**: E’ una rappresentazione semplificata di un processo. Esso fornisce una descrizione del processo da una particolare prospettiva. Es. La prospettiva delle attività del processo, o quella dei ruoli del personale coinvolto

Esistono diversi tipi di software e non c’è un processo universale valido per tutti.

Tutti i processi software includono comunque le stesse attività fondamentali. Ogni processo si caratterizza per l’organizzazione delle attività che esso include, ma anche per: i prodotti, o deliverables che le sue attività producono, i ruoli assegnati alle persone coinvolte nel processo, le pre e post-condizioni di ciascuna attività.

**Studio di fattibilità**: -Se vale la pena o meno intraprendere un progetto.

Fornisce un documento con, Definizione del problema, Valutazione Costi/Benefici, Risorse finanziarie e umane, Soluzioni alternative, Tempi di consegna e modalità di sviluppo.

**Acquisizione, analisi e specifica dei requisiti**: Definire **cosa** il sistema dovrà fare, non **come**. Specificare le funzionalità e le qualità che deve possedere, **senza vincolare** la progettazione e l'implementazione. Richiede la conoscenza del dominio.

N.B. È un'attività critica. Un errore in questa fase può costare molto

Quali potrebbero essere i rischi a cui si va incontro durante un progetto, che quindi possono portarci a non consegnare in tempo, a sviluppare tutto?

Quando magari si accetta un progetto in cui il team che lavora sul progetto non è preparato al 100% e quindi bisogna chiedere l’ intervento di una persona esterna o bisogna chiedere al proprio team di intraprendere studi su quell’ argomento, questo aumenta i costi del progetto. Un danno legato alle piattaforme e la strumentazione che utilizzo. Ce ne sono tanti di motivi.

**Analisi e specifica dei requisiti**

**L'ingegneria dei requisiti** sviluppa metodi per raccogliere, documentare, classificare e analizzare i requisiti. Sono compiti dell'analista: Identificare i portatori di interesse (stakeholders),Esplicitare i requisiti, Conciliare i vari punti di vista (eventualmente contraddittori), Specificare i requisiti

Il risultato è la produzione del **Documento di Specifica dei Requisiti (DSR)**, comprensibile, preciso, completo ,coerente, non ambiguo, modificabile. Contiene una descrizione **del dominio, dei requisiti funzionali, non-funzionali, e requisiti del processo di sviluppo-manutenzione**

**Piano di Test di Sistema(PTS).**

**Strutturazione del sistema a diversi livelli di dettaglio (Progettazione)**

**Architettura generale**(hardware e software) assegna funzionalità a componenti di alto livello(architettura ad alto livello). **Architettura dettagliata** arriva alla definizione precisa dei moduli, delle loro funzionalità e delle interfacce.

**Produzione Documento di Progetto (DSP)** •Descrive i componenti del sistema, le loro interfacce, le relazioni tra di loro •Registra le decisioni significative e ne spiega le motivazioni •Importante per eventuali richieste future di cambiamento •La forma del documento è determinata dagli standard adottati dall'azienda.

**Produzione del codice** •Può essere soggetta a standard aziendali, a convenzioni. Test di Unità Test di Integrazione e di Sistema Test di Accettazione

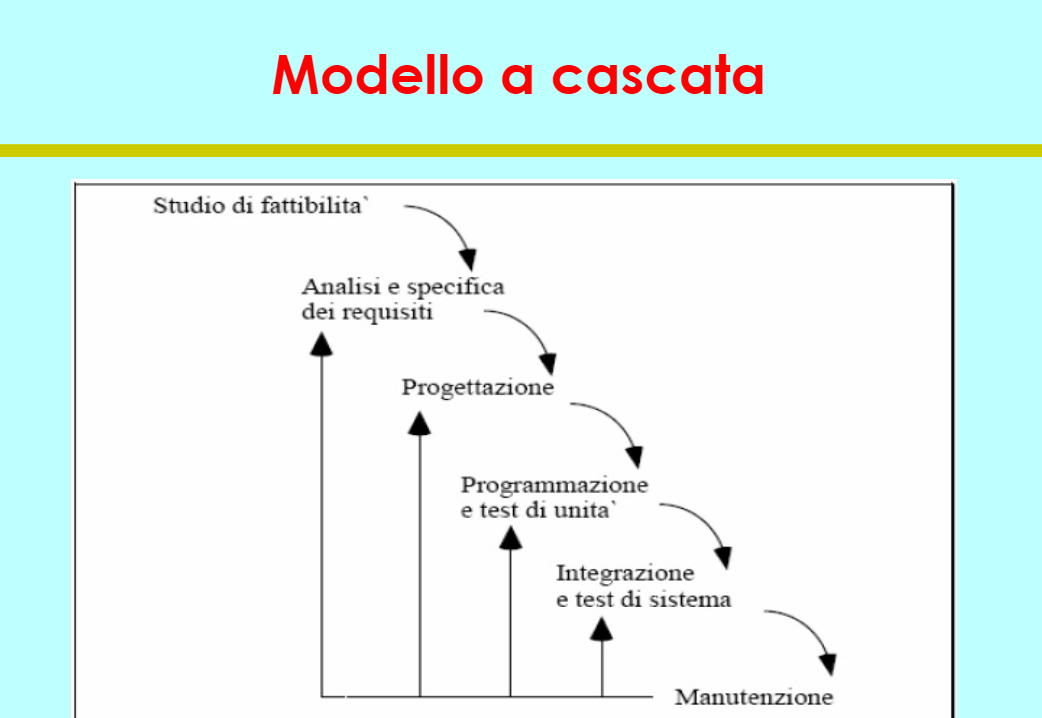
**Manutenzione**: Correttiva, Adattativa, Perfettiva .Tutte le attività descritte contengono compiti comuni: documentazione, verifica, gestione

**Modello a Cascata, organizzazione sequenziale delle fasi**

È un organizzazione strettamente lineare e sequenziale, ovvero, se non termina la fase precedente non inizia quella successiva, perché gli output della fase precedente sono gli input di quella successiva.

Ogni fase raccoglie un insieme di attività omogenee per metodi, tecnologie, skill del personale, etc.

Gli output di una fase vengono “congelati”, ovvero non sono più modificabili se non innescando un processo formale e sistematico di modifica. Il processo è guidato dalla produzione di documentazione.



La fine di ogni fase è un punto rilevante del processo (**milestone**). L'output di ogni fase è chiamato **deliverable**. La definizione precisa dei deliverable è importante per misurare il progresso di un progetto

**Criticità del modello a cascata**

È un modello ideale, che può essere solo approssimato nella pratica. È possibile caratterizzarlo mediante tre proprietà: **linearità, rigidità, monoliticità**. Nella pratica sono necessari **feedback**, uno dei grossi limiti perché quando vogliamo cambiare qualcosa dobbiamo per forza arrivare alla fase di manutenzione.

A causa della rigidità, si assume che i requisiti possano essere congelati nelle prime fasi, quando le conoscenze sono ancora preliminari. Monoliticità: se si commette un errore nei requisiti, viene fuori solo alla fine, dopo il rilascio, con costi esorbitanti. Stime dei costi difficili. Difficile anticipare i cambiamenti. Bassa evolvibilità. Può richiedere una quantità ingente di documenti. La difficoltà di produrre specifiche complete causa molta attività di manutenzione (che è particolarmente costosa). Questi problemi potrebbero essere risolti con il feedback.

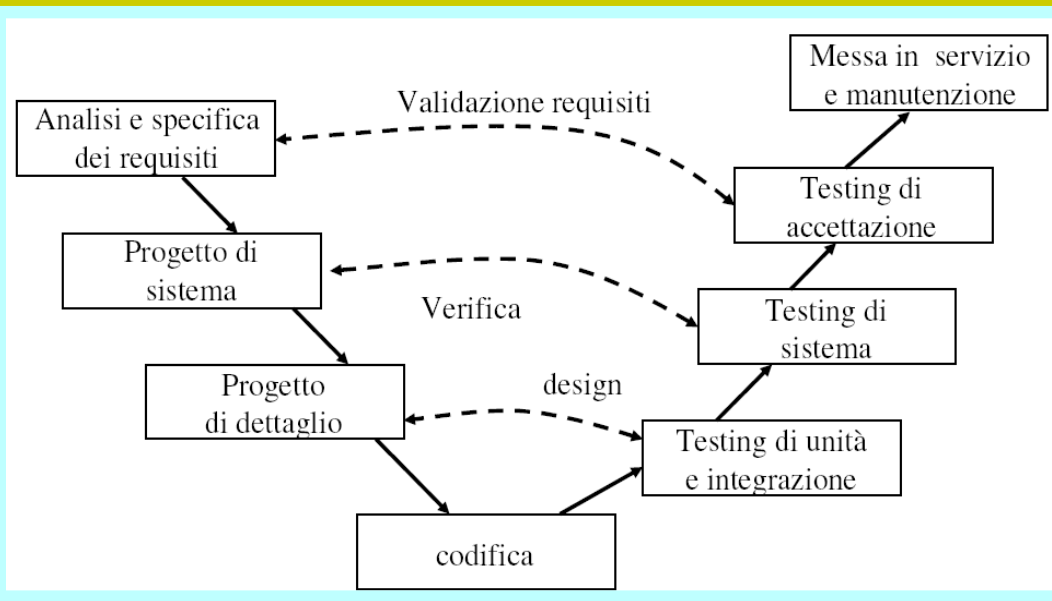
**V&V e Retroazione (verifica e validazione)**, variazione del modello a cascata con l’ inserimento di retroazione : si verifica la correttezza del risultato dell’ attività con la possibilità di ripetere l’ attività precedente.

**Verifica**: Stabilire la verità della corrispondenza tra un prodotto software e la sua specifica

**Convalida**: Stabilire l'appropriatezza di un prodotto software rispetto alla sua missione operativa.

Introduce dei **feedback in ogni fase**. Si possono così rilevare errori prima del rilascio. Tuttavia resta un modello che non “anticipa” i possibili cambiamenti. Può essere utile quando si prevede che il sistema sarà poco soggetto a cambiamenti.

**Il modello V** importante in ambito industriale, dove si progettano sistemi che devono essere sicuri (saftey critical) .



Le attività del ramo di sinistra sono collegate a quelle del ramo di destra. Durante le attività di sinistra, vengono progettati i test della fase a destra corrispondente. Se si trova un errore in una fase a destra si riesegue la fase a sinistra collegata. Si può iterare migliorando requisiti, progetto, codice.

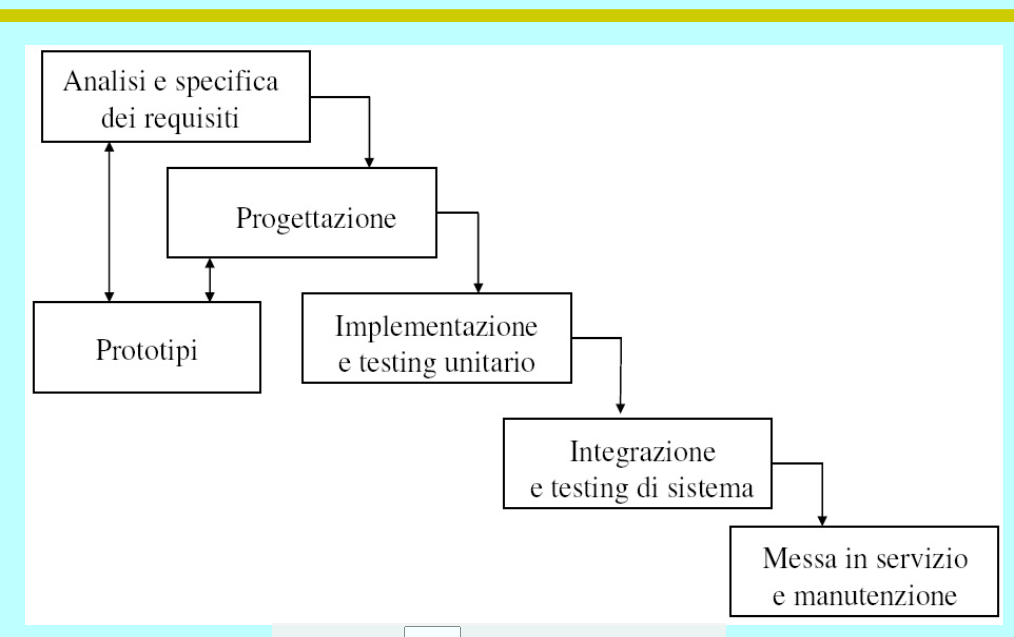
**LEZIONE 3 11/03**

I **modelli evolutivi** si basano sull’ idea che il software si deve evolvere,ovvero, i requisiti si evolvono nel tempo.

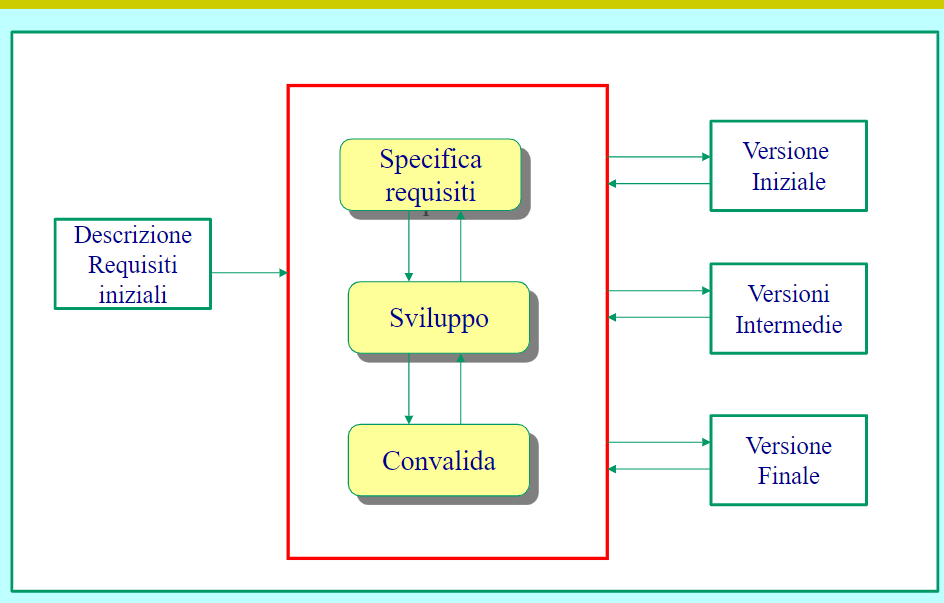
**Prototipo:** Modello approssimato dell'applicazione, il cui obiettivo è di ricevere feedback per affinare i requisiti

Il modello prototipale è dunque adatto quando i requisiti non sono completi e non ambigui. Si usa il prototipo per raccogliere feedback dagli opportuni stakeholder

Due modelli fondamentali:

**Sviluppo con prototipo usa e getta**: Realizzazione di una prima implementazione (prototipo), più o meno incompleta, da considerare come una ‘prova’, con lo scopo di: accertare la Fattibilità del prodotto, validare i Requisiti. Un prototipo di scarsa qualità che di solito viene buttato. Il prototipo può essere utilizzato i ogni fase del processo per affinare requisiti, analisi, progettazione ecc. Questo approccio (ispirato al principio “Do it twice”) consente di ridurre errori sui requisiti.

**Sviluppo esplorativo (prototipo evolutivo)**: rilascio delle versioni che si cerca di fare evolvere attraverso i feedback del nostro cliente, per appunto consegnarlo integrato o addirittura completo. Gli svantaggi sono che non si può prevedere quanto tempo e iterazioni ci vorranno per completare il lavoro, questo modo di lavorare può generare sistemi spesso mal strutturati. Il vantaggio sta nel fatto che questo è un approccio agile, ovvero , avendo controlli continui con i clienti si consegna un prodotto sempre affine con la richiesta. Può essere applicabile a sistemi interattivi di piccole e medie dimensioni. Per sviluppare alcune parti di sistemi di grandi dimensioni.



Nei **modelli incrementali** piuttosto che consegnare tutto in una volta, lo sviluppo e la consegna sono eseguiti per incremento. Dove ogni incremento rilascia parte delle funzionalità richieste. In ogni step si usa il modello a cascata.

Ai requisiti Utente vengono associati livelli di priorità e quelli a priorità maggiore vengono rilasciati con i primi incrementi. Una volta partito lo sviluppo di un incremento, i relativi requisiti devono essere congelati, mentre i requisiti coinvolti in incrementi successivi possono continuare ad evolvere. I servizi comuni possono essere implementati all’inizio del processo, o quando una funzione è richiesta da un dato incremento.

I vantaggi sono che i clienti non devono aspettare il sistema completo per la consegna, ma possono disporre al più presto dei requisiti più critici, attraverso i primi incrementi. I primi incrementi possono essere usati come prototipo per aiutare a definire i requisiti degli incrementi successivi. Si riduce il rischio di un fallimento totale del progetto. E’ possibile gestire esigenze di cambiamento dei requisiti. I servizi a più alta priorità saranno anche testati più intensamente degli altri.

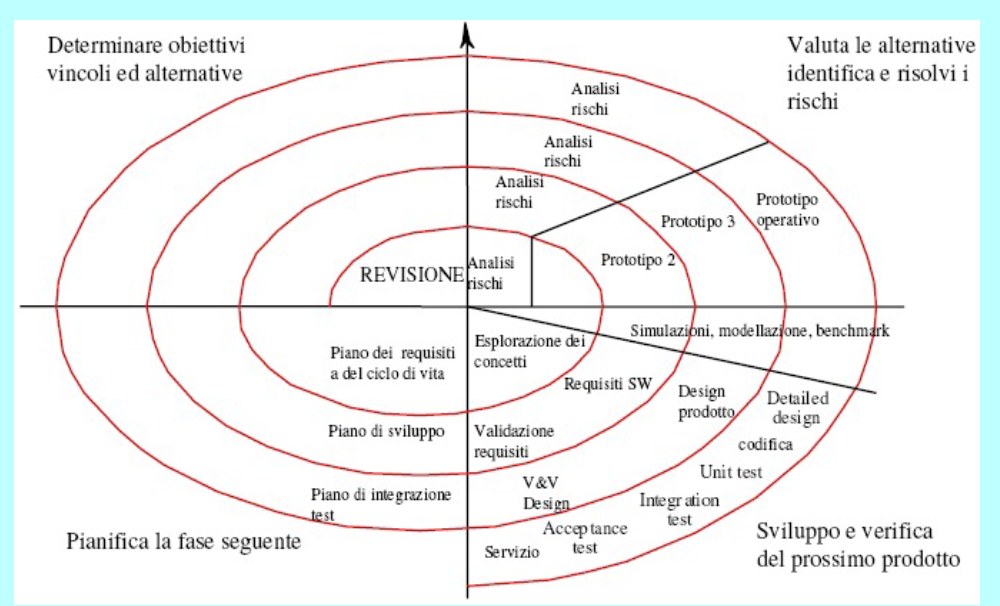
Gli svantaggi sono lo sviluppo incrementale può essere problematico quando gli utenti hanno bisogno di un sistema funzionante completo che sostituisca un sistema preesistente. Può essere difficile identificare le funzionalità comuni (richieste da tutti i requisiti), giacchè bisogna prima attendere che gli incrementi siano completati per avere ben chiari tutti i requisiti. Quando la specifica completa deve far parte del contratto di sviluppo del sistema, questo modello non è adeguato.

Il **modello trasformazionale** si parte da requisiti formalmente e si procede formando una serie di specifiche formali in una descrizione meno astratta fino ad arrivare proprio ad un codice. Le specifiche vengono convalidate prima si essere trasformate, quelle eseguibili possono essere viste come un prototipo evolutivo.

Le trasformazioni possono essere eseguite manualmente o supportate da appositi strumenti. l processo di trasformazione può avvantaggiarsi di componenti riusabili. Il processo si avvale della storia dello sviluppo, propriamente immagazzinata, per il supporto a future richieste di cambiamenti

**Modello a spirale:** l’ obiettivo è fornire un quadro di riferimento per la progettazione dei processi. Viene visto come un **meta-modello** ovvero configurare il proprio modello in base ai rischi. Il rischio è visto come una circostanza potenzialmente avversa in grado di pregiudicare il processo di sviluppo e la qualità del prodotto. **Gestione dei rischi**: “identificare, affrontare ed eliminare i rischi prima che insorgano problemi seri o causa di re-implementazioni costose”.

Il processo è rappresentato come una spirale, piuttosto che una sequenza di attività con retro-azioni. Ogni giro nella spirale rappresenta una fase del processo. Non prevede fasi prefissate a priori, ma i cicli sono definiti in base al caso specifico. C’è una esplicita gestione dei rischi che vengono valutati e risolti durante tutto il processo.



Nel 3° settore si può utilizzare: Modello a cascata, modello evolutivo o Trasformazionale in base ai requisiti.

**Lezione 4 15/03**

**Ingegneria dei requisiti**, è il processo condotto per stabilire le funzionalità/servizi e le caratteristiche di qualità che il cliente richiede al sistema, i vincoli di sviluppo (eventuali paletti che il cliente ci impone) e i vincoli operativi. I requisiti sono la descrizione di tali funzionalità e caratteristiche, e dei relativi vincoli.

L’ingegneria dei requisiti, sviluppa metodi per **raccogliere, documentare, classificare, analizzare e gestire i requisiti**. È di importanza fondamentale, in quanto errori in questa fase **si propagano** nei passi successivi e il costo per porvi rimedio cresce nel tempo.

Requisiti non chiari, incorretti o incompleti possono portare a: Consegne ritardate -> Costi maggiori -> Insoddisfazione dell’utente. Comportamenti errati e imprevisti. Prodotto software corretto (realizzato conformemente alle specifiche) ma che non soddisfa i bisogni dell’utente (con eventuali contestazioni, mancati pagamenti, etc.). Alti costi di manutenzione.

**Un requisito** è una frase in linguaggio naturale che descrive:

•qualcosa che il sistema dovrà fare (**requisito funzionale**),

•una proprietà o vincolo che si desidera per il sistema (**requisito non funzionale**), e che uno o più **stakeholders**(**portatori di interesse**)richiedono al sistema stesso.

Una frase breve che descrive **cosa**, ma non la soluzione (come). Ogni requisito deve essere riferito ad almeno uno stakeholder.

Un requisito può variare da una **descrizione astratta**, ad alto livello, di qualcosa che il sistema farà, di una proprietà o vincolo, ad una **specifica formale** di una proprietà desiderata.

**Tipi di requisiti**

**Requisiti Utente**, frasi in linguaggio naturale (e diagrammi) ad alto livello, relativi a funzionalità/servizi/dati che il sistema deve fornire e i suoi vincoli operativi. Scritti per i clienti.

**Requisiti di Sistema**, un documento strutturato che fornisce una descrizione dettagliata di funzionalità/servizi/vincoli operativi del sistema. Può essere parte del contratto fra cliente e sviluppatore.

**Requisiti Funzionali**, descrivono funzionalità/servizi del sistema (cosa si richiede che faccia). due livelli di astrazione: Requisiti funzionali utente: frasi ad alto livello su ciò che il sistema farà. Requisiti funzionali di sistema: descrizioni dettagliate dei servizi.   
Quali input il sistema dovrà accettare. Quali output il sistema dovrà produrre. Quali dati il sistema dovrà immagazzinare che potranno essere usati da altri sistemi. Quali elaborazioni il sistema dovrà svolgere. Requisiti di tempificazione e di sincronizzazione delle elaborazioni.

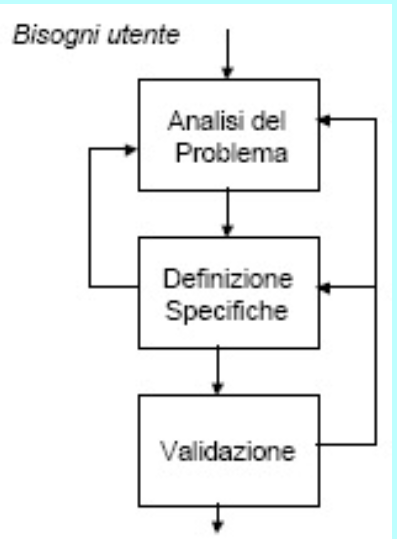
**Requisiti Non Funzionali**, caratteristiche di qualità, vincoli sul sistema/ambiente, vincoli sul processo di sviluppo, vincoli tecnologici. Definiscono o limitano le proprietà del sistema (es. Affidabilità, tempi di risposta, memoria occupata). Possono vincolare anche il processo di sviluppo da adottare (es. Uso di particolari standard per la documentazione, su sistemi CASE, linguaggi di codifica, metodi di sviluppo). Possono essere più critici dei req. funzionali: se non sono soddisfatti, il sistema è inutile. Tipi di requisiti non funzionali. Requisiti del prodotto, specificano il comportamento del prodotto (usabilità, efficienza, affidabilità, portabilità). Requisiti organizzativi, derivano dalle politiche e procedure dell’organizzazione del cliente e dello sviluppatore (es. Regole organizzative del dominio, standard di processo da usare, piattaforme, requisiti di consegna, etc.).Requisiti esterni, derivano da fattori esterni al sistema e al suo processo di sviluppo (come i requisiti di interoperabilità, legislativi, etici, etc.). I requisiti non funzionali possono essere difficili da definire precisamente, e quindi difficili da verificare. L’utente li specifica come obiettivi. Requisito verificabile, una frase che contiene qualche misura che potrà essere oggettivamente verificata.

**Requisiti di dominio**, requisiti che derivano dal dominio applicativo. Derivano dal dominio di applicazione, piuttosto che da necessità degli utenti. I requisiti di dominio possono essere nuovi requisiti funzionali, vincoli su altri requisiti funzionali, o possono delineare calcoli da effettuarsi. Se non sono soddisfatti, il sistema potrebbe essere inutilizzabile. Problemi dei requisiti di dominio. Comprensibilità : I requisiti sono espressi nel linguaggio del dominio; Tale linguaggio potrebbe non essere immediatamente comprensibile per gli ingegneri del software. Esplicitazione: Gli specialisti del dominio conoscono così bene il dominio stesso, da lasciare fuori dai requisiti informazioni che sembrano loro ovvie (e invece non lo sono per l’ingegnere del software...). L’analista deve rendere espliciti eventuali requisiti di dominio impliciti

I requisiti devono essere completi e consistenti. **Completezza**, tutti i requisiti richiesti devono essere presenti. **Consistenza**, non ci devono essere definizioni contraddittorie.

**LEZIONE 5 16/03**

Analisi dei requisiti, processo di comprensione del dominio del problema al fine di: stabilire le funzionalità, i servizi, le interfacce, le prestazioni, i fattori di qualità che il committente richiede al sistema da sviluppare, ed i vincoli di utilizzo e sviluppo; produrre un Documento di Specifica dei Requisiti Software, che sia completo, consistente, non ambiguo e comprensibile; predisporre un piano di test. **Definisce Cosa il sistema deve fare, non il Come**

**Analisi Comprensione** dei bisogni (es.: cosa deve fare il sistema software). **Specifica dei requisiti**, trasformazione dei requisiti in un documento di specifiche tecniche e funzionali caratterizzanti il sistema. **Validazione delle specifiche**, le specifiche, formalizzate, vengono riviste con l’utente/committente.

**Analisi dei requisiti**

• Processo di valutazione delle necessità del committente, con redazione di un documento di analisi dei requisiti, di solito in linguaggio naturale

• La raccolta dei requisiti avviene tramite di interviste svolte dagli analisti con gli stakeholder (committente, utente, installatore, ...), lo studio nelle norme di dominio, etc.

**Specifica dei requisiti**

• Processo di schematizzazione dei requisiti di un sistema, con redazione di un documento secondo un modello (template) prefissato

• È la base del contratto di fornitura

Alcune linee guida

Adottare un formato standard (interno all’azienda/organizzazione) per scrivere i requisiti. Adoperare frasi semplici e brevi. Evitare mescolanza di requisiti. Una frase non deve esprimere più requisiti. Usare il linguaggio naturale in maniera coerente. Es. distinguere i requisiti obbligatori (indicati con “deve”) da quelli desiderati (indicati con “dovrebbe”). Usare diversi stili di testo per evidenziare i punti chiave del requisito. Evitare il gergo informatico.

Specifica dei requisiti di Sistema. Sono specifiche più dettagliate dei requisiti utente che serviranno come base per la progettazione, ma non sono le specifiche di come verrà sviluppato. Spesso sono inclusi nel contratto fra cliente e sviluppatore. In alcuni casi potrebbero però includere già riferimenti all’architettura che il sistema avrà , per poter strutturare meglio i requisiti relativamente ai sotto-sistemi, per soddisfare specifici requisiti non-funzionali, per esigenze di inter-operabilità con altri sistemi.

Spesso anche i requisiti di sistema vengono espressi in linguaggio naturale (NL). Alcune alternative al NL: Linguaggio naturale strutturato e stilizzato (con uso di Moduli/Template standard) . Modelli grafici (come gli use case e sequence diagrams). Specifiche matematiche formali (con macchine a stati, notazioni insiemistiche, o altro)

La libertà di chi scrive i requisiti è limitata usando un ‘**modello’** predefinito e standard per tutti i requisiti. Viene limitata anche la terminologia usata per descrivere il requisito. Il **vantaggio** è che si conserva l’espressività del linguaggio naturale, ma al tempo stesso si impone un grado di uniformità per descrivere le specifiche.

**Specifiche basate su un Template**

Definizione della funzione o entità. Descrizione degli input e relative sorgenti. Descrizione degli output e loro destinazione. Informazioni richieste per l’elaborazione o altre entità del sistema usate (ossia richieste). Pre-condizioni e Post-condizioni. Eventuali effetti collaterali della funzione.

**LEZIONE 6 18/03**

**Il Documento di Specifica dei Requisiti (SRS)**, è una dichiarazione ufficiale di ciò che gli sviluppatori dovrebbero implementare (attenzione: il COSA ma non il COME). Deve prima includere i requisiti utente (RU), e poi una specifica dettagliata dei requisiti di sistema (RS). Talora RU e RS sono contenuti nello stesso documento, talora in documenti separati.

Il documento di Specifica dei Requisiti Software(SRS) costituisce il punto di convergenza di tre diversi punti di vista: **cliente, utente, sviluppatore**. Un SRS serve a: costituire la base per la successiva progettazione, costituire la base per la validazione del prodotto finale, costituire la base per il contratto (evitando controversie, avvocati, tribunali…).

Di solito né il fornitore né il committente sono in grado, da soli, di scrivere un buon SRS. Il committente non ha la necessaria conoscenza dei processi software per scrivere un buon SRS. Il fornitore non ha una conoscenza perfetta del dominio applicativo, e non può esprimere le effettive necessità. Il processo di raccolta dei requisiti è un processo di **confronto** fra necessità e possibilità (ovvero tra committente e il fornitore). Un SRS di qualità è il pre-requisito per un software di alta qualità, un errore nell’SRS produrrà errori nel sistema finale. Un SRS di qualità riduce i costi di sviluppo, correggere un errore dell’SRS dopo lo sviluppo costa 100 volte più che correggerlo durante la fase di Analisi...

**Lo standard IEEE 830 -1998**

Definisce la struttura generica di un documento dei requisiti, che deve essere istanziata per ogni specifico sistema. Contiene più capitoli: 1.Introduzione 2.Descrizione Generale 3.Requisiti Specifici 4.Appendici 5.Indice.

Perché viene utilizzato uno standard? Perché se ogni azienda usasse un proprio standard interno quando quel documento va in mano ad un'altra azienda potrebbe non capire il documento.

**Le caratteristiche di qualità di un buon SRS**

**Corretto**, se ogni requisito presente nell’SRS è realmente richiesto/necessario per gli utenti del sistema finale. **Completo**, se ogni funzione richiesta al software ed il suo comportamento rispetto ad ogni possibile input è specificato. **Non** **Ambiguo**, ogni requisito ha una sola interpretazione (formale vs. informale). Per ogni caratteristica usare sempre e solo lo stesso termine, se un termine può avere diversi significati, in base al contesto, specificarlo in un Glossario. È utile costruire un Data Dictionary, che dia una interpretazione univoca ai termini usati .**Verificabile**, se è possibile verificare che il sistema realizzi ogni requisito (richiede la non ambiguità dei requisiti). **Consistente**, se nessun requisito è in contraddizione con gli altri. **Modificabile**, se la struttura e lo stile dell’SRS sono tali da consentire facili modifiche, preservando consistenza e completezza (un SRS con ridondanze non si modifica facilmente). **Tracciabile**, se l’origine di ciascun requisito è chiara e può essere referenziata nello sviluppo futuro (def. IEEE), forward traceability: requisito collegabile a qualche elemento del progetto e del codice, backward traceability: dal progetto e dal codice è possibile risalire al requisito corrispondente. Quando un requisito è conseguenza di un altro, i due devono essere tracciabili in avanti ed indietro.

Validazione delle specifiche: Revisione delle specifiche al fine di trovare. Errori , aree poco chiare, informazioni mancanti, inconsistenze, requisiti contrastanti o irrealizzabili. Nel caso dell’SRS, l’obiettivo è assicurare che esso rifletta accuratamente e con chiarezza i requisiti richiesti al software.

In certi settori (ferroviario, avionico, automotive, spazio, elettromedicali, etc..) la specifica (SRS) così descritta è assolutamente obbligatoria. In ambienti meno restrittivi all’ inizio conviene sempre produrre un SRS, poi casi d’ uso.

**Gestione dei requisiti:** Man mano che il processo di sviluppo avanza, è probabile che vi siano cambiamenti/raffinamenti nei requisiti. **Requirements Management**: il processo di gestione di cambiamenti nei requisiti di sistema. Controllo dei cambiamenti: stima dell’impatto dei cambiamenti sugli altri requisiti e sulle altre attività.

**LEZIONE 7 22/03**

I **casi d’ uso** sono storie scritte, testuali, di qualche attore che usa un sistema per raggiungere degli obiettivi, l’ attore produce delle informazioni al sistema e viceversa il sistema all’ attore. Influenzano molti aspetti di un progetto, dall’analisi alla progettazione ad oggetti. Utilizzeremo i casi d’ uso per definire i requisiti di un sistema software utilizzando un linguaggio abbina la notazione grafica al linguaggio locale.

L’analisi dei casi d’uso viene impiegata in fase di analisi per modellare il comportamento esterno del sistema da sviluppare, senza specificare come esso viene realizzato, (tutta l’attenzione viene soffermata sul colloquio, l’ attore fa delle cose e il sistema risponde fornendone altre). Il sistema viene visto, al suo interno, come una scatola nera(black box), quindi non vediamo cosa c’è al suo interno. Il comportamento del sistema è analizzato dal punto di vista dei suoi possibili utenti.

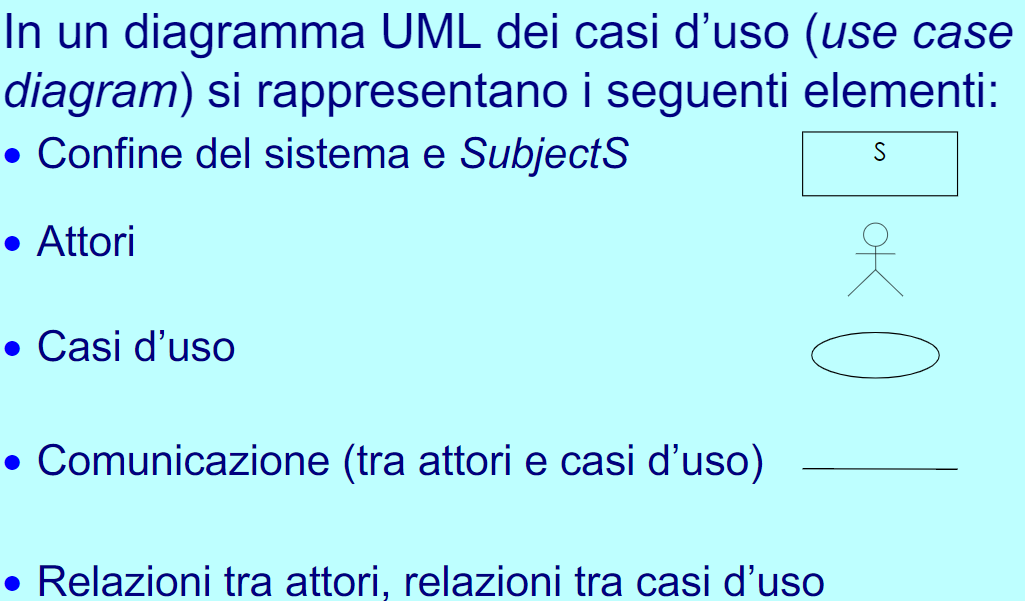
La tecnica dei casi d’uso è parte integrante della specifica UML (Unified Modeling Language). Il linguaggio UML ci da una specifica grafica per rappresentare i diagrammi dei casi d’ uso. **UML** è un linguaggio di modellazione a oggetti molto diffuso per **specificare, progettare, o documentare** sistemi software.

Esistono molti tipi di diagrammi UML , tali diagrammi sono adatti a fornire **viste** di un sistema software da diverse prospettive e a vari gradi di astrazione.

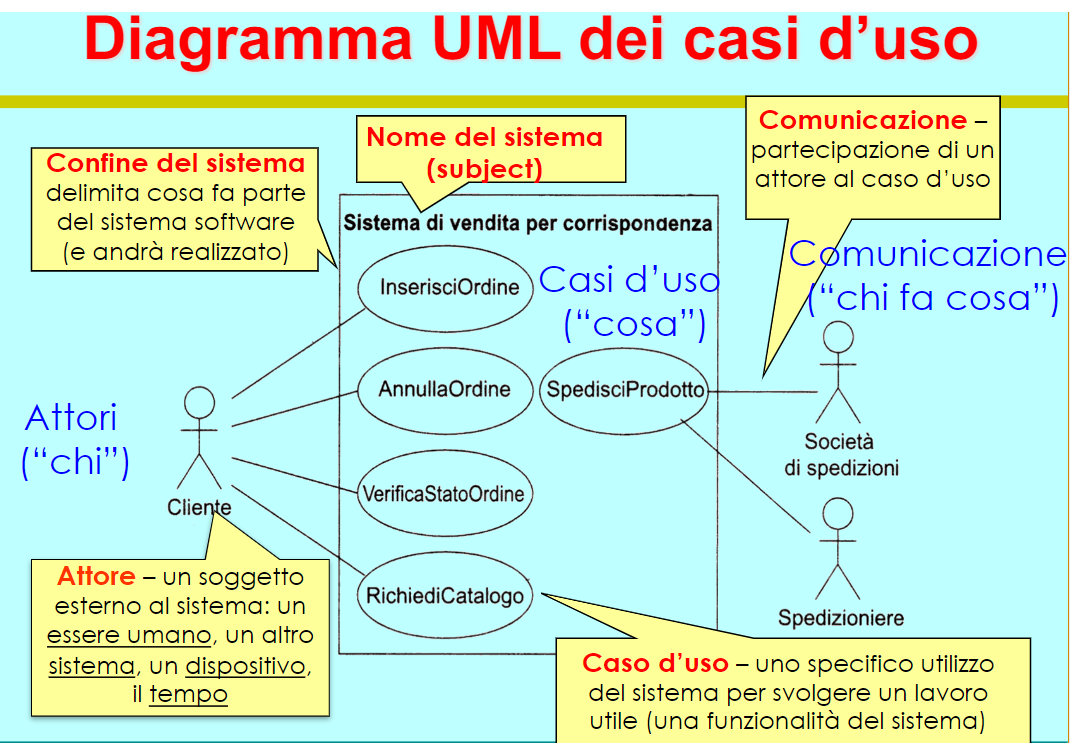
Diagrammi per modellare gli aspetti statici e dinamici (comportamentali) : Diagramma dei casi d’uso (use case diagram) cosa puoi fare con un sistema software e la descrizione dei modi di usare un sistema da parte dei suoi attori, Diagramma delle classi (class diagram), Diagramma di sequenza(sequence diagram), Diagramma di comunicazione (communication diagram), Diagramma di stato (state–chartdiagram),Diagramma delle attività (activity diagram).

Diagrammi di implementazione: Diagramma dei componenti (component diagram), Diagramma di distribuzione (deployment diagram).

Un caso d’uso è una tipica interazione tra un attore ed il sistema per svolgere un’unità di lavoro utile che permette di aggiungere all’ attore un obiettivo (ci deve essere un obiettivo ben preciso che si porta a termine con un’ unità di lavoro). Un caso d’uso non rivela l’organizzazione interna del sistema (non dice come è fatto il sistema, ma cosa deve fare). L’insieme dei casi d’uso rappresenta le funzionalità del sistema (i servizi che esso offre agli attori), infatti ragionare sui casi d’uso aiuta a scoprire i requisiti funzionali del sistema. La descrizione di un caso d’uso specifica cosa accade nel sistema in seguito ad uno stimolo. Generalmente lo **stimolo** parte da un attore, ma può anche essere il sistema stesso ad iniziare il caso d’uso (es.: aggiornamento automatico inventario). Un caso d’uso corrisponde ad un compito , che l’attore chiede al sistema di eseguire (l’attore inizia il caso d’uso) oppure, che il sistema esegue autonomamente (il sistema inizia il caso d’uso).

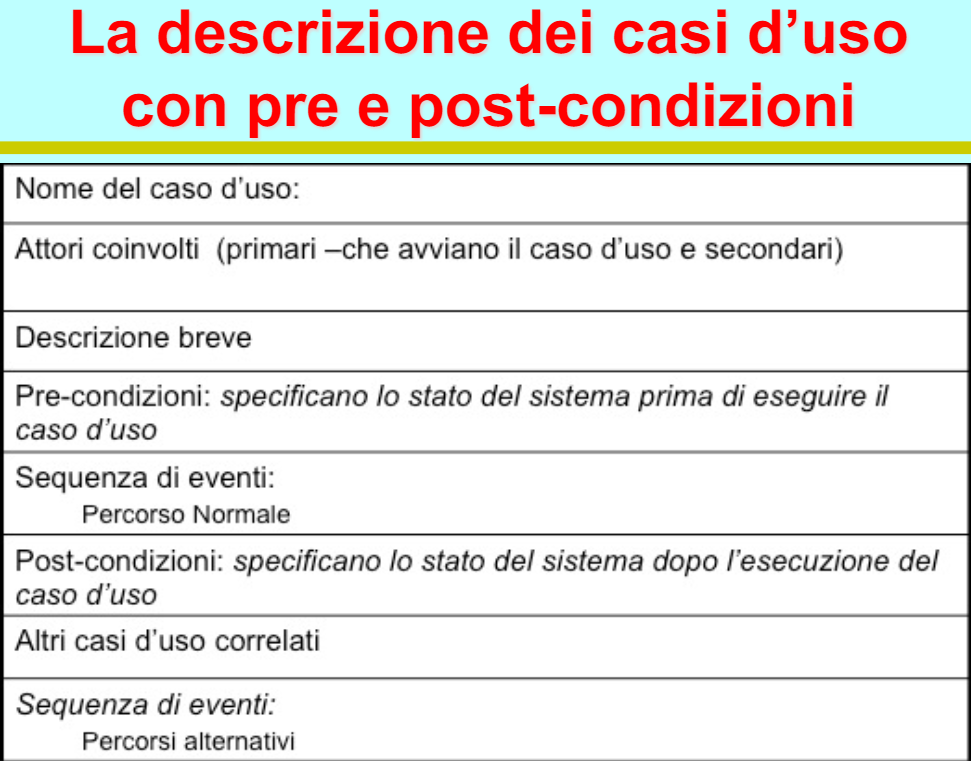
****

**Subject** (o Confine del sistema), rappresenta il limite tra ciò che è interno al sistema da sviluppare e ciò che è esterno ad esso. Gli **attori** sono esterni al sistema. Un attore di un caso d’uso può essere : Una classe di persone fisiche(es. Fornitore), un altro sistema software(es. Sistema di contabilità), un dispositivo hardware esterno(es. Sensore), il tempo(es.: per una funzionalità periodica). Un attore di un caso d’uso è detto: **Attore primario**, se fornisce lo stimolo che avvia il caso d’uso o **Attore secondario**, se interagisce con il caso d’uso dopo che è stato avviato.



Un caso d’uso va descritto tramite un insieme di **scenari** di interazione tra gli attori ed il sistema. Uno scenario è una sequenza di azioni ed interazioni fra sistema ed attori. Uno scenario definisce cosa accade nel sistema in seguito all’evento di innesco: come e quando il caso d’uso inizia, chi inizia il caso d’uso, interazione tra attore/i e caso d’uso e cosa viene scambiato, come e quando c’è bisogno di dati memorizzati o di memorizzare dati, come e quando il caso d’uso termina. Per ciascun caso d’uso sono previsti **scenari normali** (quelli che ti portano a chiudere con successo l’ unità di lavoro) e **scenari alternativi(o eccezionali)**(ovvero quelli che non ti permettono di portare a termine l’ azione).

**Stili di descrizione degli scenari** : testuali, con un flusso chiaro di eventi da seguire o diagrammatici: diagramma di stato, di sequenza, di interazione, essendo diagrammi utili anche in fase di progettazione. Per ciascun caso d’uso, è opportuno documentare gli scenari, ad es. con una scheda strutturata per ciascuno scenario. UML non definisce uno standard, ogni azienda/organizzazione ne definisce uno proprio.



Un percorso alternativo viene descritto da due parti (condizione e gestione)

Le alternative possono essere descritte attraverso **If** oppure attraverso percorsi alternativi. **Cicli While o For** possono essere usati per racchiudere gruppi di passi che devono essere ripetuti più volte.

**Linee guida per la descrizione degli scenari**: **1**.Scrivere in stile essenziale, senza riferimento all’interfaccia utente. Concentrarsi sullo scopo reale dell’utente e non su UI.

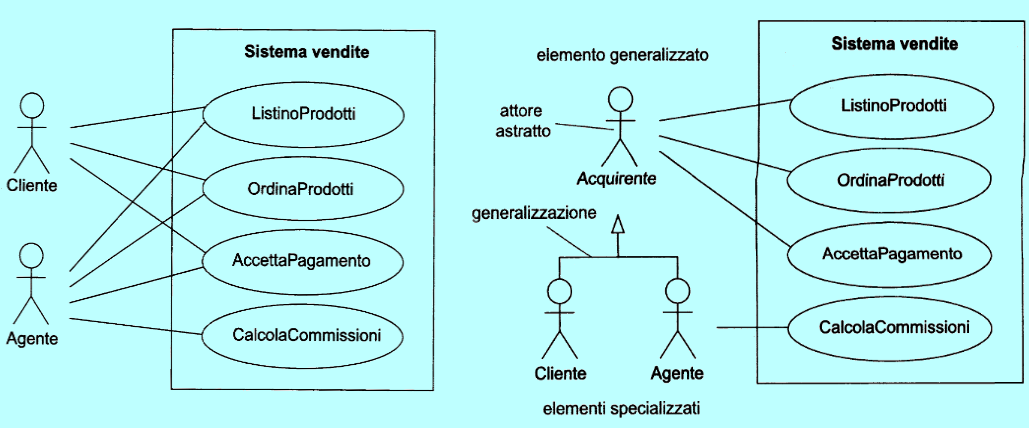
**2**.Descrivere casi d’uso concisi e completi. Esempio: Il Sistema autentica il Cassiere e non: il sistema richiede username e password.

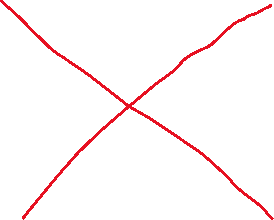
**3**.Descrivere casi d’uso a scatola nera. Concentrarsi sulle responsabilità del sistema (cosa fa, non come) e non anticipare scelte realizzative. Esempio errato: Il sistema memorizza la vendita in una base dati (o peggio: Il Sistema esegue una istruzione SQL Insert per la vendita...).

Nella descrizione di un caso d’uso NON devono essere indicati dettagli che rivelino le scelte di progetto del software. Non possono esserci riferimenti a specifici file/ moduli/ specifiche interfacce utente, a meno che essi non rappresentino dei **vincoli** (ad esempio dei sistemi già esistenti con i quali bisogna integrarsi).

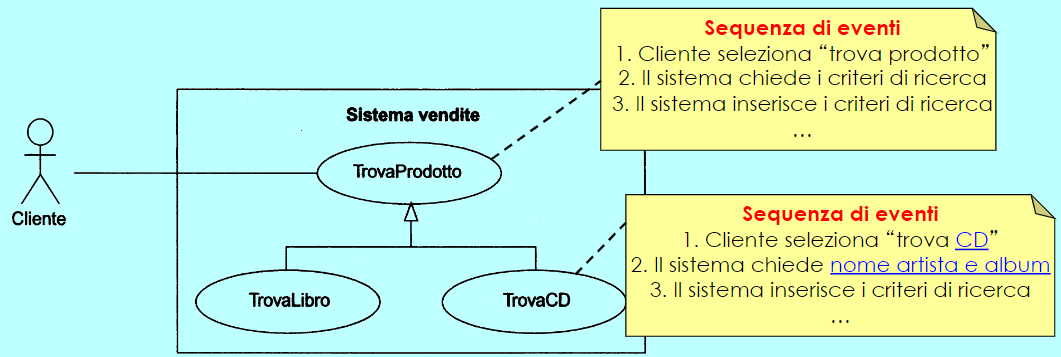
In un diagramma dei casi d’uso si possono specificare relazioni tra attori e relazioni tra casi d’uso.

**Relazioni tra attori**: **relazione di generalizzazione tra attori** permette di astrarre ruoli comuni a più attori e di semplificare i diagrammi. Gli attori specializzati (o figli) ereditano i ruoli e le relazioni dell’attore generalizzato ed hanno inoltre altri ruoli.

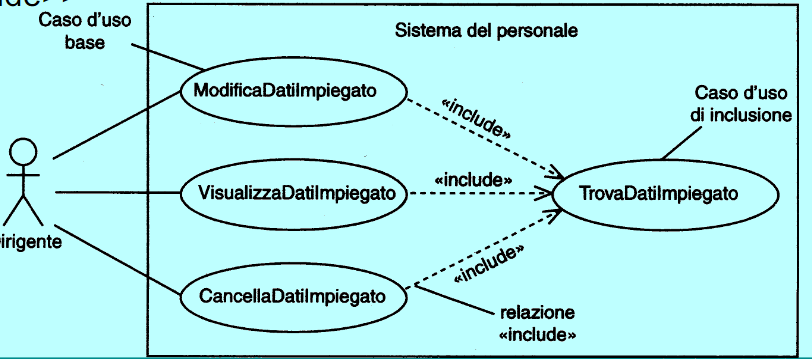




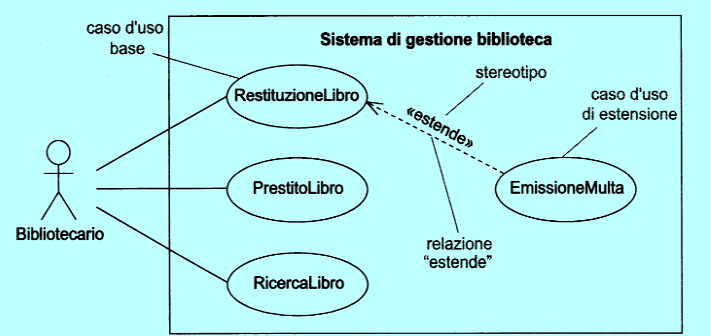
**Relazioni tra casi d’uso**: **Relazione di generalizzazione** simile alla generalizzazione fra attori. Un caso d’uso figlio eredita comportamento e significato del caso d’uso padre ,il caso d’uso figlio può aggiungere o modificare il comportamento del padre.



**Relazione d’uso <<include>>** formalizza i casi in cui più casi d’uso racchiudono una serie di azioni comuni. Il comportamento comune a più casi d’uso diventa un caso d’uso che è incluso nei casi d’uso di partenza. Il caso d’uso base è incompleto senza il caso incluso. Rappresentato graficamente come una dipendenza stereotipata (uno stereotipo è incluso nelle freccette) come <<include>>.



**Relazione d’estensione <<extends>>** si usa per modellare una sequenza opzionale di eventi oppure casi eccezionali. La sequenza opzionale definisce un nuovo caso d’uso che estende il caso d’uso di partenza e ne varia il comportamento “normale”. Nel caso d’uso esteso (base) si aggancia ad uno o più punti d’estensione. I casi d’uso di estensione aggiungono un comportamento in corrispondenza di questi punti. Il caso d’uso base si può svolgere anche senza i casi d’uso d’estensione. Possiamo inserire nel caso d’ uso generale l’ Extension Point (XP) ovvero il vincolo che fa scattare l’ estensione.



**LEZIONE 8 23/03**

**Estensione VS Scenari Alternativi**

La soluzione con trecasi d’uso è più utile nel caso in cui i casi d’uso d’estensione abbiano ulteriori legami e/o siano direttamente richiamabili dall’utente. La soluzione con un solo caso d’uso e più scenari fornisce una vista più compatta del sistema, e potrebbe essere preferibile se si vuole realizzare un modello dei casi d’uso meno dettagliato.

Per creare il diagramma dei casi d’uso (partendo dai requisiti informali, espressi in linguaggio naturale), procedere come segue, iterando per raffinamenti successivi:

**Scegliere i confini del sistema.** Quali responsabilità rientrano nei confini del sistema che stiamo modellando ? È costituito solo da una applicazione software? Oppure è un sistema composto da hardware e software insieme? Oppure è un sistema completo, più una persona / organizzazione che lo usa?

**Identifica gli attori.** Identifica gli attori che interagiscono con il sistema per eseguire qualche compito. Identifica gli attori che necessitano del sistema per svolgere qualche compito. Identifica gli attori cui il sistema si rivolge per svolgere qualche compito, considera sia i compiti principali che quelli di supporto al sistema, quali manutenzione ed amministrazione. Raggruppa le persone identificate secondo i loro ruoli (responsabilità) rispetto al sistema. Identifica altri sistemi software e dispositivi esterni che interagiscono con il sistema per svolgere qualche compito (potrebbero essere altri attori).

**Identifica i casi d’uso.** Per ogni attore: identifica i compiti o funzioni di più basso livello che l’attore deve essere in grado di eseguire , identifica i compiti che il sistema richiede che l’attore esegua. Raggruppa compiti e funzioni in casi d’uso, i casi d’uso devono corrispondere ad un obiettivo specifico per l’attore o per il sistema, raggruppa funzioni che sono eseguite in sequenza o che sono innescate dallo stesso evento, il caso d’uso deve essere nè troppo grande nè troppo piccolo (non decomporre funzioni complesse in casi d’uso per ogni sottofunzione). Nomina il caso d’uso sintetizzando la funzionalità svolta

**Definisci la comunicazione fra attori e casi d’uso**. Ogni attore deve partecipare ad almeno un caso d’uso. Ogni caso d’uso deve avere almeno un attore con cui comunica. Se due attori partecipano agli stessi casi d’uso considera la possibilità di combinarli in un unico attore

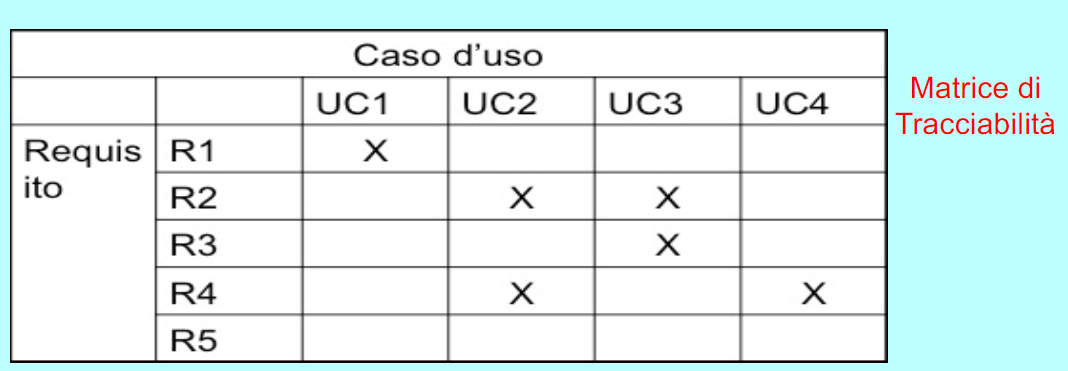
**Descrivi i casi d’uso.** Considera sia lo scenario principale che scenari alternativi ed eccezionali. Per ogni scenario specifica: Quale evento scatena il caso d’uso (trigger), chi inizia il caso d’uso, quali precondizioni sono da ritenersi verificate nel momento in cui il caso d’uso inizia, quali sono le interazioni tra il/gli attore/i e il sistema e quali dati/comandi vengono scambiati, come e quando c’è bisogno di memorizzare dati, come e quando il caso d’uso termina, quali post-condizioni (garanzie) sono da ritenersi verificate nel momento in cui il caso d’uso termina. Se due casi d’uso hanno comportamenti leggermente diversi e gli stessi attori, considera la possibilità di avere un unico caso d’uso con scenari alternativi

**Struttura i casi d’uso.** Identifica le varie relazioni (generalizzazione, inclusione, estensione) che potrebbero servire a migliorare la leggibilità del diagramma.

Nel diagramma è contenuto solo un piccolo sottoinsieme delle informazioni contenute nelle descrizioni. Quindi, il diagramma dei casi d’uso non può mai essere considerato separatamente dalle descrizioni.

**Relazione tra requisiti funzionali e casi d’uso**. Un requisito funzionale rappresenta una precisa funzionalità che il sistema software deve offrire, il caso d’uso invece è un modo di utilizzare il software da parte dei suoi attori, quindi un caso d’uso può includere più requisiti funzionali al suo interno, ma anche il viceversa.

È importante incrociare gli archivi dei requisiti funzionali e dei casi d’uso per verificare la reciproca copertura: Ossia che ogni requisito sia coperto da almeno un caso d’uso e viceversa (in generale la relazione è molti-a-molti)





**LEZIONE 9 25/03**

ESERCITAZIONE

**LEZIONE 10 29/03**

La **stima dei costi** è essenziale in fase di pianificazione. Per un nuovo prodotto software da realizzare, serve a: Stimare il prezzo del prodotto, dimensionare il team di sviluppo del prodotto.

Per un prodotto esistente, serve a: Stimare il costo della manutenzione evolutiva, stimare il suo valore (per es. per decidere se effettuarne una manutenzione adattativa o risvilupparlo ex novo, a fronte di cambiamenti tecnologici).

Malgrado l'importanza della stima del costo, è difficile stimare un valore che si discosta di poco dal costo reale, a causa della difficoltà di misurare i due fattori che maggiormente la influenzano: produttività del personale coinvolto nella produzione del software (per esempio quante linee di codice per unità di tempo si riescono a produrre), complessità del software.

Una metrica di produttività ideale dovrebbe misurare la quantità di valore o di funzionalità prodotta (output) per unità di Effort (input). Effort: quantità di lavoro necessaria a completare un “Workpackage”. Normalmente espressa in giorni-uomo o mesi-uomo.

I **punti funzione** sono una metrica per la stima della “quantità di funzionalità” del software, si svincolano dalla tecnologia usata; sono applicabili soprattutto nelle prime fasi del ciclo di sviluppo del software (studio di fattibilità, analisi dei requisiti).

La **Function Point Analysis** (analisi a punti funzione) usa i punti funzione per quantificare le funzionalità del sistema software. Si usano il numero di informazioni in entrata, in uscita, e memorizzate nel sistema. I Function Point (FP) sono una metrica “orientata” ai requisiti dettati dal cliente poiché misurano le funzionalità che questi richiede. Non è richiesta la conoscenza del numero di linee di codice per effettuare le stime, sono sufficienti i requisiti funzionali. La function point analysis può essere adottata anche in fase di manutenzione (software esistente).

I Function Point sono conteggiati in due passi consecutivi:

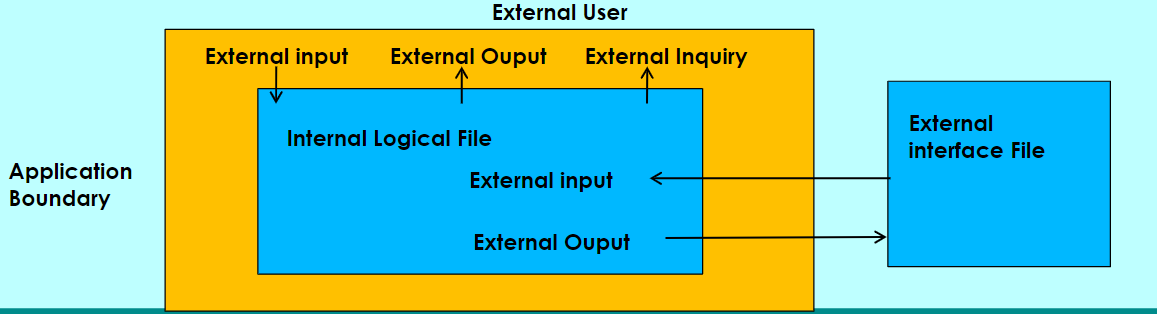
**1.Valutazione degli Unadjusted Function Point(UFP)**

Gli UFP prendono in considerazione gli aspetti funzionali del sistema e li classificano secondo degli indicatori.

Questa attività prevede: determinazione del tipo di conteggio, identificazione dell'ambito di conteggio e dei confini applicativi, identificazione delle funzioni dati, identificazione delle funzioni transazionali, conteggio UFP.

**2.ValutazionedeiFattoridiAggiustamento**

Il risultato che si ottiene con la valutazione degli UFP è raffinato con l'applicazione dei Fattori di Aggiustamento. Attraverso dei fattori correttivi si adegua il valore ottenuto dagli UFP al sistema che si sta progettando. Al termine di questa attività si ottiene il conteggio finale. Sia la tecnica per il conteggio degli UFP, sia i fattori di aggiustamento, sono frutto di considerazioni empiriche ovvero attraverso esperienze passate.

**Il confine applicativo** identifica le “linee di divisione” fra: l'utente e l'applicazione e che si vuole misurare e l'applicazione che si vuole misurare e eventuali altre applicazioni. Il confine è individuato tenendo conto della percezione che ha l'utente delle funzionalità offerta dal sistema. Quindi il confine è legato agli aspetti di business piuttosto che a considerazioni tecnologiche (difficilmente comprese dall'utente).

Per l'analisi dei FP si usano indici di due tipi: **Dati e Transazioni**.

**Dati**

**Internal Logical File (ILF**) - aggregati logici di dati generati, usati e gestiti internamente dal sistema identificabili dal cliente. Esempi tipici sono: tabelle di un database, file di configurazione, file di errore, password mantenuti dal sistema. I file temporanei non rientrano in questo tipo perché non visibili o accessibili dall'utente.

**External Interface File(EIF)** - aggregati logici di dati scambiati o condivisi con altre applicazioni identificabili dall'utente. Un EIF contato per un'applicazione deve essere un ILF per un'altra applicazione.

**Transazioni**

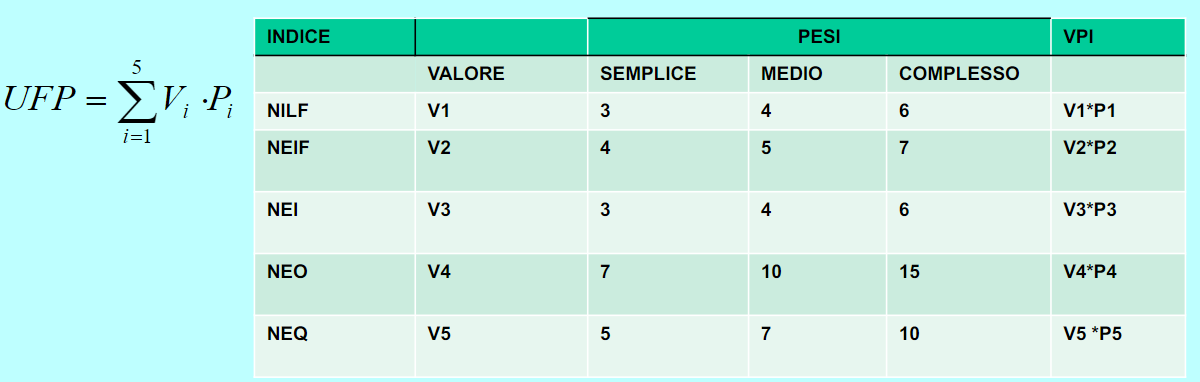
**External Input (EI)** – Informazioni distinte fornite dall'utente o da altre parti del sistema o da altre applicazioni, usate come dati di ingresso. Le informazioni di ingresso possono alterare un ILF oppure agire sul funzionamento del sistema. Esempi: dati transazionali (una vendita, un appuntamento inserito), messaggi di altre applicazioni, ingressi che portano informazioni di controllo (un sensore). Non sono da considerare EI schemi di menu che sono usati per navigare ma non agiscono su un ILF, richieste di input che sono parte di una query, meccanismi per il refresh della schermata video.

**ExternalOutput(EO)** – output distinti che il sistema restituisce all'utente come risultato delle proprie elaborazioni. L'intento dell'EO è di presentare dati all'utente attraverso una logica elaborativa diversa, o aggiuntiva del semplice recupero di dati (da un ILF) o di informazioni di controllo. Pertanto la logica elaborativa deve contenere una formula o un procedimento di calcolo e creare dei dati derivati. Esempi: report su terminale o stampante che richiedono l'uso di algoritmi o calcoli, trasferimenti di file o dati da altre applicazioni verso l'utente, documenti di varia natura con calcoli (fatture, bollette, assegni, ...). Non sono da considerare EO: messaggi di conferma o errore, report generati da un semplice query, più modi di invocare lo stesso processo di output.

**External Inquiry (EQ)** – interrogazioni in linea che producono una risposta immediata del sistema. L'intento primario è di presentare l'informazione all'utente attraverso il recupero da un ILF o EIF senza effettuare elaborazioni né modifiche ad un ILF o EIF. Esempi: dati prelevati da uno o più ILF/EIF e visualizzati, funzioni utente per la ricerca, file inviata ad altre applicazioni che non prevedono conteggio contengono formule. Non sono External Inquiry: schermate di menu per navigazione o selezione, messaggi di errore o conferma.

Il **conteggio degli Unadjusted Function Point richiede che si contino**: Numero degli ILF (NILF), Numero degli EIF (NEIF), Numero degli EI (NEI), Numero degli EO (NEO), Numero degli EQ (NEQ).

I valori precedentemente conteggiati sono pesati secondo la tabella, dove VPi sono i valori pesati per ciascun indice, ottenuti moltiplicando il campo Valore (cioè il numero fornito dal progettista relativo all'indice considerato) con uno dei tre pesi indicati, che variano a seconda che l'applicazione sia considerata semplice, media o complessa.



Il conteggio UFP produce un valore che può essere raffinato secondo 14 **fattori correttivi** che tengono in conto la “complessità” del sistema; il loro valore varia da 0 a 5: 0 ininfluente, 1 incidenza scarsa, 2 incidenza moderata, 3 incidenza media, 4 incidenza significativa, 5 incidenza essenziale. L'assegnazione del grado di incidenza avviene secondo l'esperienza di chi effettua il conteggio.

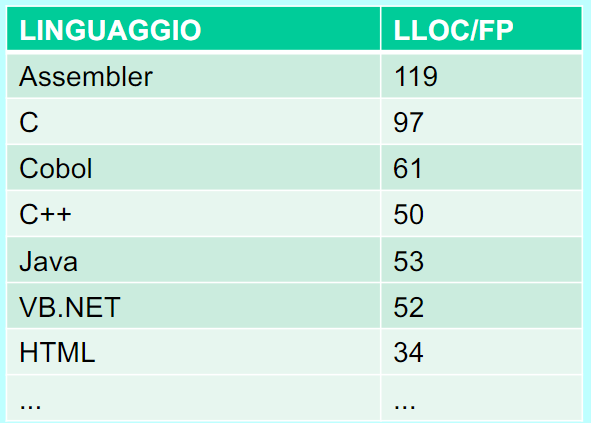


Il valore finale del conteggio FP è il prodotto fra il conteggio UFP e l'aliquota introdotta dai fattori correttivi.

Immagine che contiene testo

Descrizione generata automaticamente

Alcuni studi hanno stimato il numero di linee di codice logiche necessarie per codificare un punto funzione nei vari linguaggi. Il valore LLOC/FP è un valore medio.



I modelli per la stima dei costi del software mirano a valutare: La durata del processo di sviluppo, lo sforzo (effort) richiesto, il numero di persone necessarie (progettisti, programmatori, ...).

La formula generale per il calcolo dello sforzo nominale è:



**PM**, people month, **KLOC**, migliaia di linee codice, **c** e **L** sono costanti dipendenti dal tipo di modello.

**La formula è calibrata al processo software** mediante dei fattori correttivi, classificabili per:

**Prodotto**, requisiti di affidabilità o di complessità dell'applicazione.

**Computer**, esistenza di vincoli sulle memoria occupata e tempi di esecuzione.

**Personale**, l'esperienza delle persone nell'area applicativa e sulle tecnlogie usate.

**Progetto**, disponibilità di strumenti di sviluppo sofisticati.

La stima dei costi si articola in generale in tre passi:

**Stima della dimensione del software.**

**Calibrazione della stima**

**Applicazione degli strumenti del modello alla stima per determinare le grandezze di interesse** (sforzo totale, durata, distribuzione attività, etc.).

Il **COnstructive Cost Model (COCOMO)** è un modello per la stima dei costi, evoluto, successivamente, in COCOMOII. Il COCOMO assume che il ciclo di sviluppo sia a cascata. Originariamente il COCOMO prevedeva tre diversi livelli di dettaglio:

**Basic**, utile per una veloce stima dei costi, ma approssimato

**Intermediate**, aggiunge alla stima effettuata dal basic dei fattori correttivi (Cost Drivers)

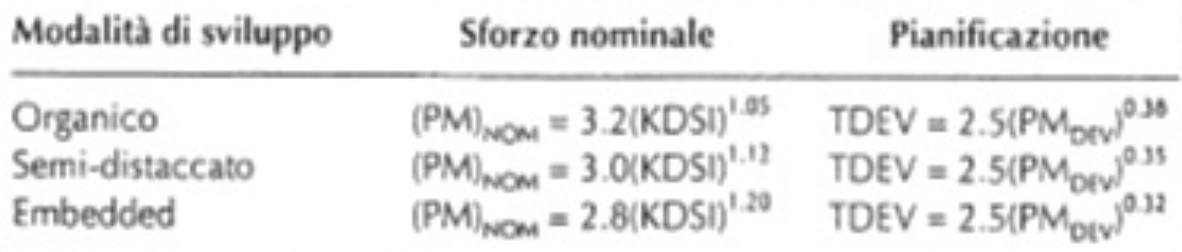
**Detail**, stima i costi alla stessa maniera dell'intermediate ma si sofferma su ogni fase del ciclo di sviluppo del software (analisi, progettazione, ...).

Nel modello COCOMO Intermediate i tre passi della stima dei costi si specializzano come segue:

**Scelta della modalità di sviluppo**. Il COCOMO definisce tre modalità di sviluppo (**organico, semi-distaccato, embedded**).

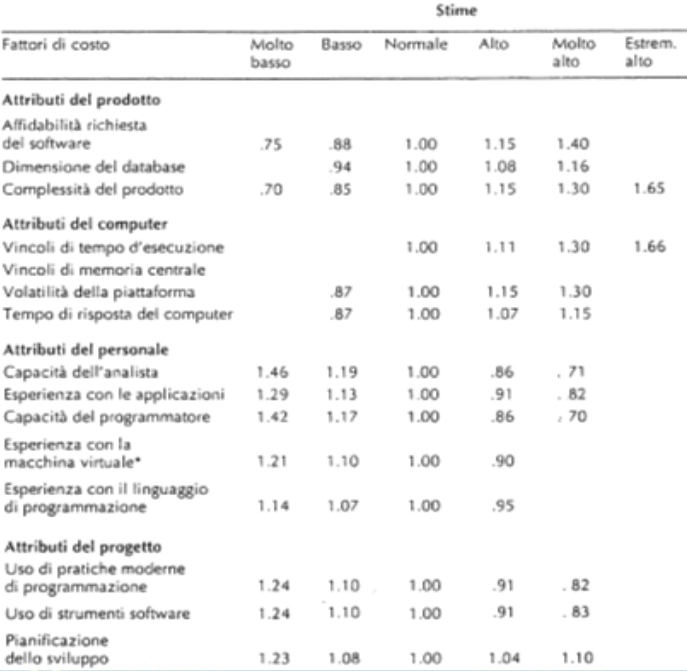


**Determinazione dello sforzo nominale**, sulla base della modalità. Ad ogni modalità di sviluppo corrisponde nel COCOMO Intermediate una formula per il calcolo dei costi. Lo sforzo nominale è determinato in base ai valori della tabella seguente.

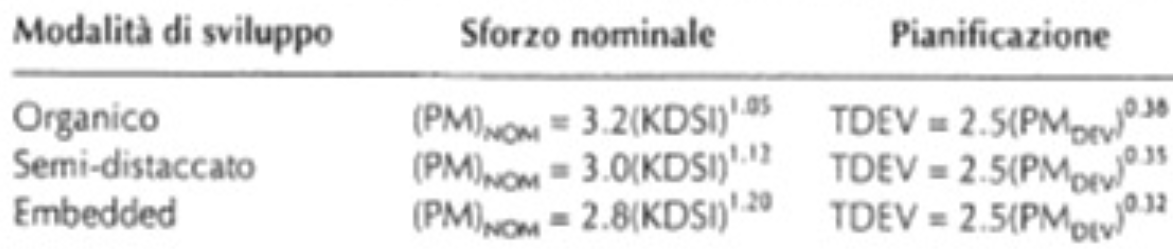


TDEV : time to development.

**Correzione della stima**. Il valore PM ottenuto viene corretto per tener conto di fattori quali l'esperienza dei programmatori, e l'affidabilità richiesta. I fattori correttivi sono riportati in tabella; per ognuno si sceglie un valore e si moltiplicano fra loro. Il prodotto ottenuto è moltiplicato per il valore PM calcolato in precedenza.



**Applicazione degli strumenti del modello alla stima per determinare le grandezze di interesse**. Ad es., il calcolo del tempo di sviluppo (TDEV) si ottiene in funzione della modalità secondo le formule riportate nella terza colonna della tabella sottostante, in cui PMDEV è calcolato a partire da PMNOM applicando i fattori correttivi ricavati.



**LEZIONE 11 30/03**

Nella progettazione di un sistema software è opportuno basarsi sul principio della **modularità**, perché esso risulti più semplice da comprendere e manipolare. I moduli sono componenti (parti) del sistema che realizzano un’astrazione. I tipi di astrazione più diffusi sono:

**ASTRAZIONE SUL CONTROLLO** Astrazione di una funzionalità dai dettagli dell’implementazione. È ben supportata dai linguaggi di programmazione tradizionali tramite il concetto di sottoprogramma.

**ASTRAZIONE SUI DATI** Consiste nell'astrarre le **entità(oggetti)** costituenti il sistema, descritte in termini di una struttura dati e delle operazioni possibili su di essa; Può essere realizzata con un uso opportuno delle tecniche di **programmazione modulare** nei linguaggi tradizionali; E’ alla base della **Modellazione a oggetti**. E’ supportata da appositi costrutti nei linguaggi di **programmazione ad oggetti.**

**Oggetti (astrazione sui dati):**

•Strutture dati + operazioni

•Rispetto alle funzioni/procedure (e librerie di funzioni), hanno una struttura dati permanente, visibile solo alle funzioni interne

•La struttura dati fornisce loro uno **stato**.

**Tipi di dati astratti**

•Insieme alle operazioni possibili sui dati, esporta un **tipo**

•Gli esemplari del tipo sono **oggetti**

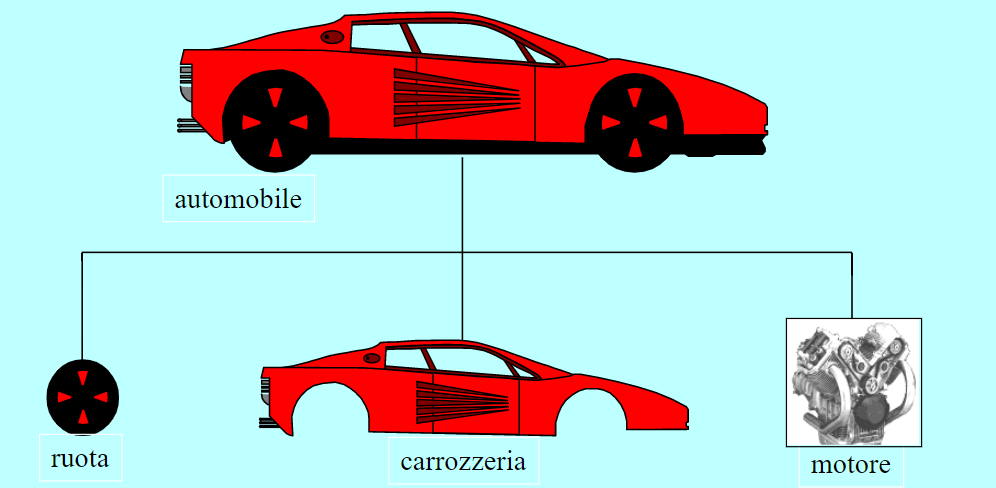
l concetto di **tipo** di dato in un linguaggio di programmazione tradizionale è quello di insieme dei valori che può assumere un dato (una variabile). Il **tipo di dati astratto(TDA)** estende questa definizione, includendo anche l'insieme di **tutte e sole le operazioni** possibili su dati di quel tipo. La struttura dati “concreta” è **incapsulata** nelle operazioni su di essa definite. Non è possibile accedere alla struttura dati incapsulata (né in lettura né in scrittura) se non attraverso le operazioni definite su di essa.

**Interfaccia**: specifica del TDA, descrive la parte direttamente accessibile dall’utilizzatore. **Realizzazione**: implementazione del TDA.

Il **cliente o utilizzatore** fa uso del TDA per realizzare procedure di un’applicazione, o per costruire TDA più complessi. Il **produttore** realizza le astrazioni e le funzionalità previste per il dato.

Una modifica nella sola realizzazione del TDA non influenza i moduli che ne fanno uso (in quanto non cambia l’interfaccia).

Un oggetto complesso può essere composto di oggetti più semplici detti componenti.



**Metodologia discendente o “top -down’‘:** basata sulla decomposizione funzionale nella definizione del sistema software, individuazione delle funzionalità e raffinamenti successivi.

**Metodologia ascendente o “bottom-up’**‘: basata su individuazione di entità del sistema (classi e/o oggetti), delle loro proprietà e delle interrelazioni tra di esse. Il sistema viene costruito assemblando componenti con un approccio “dal basso verso l’alto”.

La modellazione a oggetti è di tipo bottom-up. Si individuano le **classi di oggetti**(entità del mondo reale o concettuale) che caratterizzano il dominio applicativo, le diverse classi vengono poi modellate, progettate e implementate, ogni classe è descritta da **un’interfaccia** che specifica il comportamento degli oggetti della classe. L’applicazione si costruisce assemblando oggetti e individuando le modalità con cui questi devono collaborare per realizzare le diverse funzionalità dell’applicazione. I principi della modellazione ad oggetti non sono vincolati ad una fase, ma riguardano l’analisi, la progettazione e l’implementazione.

A livello di implementazione, distinguiamo tra:

**Programmazione con oggetti** con riferimento a tecniche di programmazione basate sul concetto di oggetto (dati + operazioni)

**Programmazione basata sugli oggetti (object-based programming**) con riferimento alle tecniche di programmazione basate sui concetti di:

•Tipo di dati astratto o Classe (tipo)

•Oggetto (istanza di un tipo)

**Programmazione orientata agli oggetti (object-oriented programming, OOP)** con riferimento alle tecniche basate sui concetti di:

•Classe ed Oggetto

•Ereditarietà, Polimorfismo

In linguaggi procedurali (ad es., in C) è possibile adottare tecniche di programmazione con oggetti o basate sugli oggetti, adoperando opportune discipline di programmazione, aderendo cioè ad un insieme di regole, il cui uso però non può essere forzato né verificato dal compilatore.

Un linguaggio di programmazione ad oggetti offre costrutti espliciti per la definizione di entità (oggetti) che incapsulano una struttura dati nelle operazioni possibili su di essa. I linguaggi ad oggetti C++ e Java, consentono di definire tipi astratti (con il costrutto class), e quindi istanze (oggetti) di tipi di dato astratto.

Esistono linguaggi ad oggetti:

**Non tipizzati.** Es.: Smalltalk. È possibile definire oggetti senza dichiarare il loro tipo. In tali linguaggi, gli oggetti sono entità che incapsulano una struttura dati nelle operazioni possibili su di essa.

**Tipizzati.** Es.: C++, Java. È possibile definire tipi di dati astratti e istanziarli. Gli oggetti devono appartenere ad un tipo (astratto). In tali linguaggi: una **classe** è una implementazione di un tipo astratto; un **oggetto** è una istanza di una classe.

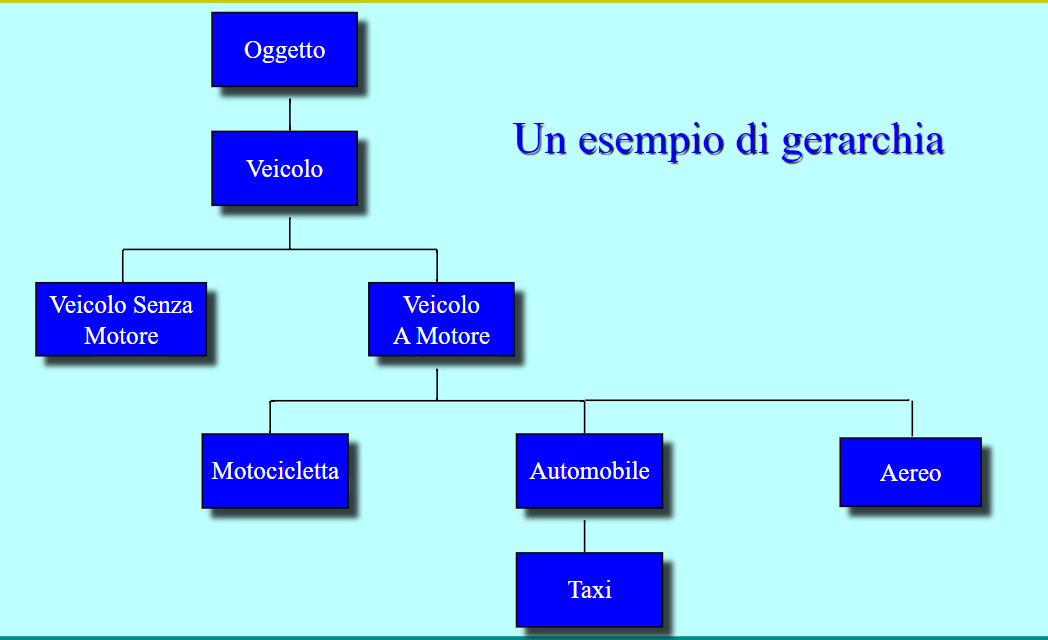
C++ è un linguaggio di programmazione **general-purpose** che supporta: la programmazione procedurale(è un C “migliore”), la programmazione orientata agli oggetti, la programmazione generica. C++ è quindi un **linguaggio ibrido**, nel senso che supporta più paradigmi di programmazione.

In **fase di analisi (OOA)**, le classi e le relazioni tra esse rappresentano una vista concettuale delle entità nel dominio del problema. Catturano i concetti principali del dominio, svincolandoli da come essi saranno rappresentati dal software. L’attenzione dovrà essere rivolta alla comprensione di **“cosa”** il sistema software dovrà fare e non “come” lo fa. In **fase di progetto (OOD)**, le classi e le relazioni tra esse formano l’architettura del sistema (ancora slegata dalla implementazione in uno specifico linguaggio). **In fase di implementazione (OOP)**, le classi e le relazioni indicano la reale struttura del software.

La classe è un modulo software con le seguenti caratteristiche: È dotata di **un’interfaccia** (specifica) e di un **corpo** (implementazione). La struttura dati “concreta” di un oggetto della classe, e gli algoritmi che ne realizzano le operazioni, sono tenuti nascosti all’interno del modulo che implementa la classe. Lo **stato di un oggetto** evolve unicamente in relazione alle operazioni su di esso invocate. Le operazioni sono utilizzabili con modalità che prescindono completamente dagli aspetti implementativi; in tal modo è possibile modificare gli algoritmi utilizzati senza modificare l’interfaccia.

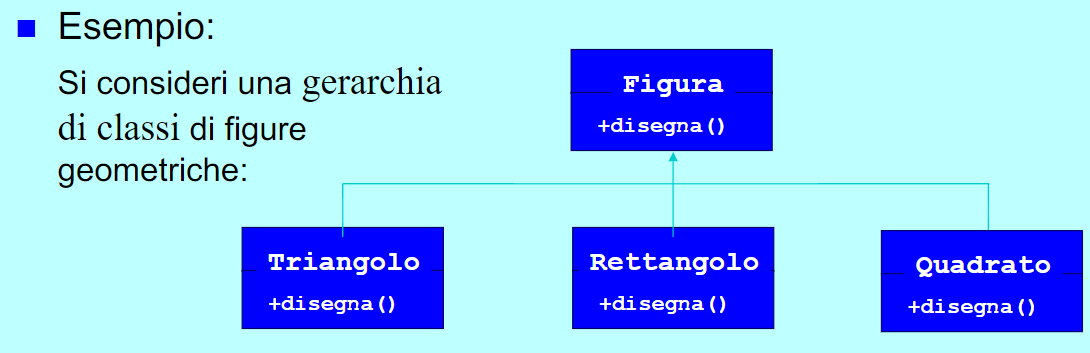
Una classe in C++ possiede, in generale, una sezione **pubblica** ed una **privata**. La sezione pubblica contiene tipicamente le operazioni (dette anche metodi) consentite ad un utilizzatore della classe. Esse sono tutte e sole le operazioni che un utente può eseguire, in maniera esplicita od implicita, sugli oggetti. La sezione privata comprende le strutture dati e le operazioni che si vogliono rendere inaccessibili dall'esterno.

**L’ereditarietà** consente di definire nuove classi per specializzazione o estensione di classi preesistenti, in modo incrementale. L’ereditarietà è di fondamentale importanza nella modellazione ad oggetti, induce una strutturazione gerarchica nel sistema software in costruzione. Consente di realizzare relazioni tra classi di tipo **generalizzazione-specializzazione**, in cui: una classe, detta **base**, realizza un comportamento generale comune ad un insieme di entità, le classi **derivate** **(sottoclassi)** realizzano comportamenti specializzati rispetto a quelli della classe base .



Oltre a poter descrivere un sistema secondo un modello gerarchico, esiste un altro motivo, di ordine pratico, per cui conviene usare l'ereditarietà; esso è legato al concetto di **riuso del software**. In alcuni casi si ha a disposizione una classe che non corrisponde esattamente alle proprie esigenze. Anziché scartare del tutto il codice esistente e riscriverlo, si può seguire con l'ereditarietà un approccio diverso, costruendo una nuova classe che eredita il comportamento di quella esistente, salvo che per i cambiamenti che si ritiene necessario apportare.

In definitiva, l’ereditarietà favorisce **riuso, manutenibilità, incrementalità** e non è necessario conoscere in dettaglio il funzionamento del codice da riutilizzare, ma è sufficiente modificare (mediante aggiunta o specializzazione) la parte di interesse.

Il **polimorfismo** consente di invocare operazioni su un oggetto, pur non essendo noto a tempo di compilazione il tipo (la classe) corrente dell’oggetto stesso. Per polimorfismo dinamico (o semplicemente polimorfismo) si intende la proprietà di una entità di assumere forme diverse nel tempo. Una entità è polimorfa se può fare riferimento, nel tempo, a classi diverse.

Sia ad es. A un vettore di N oggetti della classe Figura, composto di oggetti delle sottoclassi Triangolo, Rettangolo, Quadrato: "Figura A[N] (ad es.: A[0] è un quadrato, A[1] un triangolo, A[2] un rettangolo, etc.). Si consideri una funzione Disegna\_Figure(), che contiene il seguente ciclo di istruzioni:

for i = 1 to N do

A[i].disegna()

L'esecuzione del ciclo richiede che sia possibile **determinare dinamicamente** (a tempo di esecuzione) l'implementazione della operazione disegna() da eseguire, in funzione del tipo corrente dell'oggetto A[i]. L’istruzione A[i].disegna() non ha bisogno di essere modificato in conseguenza dell'aggiunta di una nuova sottoclasse di Figura (ad es.: Cerchio), anche se tale sottoclasse non era stata neppure prevista all'atto della stesura della funzione Disegna\_Figure(). Il polimorfismo supporta dunque la proprietà di **estensibilità** di un sistema, nel senso che minimizza la quantità di codice da modificare quando si estende il sistema, cioè si introducono nuove classi e nuove funzionalità. Un meccanismo con cui viene realizzato il polimorfismo è quello del **binding dinamico**. Il binding dinamico (o **late binding**) consiste nel determinare a tempo d'esecuzione, anziché a tempo di compilazione, il corpo del metodo da invocare su un dato oggetto.

Vantaggi dell’orientamento ad oggetti:

**Modularità**: le classi sono i moduli del sistema software;

**Coesione dei moduli**: una classe è un componente software ben coeso in quanto rappresentazione di una unica entità;

**Disaccoppiamento dei moduli**: gli oggetti hanno un alto grado di disaccoppiamento in quanto i metodi operano sulla struttura dati interna ad un oggetto;

**Information hiding**: sia le strutture dati che gli algoritmi possono essere nascosti alla visibilità dall'esterno di un oggetto;

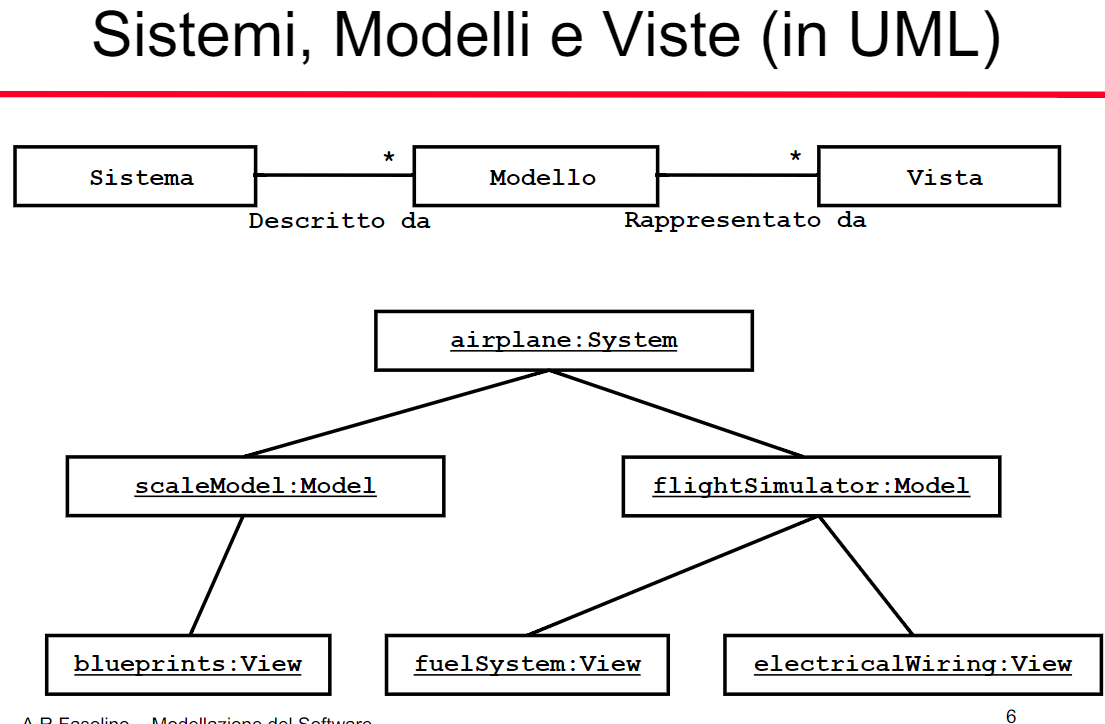
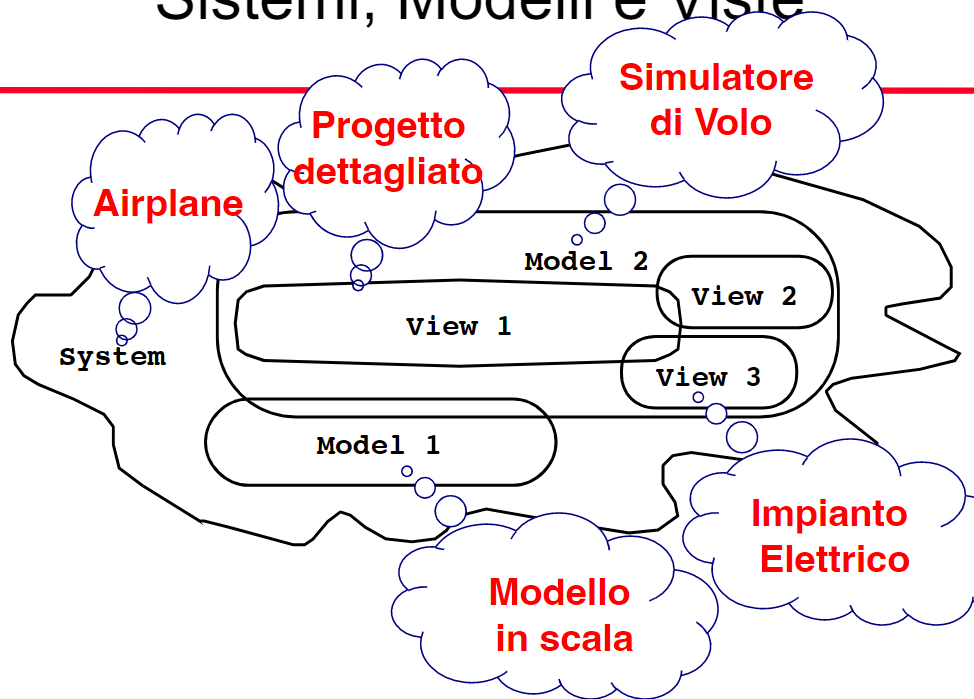
**Riuso**: l'ereditarietà consente di riusare la definizione di una classe nel definire nuove (sotto)classi; inoltre è possibile costruire librerie di classi raggruppate per tipologia di applicazioni;

**Estensibilità**: il polimorfismo agevola l'aggiunta di nuove funzionalità, minimizzando le modifiche necessarie al sistema esistente quando si vuole estenderlo.

Se dunque si effettuano l’analisi O-O, la progettazione O-O e poi la codifica O-O in un linguaggio di programmazione a oggetti come C++ o Java, si ha il grande vantaggio di adoperare lo stesso modello (quello appunto dell’object orientation, che consiste dei concetti di classe, oggetto, ereditarietà e polimorfismo) in tutte e tre queste fasi del ciclo di vita del software.

**LEZIONE 12 01/04**

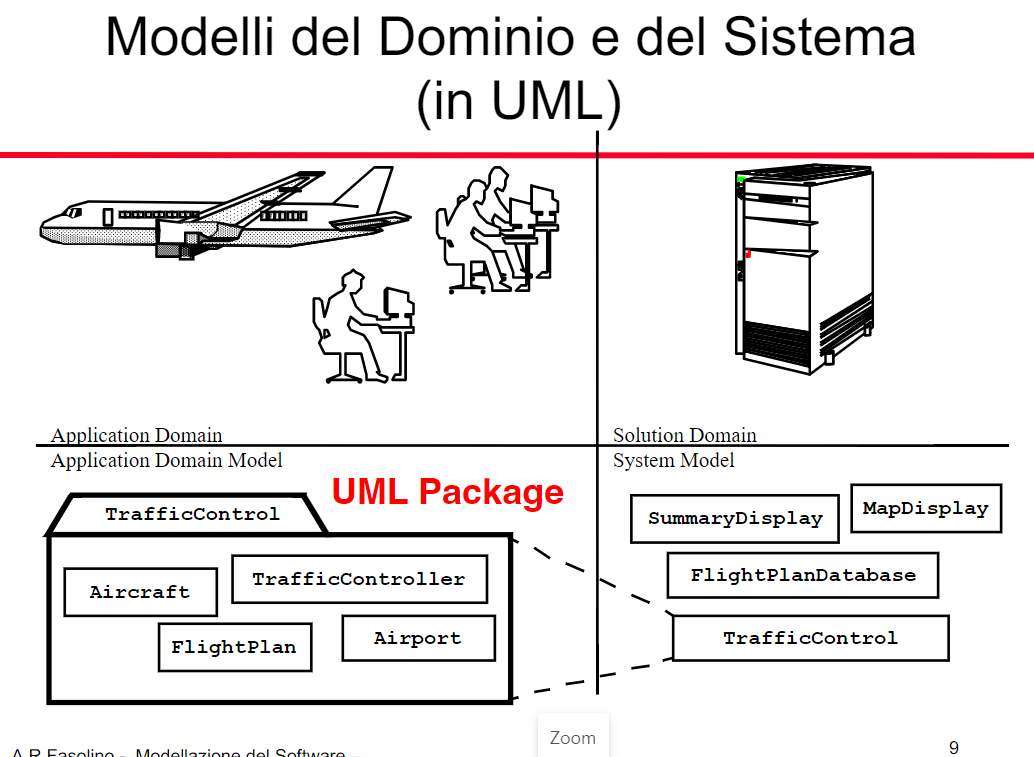
Un **modello** è una astrazione che descrive un sistema o un sotto-sistema, ogni sistema o sotto-sistema può essere descritta attraverso diversi modelli ognuno dei quali potrebbe corrispondere ad una vista .Una **vista** è una descrizione di aspetti specifici di un modello. Una **Notazione** è un insieme di regole grafiche o testuali per rappresentare le viste.



l Software è già una astrazione: perchè modellare il software?

I moderni sistemi **Software** tendono a crescere e a diventare sempre più complessi. Esempio: I moderni sistemi operativi raggiungono dimensioni di milioni di linee di codice. É impensabile gestire tali sistemi operando direttamente sul codice! Il codice è spesso incomprensibile(soprattutto per chi non ha partecipato al suo sviluppo). C’è bisogno di **forme di rappresentazione** più semplici per sistemi complessi! La Modellazione è un modo per gestire la complessità.

I Modelli possono essere usati per rappresentare il **Dominio Applicativo**: Ossia l’ambiente in cui il sistema dovrà operare. La modellazione del dominio applicativo è oggetto dell’Analisi dei Requisiti. Oppure per rappresentare il **Dominio della soluzione**: Ossia le tecniche e tecnologie disponibili per costruire il sistema. Tale modellazione è oggetto dell’attività di Progettazione del sistema.



Diversi tipi di modelli possono descrivere il dominio del problema. Tali modelli sono detti modelli di dominio oppure modelli concettuali o, talvolta, modelli di business. Attenzione: non bisogna confondere i modelli di dominio, che descrivono il problema, con i modelli di progetto, che descrivono la soluzione. La **Modellazione del dominio del problema** include: **Modelli statici** (o strutturali), che descrivono gli elementi del dominio del problema e le relazioni tra loro (Vista statica). **Modelli dinamici**, che descrivono i comportamenti riconoscibili nel dominio del problema (Vista dinamica).

Per descrivere i modelli si utilizzano linguaggi di modellazione. Negli ultimi anni il **linguaggio UML** si sta affermando come linguaggio unificato che possa essere utilizzato in tutte le attività di modellazione, nonchè in molte altre attività del ciclo di vita del software.

UML (Unified Modelling Language) nasce come linguaggio grafico standard per modellare software object oriented. Una specifica UML in pratica definisce un **metamodello** che indica secondo quali regole sia possibile costruire modelli UML. L’obiettivo di UML è di fornire un supporto al processo di sviluppo software.

UML è dotato sia di regole **prescrittive** che di regole **descrittive**. Le regole prescrittive sono regole stabilite da organismi standardizzanti che caratterizzano precisamente lessico, sintassi e semantica. Le regole descrittive, invece, sono regole che possono essere meglio comprese guardando come l’UML viene usato nella pratica. UML ha come scopo principale la descrizione efficace di situazioni reali, cosicchè le regole descrittive sono al momento predominanti.

**Regola delle informazioni soppresse**: L’assenza di qualche informazione in un diagramma UML non significa, in generale, che tale informazione non esista o sia nulla, ma semplicemente che si tratta di un aspetto del problema non ancora trattato nella fase in cui è stato tracciato il diagramma (o che non si voglia palesare in tale diagramma per poi inserirlo in diagrammi ulteriori). Se omettiamo un informazione nel nostro diagramma non possiamo dare per scontato che venga interpretata in un certo modo.

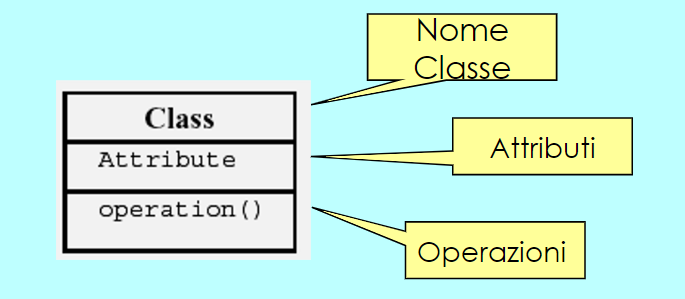
UML 2 possiede 13 differenti tipi di diagrammi, appartenenti a tre categorie: **Diagrammi Comportamentali**, quali use-case diagrams, activity diagrams, state machine diagrams. **Diagrammi di Interazione**, per modellare interazioni fra entità del sistema, quali sequence diagrams e communication diagrams. **Diagrammi Strutturali**, che modellano l’organizzazione del sistema, quali class diagrams, package diagrams, e deployment diagrams. Tali diagrammi rappresentano i deliverables di diverse fasi del ciclo di vita del software, tra cui attività di analisi dei requisiti e attività di progettazione, sia di alto che di basso livello.

Un **diagramma delle classi (Class Diagram, CD)** rappresenta le classi e le relazioni tra esse. Si tratta di una vista **statica** che può essere realizzata sia in fase di analisi e specifica dei requisiti, sia in fase di progettazione di alto livello, sia di progettazione di dettaglio (o anche di implementazione). **In fase di analisi**, un CD è una vista concettuale delle entità nel dominio del problema. Cattura i concetti principali del dominio, a prescindere: da come essi saranno rappresentati in archivi o comunque in un sistema di elaborazione, dalle funzionalità che opereranno su di essi(->software). **In fase di progetto,** un diagramma delle classi descrive le entità del sistema e le relazioni tra esse(ancora prescindendo dalla implementazione); è una delle viste statiche del sistema progettato. **In fase di implementazione**: le classi e le relazioni indicano la reale struttura del software.

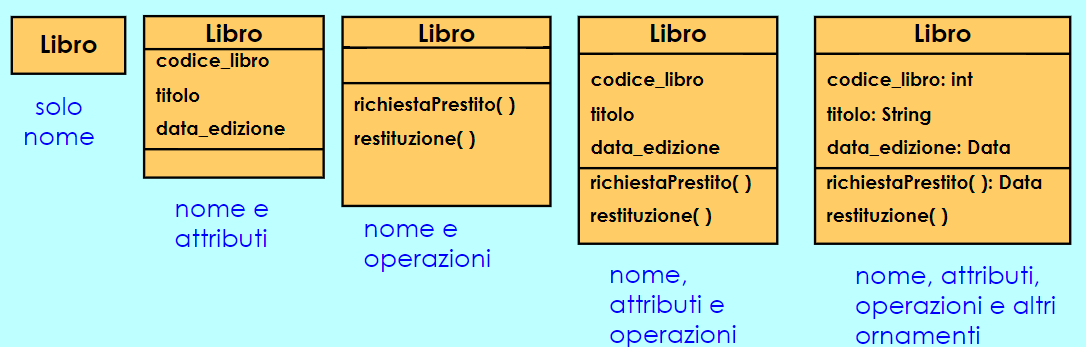
Nei CD di progettazione e implementazione trovano posto anche classi e relazioni che riguardano la realizzazione del sistema software, e non hanno diretta corrispondenza con classi e relazioni del CD di analisi.

Un CD mostra una vista statica del modello, in particolare gli elementi che esistono, le loro proprietà/attributi e responsabilità/operazioni, e le relazioni con le altre entità (non mostra informazioni temporali).

**Classe**: è il descrittore di un insieme di oggetti con le stesse proprietà (attributi), comportamento (operazioni) e relazioni con altri oggetti.



Una classe è rappresentata da un rettangolo con tre diverse sottosezioni: **nome, attributi, operazioni**. Il nome è solitamente scritto in “Upper Camel Case”(inizia con una maiuscola e se costituito da più parole, non sono previsti spazi, ma ogni parola inizia con la maiuscola). Il nome è obbligatorio, le altre sottosezioni possono essere omesse. Gli attributi sono scritti in «lowerCamelCase»



**Attributi:**



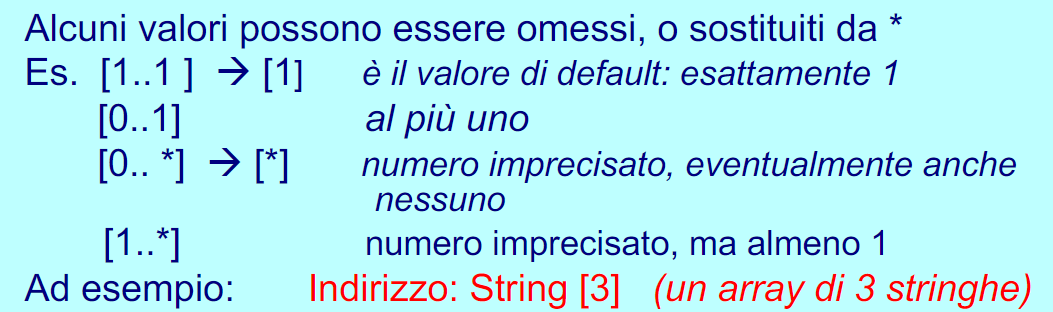
Sono ammessi tre livelli di visibilità degli attributi:

**+ Livello pubblico**: L'utilizzo è esteso a tutte le classi

**# Livello protetto**: L'utilizzo è consentito soltanto alle classi che derivano dalla classe originale

**-Livello privato**: Soltanto la classe originale può utilizzare gli attributi e le operazioni definite come tali.

Il **tipo** dell’attributo può essere un tipo primitivo (int, double, char, etc...), oppure il nome di una classe definita nello stesso diagramma (in tal caso l’ attributo andrebbe indicato con un associazione …). l tipo può essere seguito dalla **molteplicità**, che indica il quantitativo degli attributi ( lowerbound... upperbound). Il numero minimo e massimo possono essere racchiusi tra parentesi quadre quando il tipo è semplice: [ ]. Alcuni valori possono essere omessi, o sostituiti da \*



**valoreIniziale** è il valore assegnato all’attributo per default. **{proprietà}** indica caratteristiche aggiuntive dell’attributo (ad esempio la sola lettura).

Sono le operazioni invocabili sugli oggetti istanza della classe. La sintassi completa per la firma(signature) di una operazione è:



La **visibilità** e il **nome** seguono regole analoghe a quelle per gli attributi.

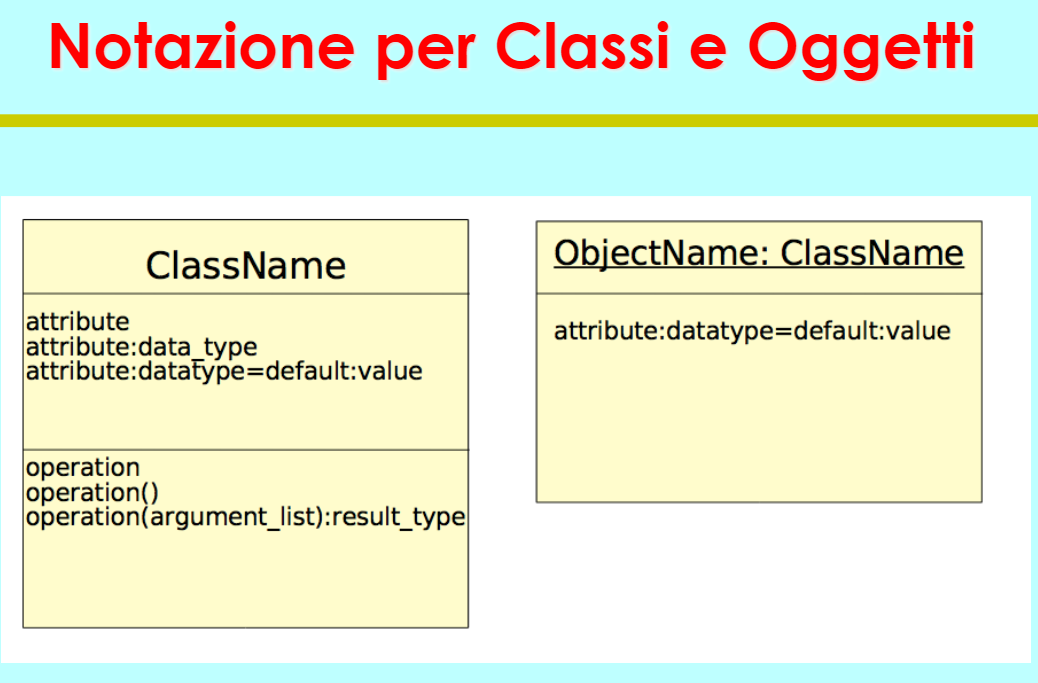
**Lista parametri** contiene nome e tipo dei parametri della funzione, secondo la forma:

direzione nome parametro: tipo= valore-di-default

**direzione**: input (in), output (out) o entrambi(inout). Il valore di default è in.

**nome**, **tipo** **e valore di default** sono analoghi a quellidegliattributi

**Tipo-restituito** è il tipo del valore di ritorno



Le classi (o gli oggetti) in un diagramma delle classi (degli oggetti) non sono in generale isolate, ma in relazione con altre entità (oggetti) con legami di varia natura, detti “**relazioni**”.

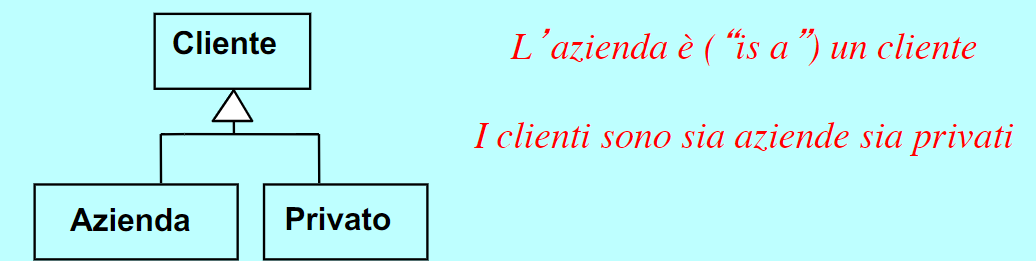
Le principali relazioni tra le classi sono:

•La relazione di **generalizzazione-specializzazione**

•La relazione di **associazione**

•La relazione di **contenimento**

La relazione di generalizzazione-specializzazione (o gen-spec), tipica dell’orientamento agli oggetti (object-orientation), corrisponde al legame tipo-sottotipo. Essa **indica legami tra classi del genere “è un”(is-a).** La **superclasse** raccoglie caratteristiche e comportamenti comuni agli elementi delle **sottoclassi**.

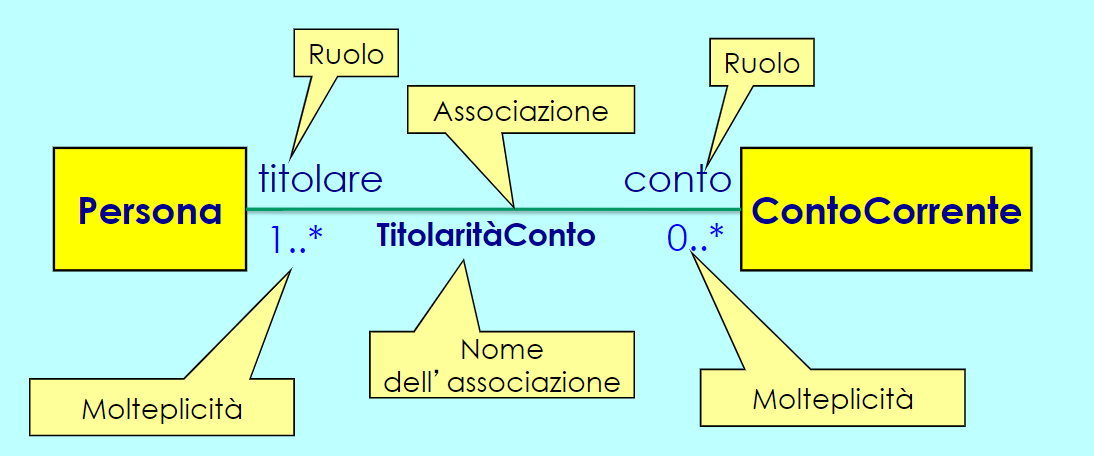


Le relazioni gen-spec descrivono **gerarchie di ereditarietà**. La classe derivata (sottoclasse) ereditale caratteristiche della classe base (superclasse). Gli attributi e le operazioni della classe base apparterranno anche alla classe derivata. La classe derivata può:-aggiungere altri attributi e operazioni, specializzare operazioni della classe base.

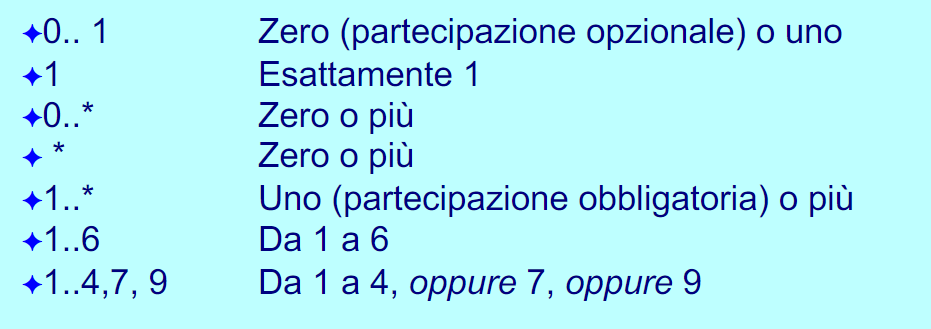
**Principio di Sostituibilità**: Ogni elemento della classe Azienda (o Privato) è anche un elemento della classe Cliente. Ne consegue che il software potrà riferirsi al tipo Cliente, senza dover distinguere se esso è un’Azienda o un Privato.

**L'associazione** tra due (o più) classi esprime un legame semantico tra le istanze (oggetti) delle classi.

Una associazione è caratterizzata da: un **nome** che esprime il legame semantico tra le classi associate, un eventuale **ruolo** giocato da ciascuna delle parti associate, la **molteplicità** dell'associazione e la **navigabilità** dell’associazione.



La molteplicità vincola il numero di oggetti di una classe che possono partecipare ad una relazione in un dato istante. Sono della forma: minimo … massimo

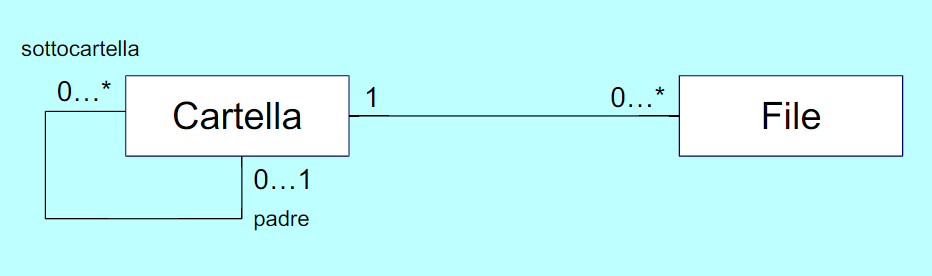


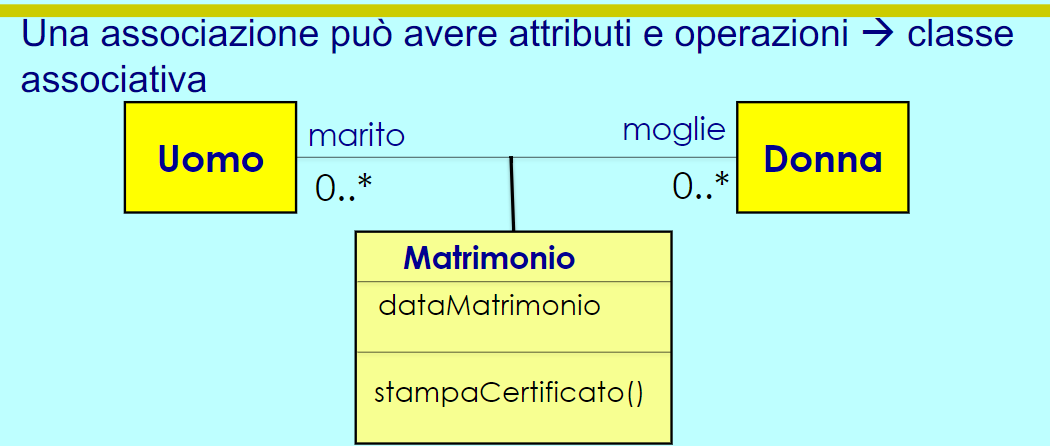
**LEZIONE 13 05/04**

**Navigabilità dell’associazione**

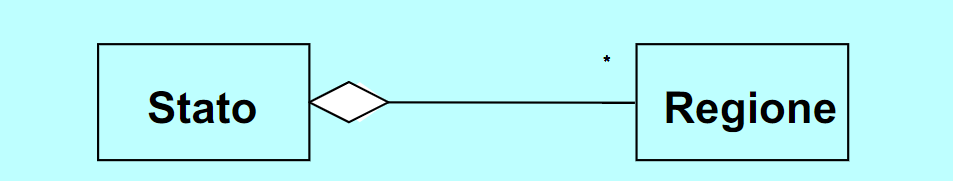
Da un punto di vista concettuale, una associazione non ha un verso di percorrenza (una direzionalità), se A è legato a B, B è legato ad A. Nonostante ciò, UML offre la possibilità di specificare una direzionalità. Il **verso di navigazione** di una associazione è un’informazione utile soprattutto in fase di progetto di dettaglio. Di solito, il verso di navigazione rappresenta una scelta di progetto, per cui non è presente nei diagrammi concettuali (di analisi).

L'associazione può essere **riflessiva** , oggetti della classe possono avere collegamenti con oggetti della stessa classe.



A volte le proprietà di un’associazione sono proprie della relazione e non delle classi coinvolte, pertanto può essere opportuno definire una **classe associativa** . **Proprietà**: è utilizzata per modellare proprietà delle associazioni, ogni attributo di una classe associativa contiene un valore per ogni istanza di associazione, è utilizzata, in genere, nelle associazioni molti a molti.

La classe associativa connette due classi e definisce un insieme di caratteristiche proprie della associazione stessa. Le istanze dell'associazione sono **collegamenti tra oggetti**, dotati di attributi e operazioni. Per reificare (rendere reale) una classe associativa sarà opportuno (tipicamente, in fase di progettazione) trasformarla in una vera classe.

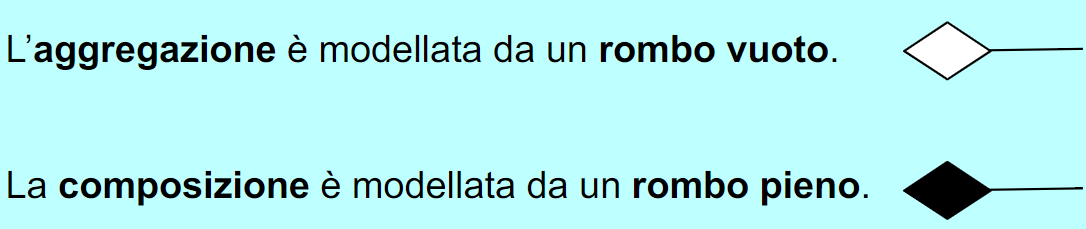
Un altro tipo di legame tra classi è la **relazione di contenimento**, che rappresenta il legame tra un insieme(il “tutto”) e le sue parti(**legame tutto-parti**), ovvero tra un contenitore e il contenuto(**legame contenitore-contenuto**).

Il legame è modellato in UML con un segmento che unisce due classi, con un rombo dal lato del contenitore. Il rombo è vuoto o pieno, secondo il tipo di contenimento.

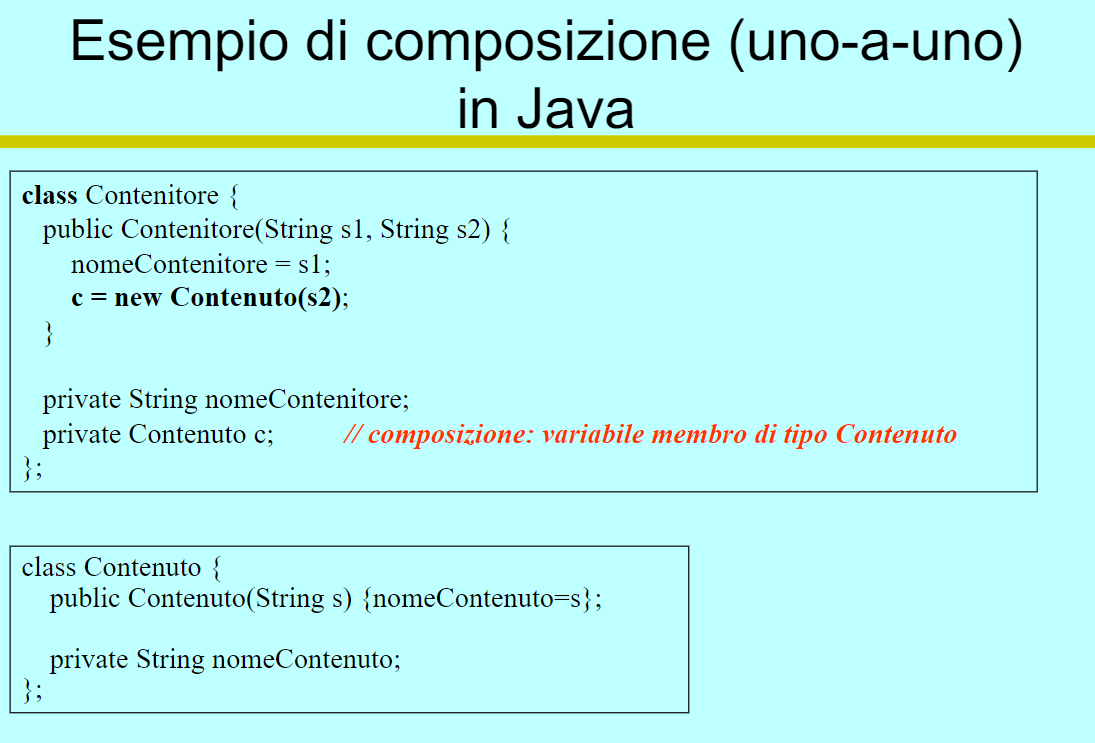
Il contenimento può infatti essere

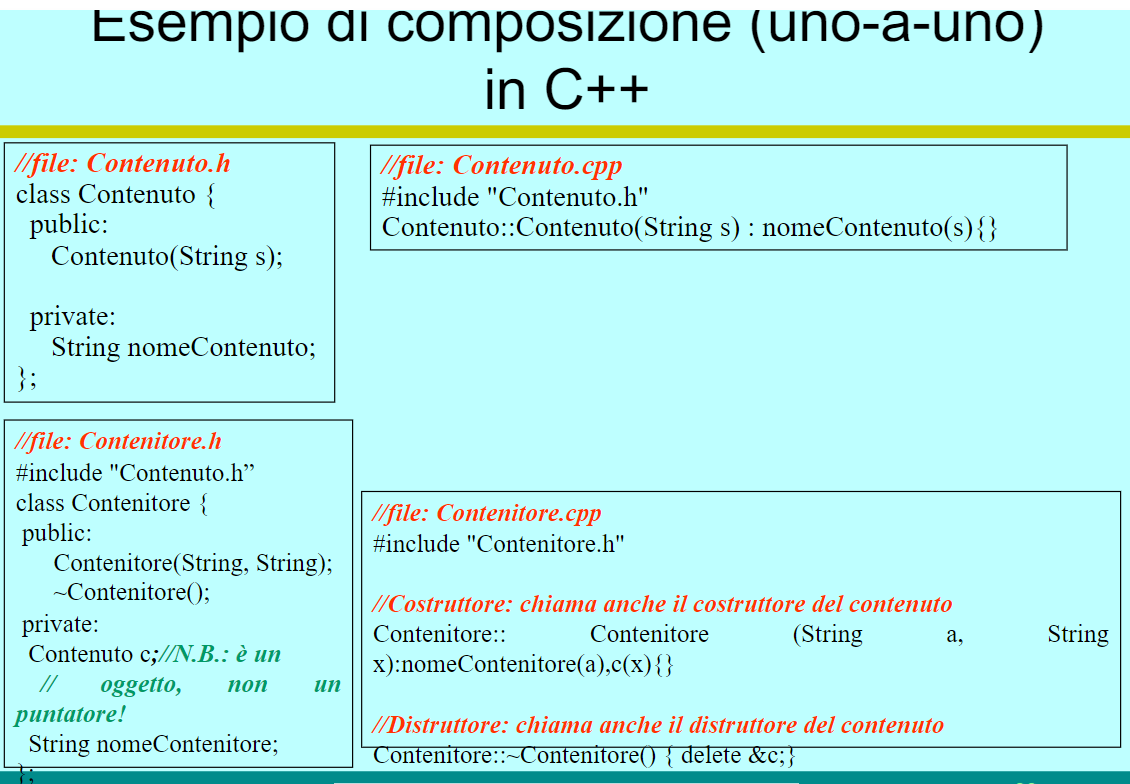
**debole** (o lasco), detto anche **aggregazione**: si ha quando la parte mantiene la sua identità quando entra a far parte del tutto(Es.: auto –motore). La relazione di aggregazione indica l'indipendenza del ciclo di vita dell'oggetto **contenuto** da quello dell'oggetto contenitore. L'oggetto contenuto può quindi esistere anche indipendentemente dal contenitore. L'oggetto contenitore NON è necessariamente responsabile della costruzione e distruzione dell'oggetto contenuto. L’aggregazione si traduce in un linguaggio ad oggetti: Inserendo nella classe contenitore una variabile membro di tipo puntatore (o un riferimento) ad un oggetto della classe contenuto, implementando un costruttore della classe contenitore che riceva in ingresso un puntatore (un riferimento) ad un oggetto della classe contenuto.

**stretto**, detto anche **composizione**: se la parte perde la sua identità quando entra a far parte del tutto(Es.: pane –farina). La relazione di composizione indica che l'oggetto **contenuto** non ha una vita propria ma esiste in quanto parte dell'oggetto contenitore. Se l’aggregato viene distrutto, anche le sue parti saranno distrutte (le parti non esistono senza il tutto). L'oggetto **contenitore** è responsabile della costruzione e distruzione dell'oggetto contenuto. La composizione si traduce in un linguaggio ad oggetti: Aggiungendo alla classe contenitore una variabile membro del tipo della classe contenuto, implementando un costruttore del contenitore in modo da richiamare il costruttore del contenuto.

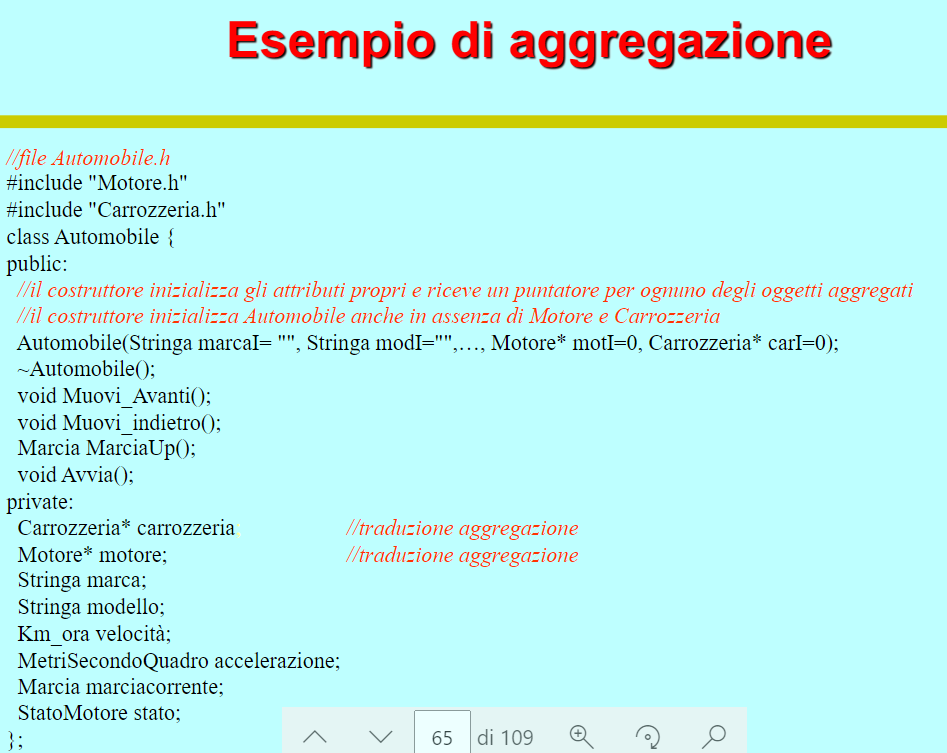


In particolare, la composizione si può realizzare in C++ o in Java come segue: in C++: il contenitore C ha come variabile membro privata un **oggetto** del tipo contenuto X. In Java: il contenitore C ha una variabile membro privata di tipo **riferimento** ad un oggetto del tipo contenuto X. In entrambi i casi il contenitore C dovrà avere un costruttore che costruisce un oggetto c di tipo C con dentro anche il suo contenuto x di tipo X.

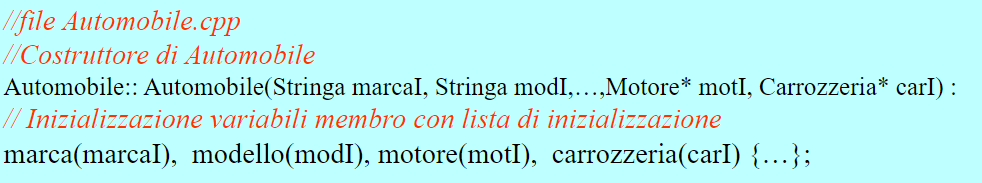




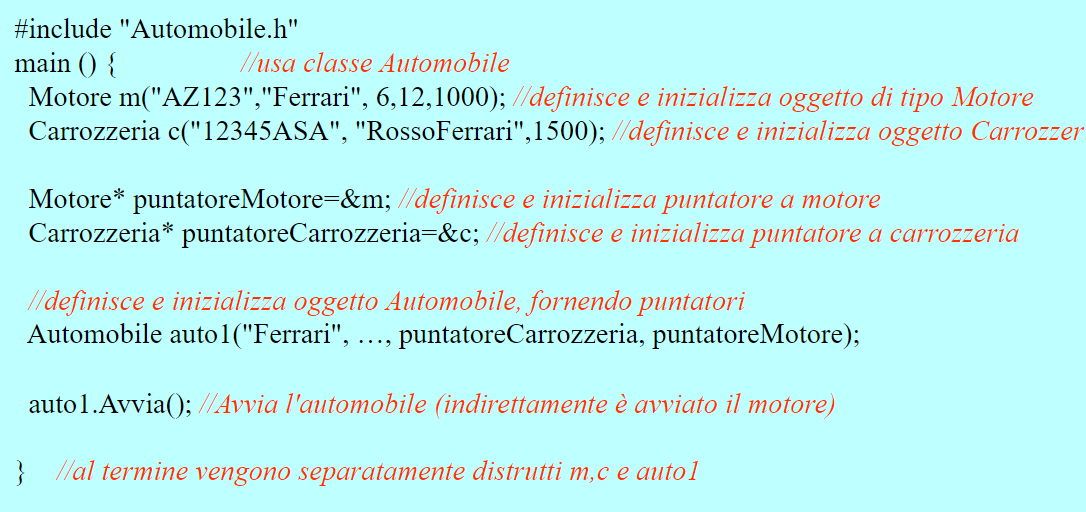
In particolare, l’aggregazione si può realizzare in C++ o in Java come segue:-In C++: il contenitore C ha una variabile membro privata di tipo **puntatore** ad un oggetto del tipo contenuto X, il contenitore dovrà avere un costruttore che riceve in input un puntatore ad un oggetto di tipo X. In Java: il contenitore C ha una variabile membro privata di tipo **riferimento** ad un oggetto del tipo contenuto X, il contenitore dovrà avere un costruttore che riceve in input un riferimento ad un oggetto di tipo X



Notiamo l'implementazione del costruttore nel seguente frammento del file di implementazione della classe Automobile (Automobile.cpp)



l programma principale dovrà creare un oggetto di tipo Carrozzeria e Motore e poi l'oggetto di tipo Automobile.



I diagrammi UML possono essere creati in diverse fasi del ciclo di vita, con diversi scopi e livelli di dettaglio.

**System domain model (Modello Concettuale dei dati del Sistema):** Modello del dominio dei dati del problema. Descrive aspetti del dominio del problema che saranno presenti nel sistema (i dati di interesse). A volte si aggiungono a questo modello anche le responsabilità delle classi.

**System model (Modello del Sistema):** Include anche classi che saranno usate per costruire il sistema completo. Classi per l’interfaccia utente (es. Windows, menu, form..). Classi associate a parti dell’architettura (client, server, file, database, etc.). Classi di utilità...

Nei diagrammi che descrivono il dominio del sistema non si inseriscono di solito metodi perchè essi rappresenterebbero un elemento del dominio della soluzione. In alternativa si possono utilizzare le Responsabilità.

Una responsabilità è un insieme di servizi e compiti che la classe dovrebbe garantire.

**Un processo di astrazione di un modello di dominio:**

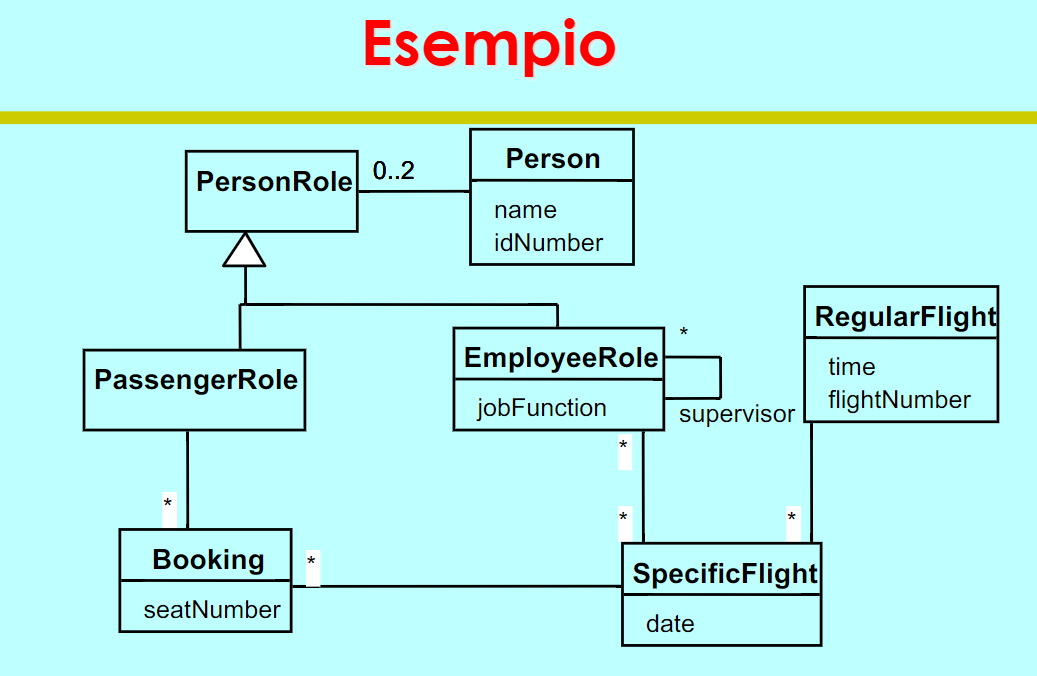
1. Identifica un primo insieme di classi candidate.
2. Aggiungi associazioni ed attributi a queste classi.
3. Trova le generalizzazioni.
4. Trova le principali responsabilità di ogni classe.
5. In base alle responsabilità decidi sulle operazioni.
6. Itera il processo finchè il modello ottenuto è soddisfacente.

Ad una classe dovrebbe corrispondere un’entità del dominio del problema. In pratica, bisogna applicare i concetti generali di modellazione Object Oriented. Una semplice tecnica per scoprire le classi del dominio è, analizzare la documentazione di partenza, come la descrizione dei requisiti ed estrarre **nomi e predicati nominali (aggettivi di nomi).** I sostantivi indicano candidati classi, attributi e ruoli. Le **espressioni verbali** che coinvolgono più classi indicano possibili relazioni tra esse. Una associazione dovrebbe esistere se una classe: Possiede o controlla, “è collegata a”, oppure “si riferisce a”, “è parte di”, oppure “ha come parti”, “è membro di” oppure, “ha come membri”, qualche altra classe del modello. Specificare le molteplicità da entrambi i lati. Assegnare un nome chiaro all’associazione. Cercare le informazioni che devono essere conservate per ciascuna classe. È possibile che nomi che sono stati scartati come classi, possano ora essere considerati attributi. Un attributo dovrebbe in genere contenere un solo valore. È bene non avere molti attributi ripetuti (es. più indirizzi). Se un sottoinsieme degli attributi di una classe forma un gruppo coerente (es. un indirizzo), crea una classe distinta per questi attributi.

**LEZIONE 14 08/04**

Due modi per trovare le generalizzazioni: **bottom-up :** Raggruppo classi simili creando una nuova superclasse. **top – down :** Cerco prima le classi più generali, e poi specializzo.

Crea un interfaccia, invece di una superclasse se: Le classi sono molto diverse fra loro, tranne che hanno poche operazioni in comune. Una o più classi hanno già le loro superclassi. Potrebbero essere disponibili diverse implementazioni della stessa classe.

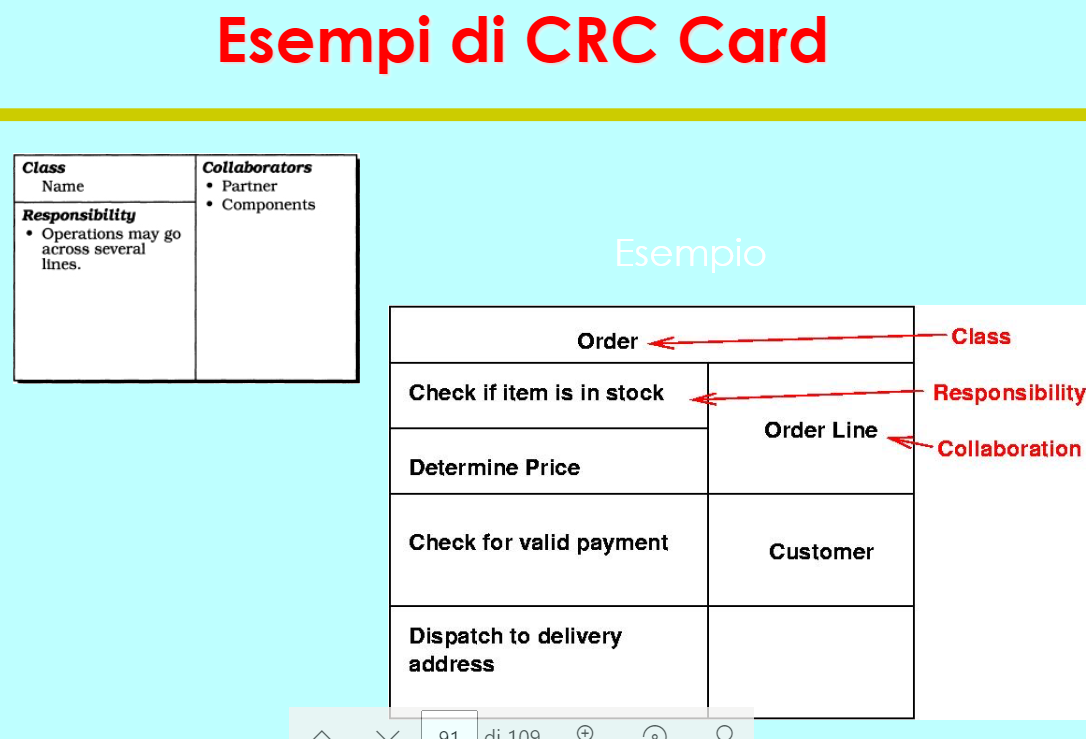


Una **responsabilità** è un qualcosa che è richiesto al sistema. La responsabilità di **ogni requisito funzionale** deve essere attribuita ad una delle classi, anche se tale requisito potrà essere svolto mediante una collaborazione fra più classi. Tutte le responsabilità di una classe dovrebbero essere chiaramente correlate. Se una classe ha troppe responsabilità, valutare l’ipotesi di dividerla in più classi. Se una classe non ha responsabilità, probabilmente è inutile. Quando una responsabilità non può essere attribuita a nessuna delle classi esistenti, dovrebbe essere creata una nuova classe.

Per stabilire le responsabilità dei requisiti funzionali: svolgere l’analisi dei casi d’uso, cercare verbi e nomi che descrivono azioni nella descrizione del sistema.

La responsabilità di creare istanze di una classe non può essere attribuita alla classe stessa, ma ad una classe collegata ad essa: **Pattern GRASP: CREATOR**. La responsabilità di cercare istanze di una classe che fanno parte di una collection (es. lista, catalogo, etc.) non può essere attribuita alla classe stessa, ma alla classe collection: **Pattern GRASP: INFORMATION EXPERT**. Cercare di attribuire ad una classe responsabilità che le richiedono di interagire soprattutto con le sue classi vicine: **Pattern: LOW COUPLING.**

Altra tecnica per ottenere un Class diagram: **CRC cards**. CRC sta per Class Responsibility Collaboration. Per ogni classe identificata, porre il nome della classe su una scheda (Card). Man mano che vengono individuati attributi e responsabilità, elencarli sulle Card. L’utilizzo delle card serve per imporre, all’analista di non realizzare classi con un numero troppo elevato di attributi e metodi. Se la card è piena allora probabilmente bisogna dividere la classe in due o più classi più semplici.



Le operazioni saranno necessarie a realizzare le responsabilità di ciascuna classe. Le operazioni che implementano una responsabilità sono normalmente dichiarate **public**. Altri metodi che collaborano a realizzare una responsabilità devono essere dichiarate **private** se possibile.