

Ingegneria del Software

Mario G.C.A. Cimino – Antonio Luca Alfeo



Unified Modeling Language e Unified Process Introduzione

Ingegneria del
Software

Introduzione allo Unified Modeling Language

1. Che cosa è lo Unified Modeling Language (UML)?

- i. È un linguaggio visuale, standard, aperto ed estensibile di modellazione, cioè è un linguaggio che fornisce una sintassi per costruire modelli. Nato per modellare sistemi software object-oriented, grazie ai meccanismi di estensione messi a disposizione dallo stesso linguaggio, viene attualmente utilizzato in una varietà di domini applicativi.
- ii. UML non fornisce nessuna metodologia di modellazione: può essere usato con qualsiasi metodologia esistente.

2. Perché unificato?

- i. UML fornisce una sintassi di modellazione visuale unica per l'intero ciclo di vita del software (dalle specifiche all'implementazione).
- ii. UML è stato usato per modellare differenti domini applicativi (da sistemi embedded a sistemi di supporto alle decisioni).
- iii. UML è indipendente dal linguaggio di programmazione e dalla piattaforma di sistema.

- iv. UML può supportare differenti metodologie di sviluppo.
- v. UML tenta di essere consistente ed uniforme nell'applicazione di un piccolo insieme di concetti predefiniti.

3. Specifica ufficiale di UML: www.uml.com

4. Quali premesse all'uso di UML per modellare sistemi?

I sistemi devono poter essere modellabili come collezioni di oggetti interagenti. Un oggetto è un contenitore di dati e comportamento e quindi contiene informazioni e può eseguire funzioni.

I modelli prendono in considerazione due aspetti:

- ✓ la **struttura statica**, che descrive quali tipi di oggetti sono importanti per modellare il sistema e come essi sono connessi,
- ✓ il **comportamento dinamico**, che descrive il ciclo di vita di questi oggetti e come essi interagiscono l'un l'altro per compiere le funzionalità richieste.

Introduzione allo Unified Process

Un processo di ingegnerizzazione del software (software engineering process – SEP), anche conosciuto come processo di sviluppo del software o metodo di sviluppo del software, definisce il *chi*, il *che cosa*, il *quando* ed il *come* dello sviluppo software. Lo Unified Software Development Process (USDP) è un SEP introdotto dagli stessi autori di UML. USDP è comunemente riferito come Unified Process o UP. UP si basa su pratiche consolidate ma non ben formalizzate, e come tale non è stato ancora standardizzato dall'OMG.



Ingegneria del
Software

UP è un processo software generico. Per poter essere utilizzato deve **essere adattato prima alle specifiche esigenze di ciascuna organizzazione e quindi allo specifico progetto**, sulla base dei relativi standard di qualità e di sicurezza, dei modelli di documento, degli strumenti di sviluppo e di amministrazione utilizzati nell'organizzazione, etc..

UP si basa su tre principi (assiomi):

- i) è **guidato dall'analisi dei requisiti e dei rischi** ("se non attacchi con forza i rischi, questi prima o poi attaccheranno te");
- ii) è **centrato sull'architettura** (architecture-centric), cioè è **finalizzato alla produzione di un'architettura robusta**, che descriva esattamente gli aspetti strategici di come il sistema è suddiviso in componenti e come questi interagiscono e sono dislocati sulle piattaforme hardware;
- iii) è **iterativo ed incrementale**, cioè suddivide il progetto in miniprogetti (le **iterazioni**) che rilasciano le funzionalità del sistema a pezzi, o incrementi, fino ad ottenere un sistema completamente funzionante. In altre parole, il software è prodotto attraverso un processo di raffinamento per passi successivi. L'idea dietro questo approccio è molto semplice: gli esseri umani trovano più semplice risolvere piccoli piuttosto che grandi problemi.

Ingegneria del
Software

Ciascuna iterazione contiene *tutti* gli elementi di un normale progetto, ma finalizzati ai sotto-obiettivi che la caratterizzano:

- pianificazione
- analisi e progetto
- costruzione
- integrazione e test
- release interna o esterna.

Ogni iterazione genera la cosiddetta *linea base (baseline)*, cioè una versione parzialmente completa del sistema finale e la documentazione associata. La differenza tra due baseline è conosciuta come incremento. Le iterazioni sono raggruppate in fasi che determinano la macrostruttura di UP.

In ogni iterazione, cinque flussi di lavoro (workflow) principali specificano che cosa deve essere fatto e quali capacità sono necessarie:

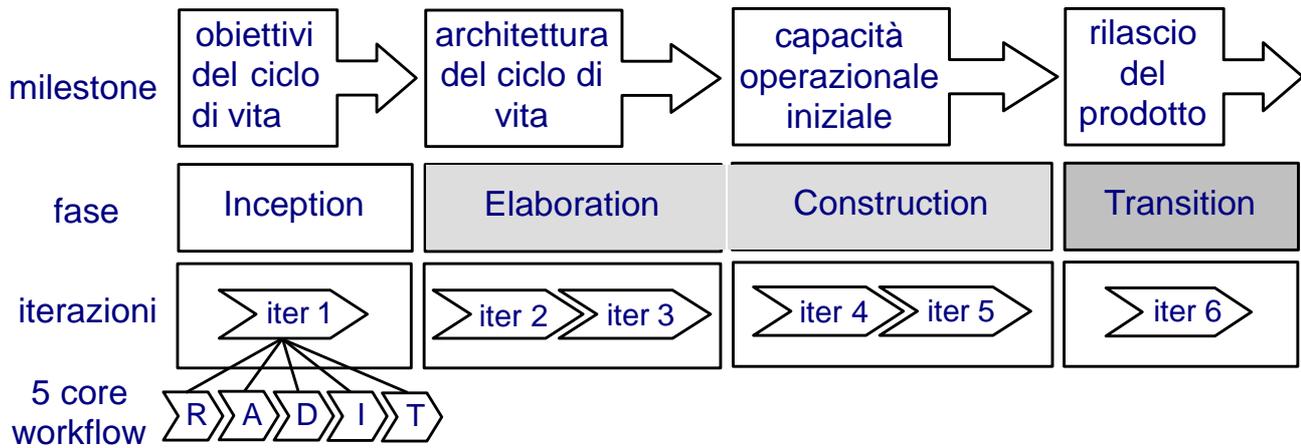
1. **R:** workflow requisiti (*requirements*): cattura che cosa il sistema dovrebbe fare;
2. **A:** workflow analisi (*analysis*): raffina e struttura i requisiti;
3. **D:** workflow progetto (*design*): realizza i requisiti attraverso un'architettura di sistema;
4. **I:** workflow implementazione (*implementation*): genera il software;
5. **T:** workflow test: verifica che l'implementazione lavori come desiderato.

L'enfasi su un particolare workflow dipende dalla fase in cui l'iterazione viene eseguita nel ciclo di vita del progetto.

La suddivisione in iterazioni permette un elevato grado di flessibilità: le iterazioni possono essere eseguite in sequenza oppure alcune di loro, quando possibile, possono essere eseguite in parallelo (mancanza di dipendenza tra gli artefatti). L'esecuzione parallela comporta una riduzione del time-to-market (tempo di commercializzazione), ma richiede una più accurata pianificazione.

La struttura di UP

Il ciclo di vita del progetto è diviso in quattro fasi: Principio (*Inception*), Elaborazione (*Elaboration*), Costruzione (*Construction*) e Transizione (*Transition*)

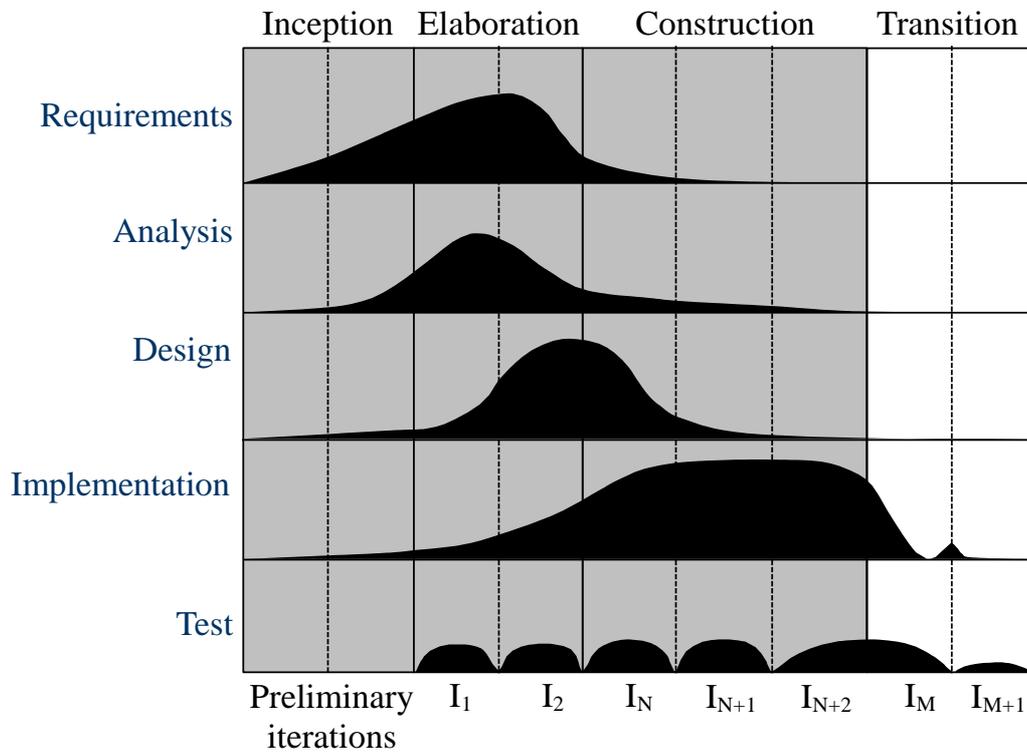


Ogni fase può avere una o più iterazioni (dipende dalla dimensione del progetto). In generale, le iterazioni non dovrebbero durare più di 2 o 3 mesi. Ogni fase termina con una pietra miliare (milestone).

Ingegneria del Software

Fase	Milestone
Inception	obiettivi del ciclo di vita
Elaboration	architettura del ciclo di vita
Constuction	capacità operativa iniziale
Transition	rilascio del prodotto.

UP è un processo basato sugli obiettivi piuttosto che su ciò che deve essere consegnato (*deliverable*): ogni fase finisce con una milestone che consiste di un insieme di condizioni che devono essere soddisfatte e queste condizioni possono portare alla creazione di un deliverable oppure no in base alle specifiche necessità del progetto



Le Fasi di UP

Ogni fase possiede un goal, un'attività principale concentrata su uno o più workflow e una milestone.

Inception

Goal: Far partire il progetto.

Questo richiede:

- *Stabilire la fattibilità:* può richiedere lo sviluppo di qualche prototipo sia per validare le decisioni tecnologiche che le richieste del business;
- *Creare un caso di business:* per dimostrare che il progetto produrrà benefici quantificabili al business;
- *Catturare le specifiche essenziali* per aiutare a scoprire correttamente il sistema;
- *Identificare i rischi critici.*

Worker principali: project manager e l'architetto del sistema.

Attività: l'attività principale è concentrata sui workflow Requisiti ed Analisi. Può essere prodotto qualche prototipo, ma non verrà effettuato alcun test.

Milestone: Obiettivi del ciclo di vita. La tabella seguente riassume le condizioni che devono essere soddisfatte per considerare raggiunta la milestone. La tabella mostra anche i deliverable che dovrebbero essere prodotti.

Condizioni	Deliverable
Le persone coinvolte sono d'accordo sugli obiettivi del progetto	Un documento d'insieme che stabilisce i requisiti principali, le caratteristiche ed i vincoli del progetto.
L'ambito del sistema è stato definito e concordato con le persone coinvolte	Un modello iniziale dei casi d'uso (completo solo al 10% - 20%)
I requisiti chiave sono stati catturati e concordati con le persone coinvolte	Un glossario di progetto
Le stime di costo e di schedulazione sono state concordate con le persone coinvolte	Un piano iniziale di progetto
Il manager del progetto ha prodotto un caso di business	Un caso di business
Il manager del progetto ha realizzato una stima dei rischi	Un documento o un database di stima dei rischi
È stata confermata la fattibilità attraverso studi tecnici e/o di prototipazione	Uno o più prototipi "usa e getta"
È stata tracciata un'architettura	Un documento di architettura iniziale

Ingegneria del
Software

Elaboration

Goal:

- *Creare un'architettura eseguibile;*
- *Perfezionare la valutazione del rischio;*
- *Definire gli attributi di qualità (tasso di difetti accettabile, etc.);*
- *Catturare almeno l'80% delle specifiche funzionali con i casi d'uso;*
- *Creare un piano dettagliato per la fase di costruzione;*
- *Formulare un'offerta che include risorse, tempo, staff, equipaggiamento.*

NOTA BENE: l'architettura creata durante la fase di Elaborazione non è un prototipo, ma è un sistema eseguibile reale costruito in accordo all'architettura specificata. Questo sistema sarà ulteriormente sviluppato durante le fasi successive fino ad arrivare al sistema finale che verrà rilasciato.

Ingegneria del
Software

Attività: l'attività considera tutti i workflow principali. In particolare:

- *Requirements* – raffina le specifiche del sistema;
- *Analysis* – stabilisce che cosa costruire;
- *Design* – identifica un'architettura stabile;
- *Implementation* – costruisce l'architettura;
- *Test* – verifica l'architettura.

Milestone: Architettura del ciclo di vita. La tabella seguente riassume le condizioni che devono essere soddisfatte per considerare raggiunta la milestone. La tabella mostra anche i deliverable che dovrebbero essere prodotti.

Condizioni	Deliverable
É stata creata un'architettura eseguibile dotata di flessibilità e robustezza.	L'architettura eseguibile
L'architettura eseguibile dimostra che sono stati identificati e risolti rischi importanti.	Il modello UML statico Il modello UML dinamico Il modello UML dei casi d'uso
La visione d insieme del prodotto si è stabilizzata	Il documento d'insieme
É stata riveduta la stima del rischio	Una stima aggiornata del rischio
I casi di business sono stati riveduti e concordati con i committenti	Un caso aggiornato di business
É stato creato un piano di progetto con dettaglio sufficiente da consentire la formulazione di un'offerta realistica in termini di tempo, denaro e risorse da destinare alle prossime fasi. Le persone coinvolte concordano sul piano di progetto. É stato verificato il caso di business con riferimento al piano di progetto.	Un piano aggiornato di progetto Caso di business
É stata raggiunta una intesa con le persone interessate per continuare il progetto	Documento di intesa

Construction

Goal: *completare le specifiche, l'analisi ed il progetto, ed evolvere l'architettura generata nella fase di Elaborazione nel sistema finale.*

ATTENZIONE: Mantenere l'integrità dell'architettura del sistema.

Attività: l'enfasi è sul workflow implementazione, ma qualche attività è anche fatta negli altri workflow per completare i requisiti, l'analisi e il progetto. Anche il testing comincia ad essere rilevante.

In dettaglio:

- *Requirements* – introduci ogni requisito mancante;
- *Analysis* – termina il modello di analisi;
- *Design* – termina il progetto;
- *Implementation* – costruisci la capacità operativa iniziale;
- *Test* – verifica la capacità operativa iniziale.

Ingegneria del
Software

Milestone: Capacità operativa iniziale. Il sistema è finito e pronto per il beta test al sito dell'utente. La tabella seguente riassume le condizioni che devono essere soddisfatte per considerare raggiunta la milestone. La tabella mostra anche i deliverable che dovrebbero essere prodotti.

Condizioni	Deliverable
Il prodotto software è sufficientemente stabile e di sufficiente qualità da essere distribuito nella comunità degli utenti	Il prodotto software Il modello UML Il corredo per il test
I committenti hanno raggiunto un'intesa e sono pronti per l'installazione del software in loco	Manuali utente Descrizione della versione rilasciata
Le spese sostenute sono accettabili in confronto alle spese pianificate.	Piano di progetto

Transition

Goal: *fissare i difetti trovati nel beta test e preparare l'installazione del software in tutti i site dell'utente.*

In dettaglio:

- *Correggere i difetti;*
- *Preparare i siti dell'utente per il nuovo software;*
- *Predisporre il software per operare nei siti dell'utente;*
- *Modificare il software se sorgono problemi imprevisti;*
- *Creare manuali utente ed altra documentazione;*
- *Fornire consulenza all'utente;*
- *Condurre una revisione del progetto.*

Attività: *l'enfasi è sui workflow implementazione and test. Qualche attività è anche fatta nel workflow progetto per correggere eventuali errori di progetto trovati nel beta test.*

Ingegneria del Software

Nessuna attività dovrebbe essere richiesta nei workflow requisiti ed analisi. Se questo non è il caso, allora il progetto ha sicuramente dei problemi.

In dettaglio:

- *Requirements* – non applicabile;
- *Analysis* – non applicabile;
- *Design* – modifica il progetto se si sono evidenziati problemi durante il beta test;
- *Implementation* – configura il software per il sito dell'utente e correggi eventuali problemi che si sono manifestati nel beta test;
- *Test* – sia il beta test che il test di accettazione svolti in loco (al sito dell'utente).

Milestone: Rilascio del prodotto. Il prodotto è rilasciato ed accettato dalla comunità dell'utente. La tabella seguente riassume le condizioni che devono essere soddisfatte per considerare raggiunta la milestone. La tabella mostra anche i deliverable che dovrebbero essere prodotti.

Condizioni	Deliverable
<p>Il beta testing è completato, le modifiche necessarie sono state effettuate e gli utenti concordano che il sistema sia stato installato con successo</p> <p>La comunità degli utenti adopera attivamente il prodotto</p>	<p>Il prodotto software</p>
<p>Le strategie di supporto al prodotto sono state concordate con gli utenti ed implementate</p>	<p>Il piano di supporto utenti, i manuali utente</p>

Ingegneria del Software

Mario G.C.A. Cimino - Antonio Luca Alfeo



Il workflow Requisiti

Ingegneria del
Software

Introduzione

L'obiettivo del workflow Requisiti è di individuare i requisiti del sistema, in modo da trovare un accordo su cosa il sistema deve fare, prima di cominciare a lavorare all'analisi ed al progetto. La maggior parte dell'attività richiesta da questo workflow è realizzata durante le fasi di Inception and Elaboration.

Per ogni dato sistema, ci sono molte persone differenti coinvolte: molti tipi di utenti, ingegneri della manutenzione, lo staff di supporto, i commerciali, i manager, etc.. Ognuna di queste persone ha delle esigenze e propone dei requisiti. L'**ingegneria dei requisiti** (requirements engineering) si occupa di estrarre i requisiti e dare loro delle priorità. A questo fine, può essere necessaria una negoziazione tra requisiti contrastanti.

Vi sono in generale due tipi di requisiti: **funzionali** (che cosa il sistema farà) e **non funzionali** (vincoli sul sistema dovuti a prestazioni, affidabilità, etc.). I casi d'uso possono solo catturare i requisiti funzionali.

La specifica dei requisiti dovrebbe essere la base di ogni sistema. Idealmente i requisiti dovrebbero descrivere *soltanto cosa* il sistema dovrà fare e non **come** dovrà farlo. Tuttavia, non di rado tale distinzione viene ignorata: è più semplice scrivere e capire una descrizione implementativa piuttosto che una descrizione astratta del problema. Inoltre, spesso vi sono dei vincoli implementativi che predeterminano il "come" del sistema.

Ingegneria del
Software

Molte aziende non adottano ancora una nozione formale di requisito: spesso vengono utilizzati dei documenti di specifica scritti in linguaggio naturale. Qualsiasi sia la forma utilizzata, ci si deve chiedere:

- Quanto è utile questo requisito?
- Aiuta a capire cosa il sistema dovrebbe fare?

Il modello dei casi d'uso viene generato tipicamente mediante appositi strumenti di modellazione UML, mentre il modello dei requisiti (requirements model) è creato in forma testuale o con speciali strumenti adottati nell'ingegneria dei requisiti.

Definire i requisiti

UML non fornisce nessuna raccomandazione per scrivere i requisiti tradizionali (UML gestisce i requisiti solo attraverso i casi d'uso).

Si raccomanda l'uso di un formato molto semplice, del tipo:

<id> Il <nome del sistema> deve <funzione da realizzare>

dove <id> è l'identificatore unico del requisito.

Ad esempio, di seguito vengono elencati alcuni requisiti funzionali e non-funzionali per un bancomat (automated teller machine – ATM):

Requisiti funzionali:

1. Il sistema ATM deve controllare la validità della carta inserita
2. Il sistema ATM deve convalidare il PIN digitato dal cliente
3. Il sistema ATM deve dispensare non più di € 250 per carta nelle 24 ore

Requisiti non funzionali:

1. Il sistema ATM deve essere scritto in C++
2. Il sistema ATM deve comunicare con la banca usando un codice cifrato a 256-bit
3. Il sistema ATM deve convalidare la carta entro 3 secondi
4. Il sistema ATM deve convalidare il PIN entro 3 secondi

All'aumentare dei requisiti è opportuno organizzarli in una tassonomia. Questa è una gerarchia di tipi di requisiti che può essere usata per raggruppare i requisiti stessi. Per esempio

Requisiti funzionali

_____	Clienti
_____	Prodotti
_____	Ordini
_____	Canali di vendita
_____	Pagamenti

Requisiti non funzionali

_____	Prestazioni
_____	Capacità
_____	Disponibilità
_____	Adesione a standard
_____	Sicurezza

In generale, due o tre livelli di profondità della gerarchia sembrano appropriati quando non si lavora con sistemi particolarmente complessi.

Ciascun requisito può avere un insieme di **attributi**, che cattura informazione extra circa il requisito stesso. Ogni attributo ha un nome descrittivo ed un valore. Ad esempio, Data = "31/10/2005" oppure Sorgente = "Esperto del Dominio". Probabilmente

l'attributo più comunemente utilizzato è *priority*. Uno schema comune per assegnare la priorità è l'insieme di criteri MoSCoW, riportati nella Tabella seguente:

Valori di Priorità	Semantica
Must have	Requisito fondamentale per il sistema
Should have	Requisito importante che però può essere omesso
Could have	Requisito opzionale (da realizzare se c'è tempo).
Want to have	Requisiti che possono essere realizzati in successive release

Sebbene MoSCoW sia molto semplice da usare confonde le dimensioni di importanza e precedenza. Mentre l'importanza di una specifica tende a rimanere stabile, la sua precedenza può cambiare durante il progetto (per esempio, per la mancanza di disponibilità delle risorse).

Individuare i requisiti

I requisiti sono generati dal contesto del sistema che si sta cercando di modellare. Questo contesto include:

- Gli utenti del sistema;
- Altre persone coinvolte (manager, installatori, etc.);
- Altri sistemi con cui il sistema deve interagire;
- Dispositivi hardware con cui il sistema interagisce;
- Vincoli legali e di regolamento;
- Vincoli tecnici;
- Obiettivi del business.

Il processo di ingegnerizzazione dei requisiti inizia generalmente con un documento di visione d'insieme (vision document), scritto in linguaggio naturale, che delinea ciò che il sistema farà e quali benefici apporterà al gruppo di persone coinvolte nel progetto. Lo scopo di questo documento è catturare gli obiettivi essenziali del sistema dal punto di vista delle diverse persone coinvolte.

Ingegneria del
Software

Il documento non è che il primo passo per l'individuazione dei requisiti. Per completare la specifica dei requisiti, tipicamente vengono usate quattro tecniche differenti: deduzione dei requisiti, interviste, questionari e gruppi di lavoro.

Deduzione dei requisiti (requirements elicitation)

Con questa tecnica si cerca di dedurre i requisiti dalle persone coinvolte nel progetto. In particolare, si cerca di estrarre un quadro o mappa del loro modello del mondo.

ATTENZIONE: questa mappa è corrotta da tre processi che gli esseri umani generalmente compiono per limitare la complessità del mondo reale (secondo la teoria di N. Chomsky [1975, Syntactic Structures]):

- **Cancellazione** – l'informazione è filtrata e parzialmente rimossa;
- **Distorsione** – l'informazione è modificata dai meccanismi correlati di creatività e fantasia;
- **Generalizzazione** - astrazione dell'informazione in regole, opinioni e principi.

Conoscere l'effetto dei questi tre filtri è importante quando si devono individuare requisiti dettagliati. Le informazioni mancanti vengono ricostruite mediante successive risposte date a specifiche richieste di chiarimento. Per esempio, la tabella seguente mostra gli effetti dei tre

Ingegneria del
Software

filtri e come recuperare l'informazione nel processo di identificazione dei requisiti di un sistema di gestione di una biblioteca.

REQUISITO	FENOMENO DI DISTURBO	RICHIESTA DI CHIARIFICAZIONE	RISPOSTA
“Essi usano il sistema per prestare dei libri.”	cancellazione (estensione degli utenti)	“In particolare, chi usa il sistema per prestare libri?”	“Dipendenti di una biblioteca, altre biblioteche e bibliotecari.”
“Non si può prendere in prestito un libro se non si restituiscono tutti i libri a prestito scaduto”	distorsione (modifica alla regola)	“Vi sono circostanze in cui si potrebbe prestare un libro a chi ne ha altri con prestito scaduto?”	“Sì, nel caso in cui i libri a prestito scaduto siano stati ripagati.”
“Ognuno deve avere una ricevuta per il libro in prestito”	generalizzazione (estensione della regola)	“Esiste un utente del sistema che potrebbe non aver bisogno della ricevuta?”	“Alcuni utenti del sistema (altre biblioteche) possono non aver bisogno di alcuna ricevuta oppure di un tipo speciale di ricevuta con diversi termini e condizioni.”

In generale, ogniqualvolta si trova un quantificatore universale (tutti (all), ognuno (everyone), sempre (always), mai (never), nessuno (nobody), niente (none)) potrebbe essersi verificata una cancellazione, una distorsione o una generalizzazione.

Ingegneria del Software

Interviste

È il modo più diretto. Si deve cercare di intervistare le persone coinvolte nel progetto una alla volta, facendo attenzione a:

- *non proporre soluzioni* – le buone idee personali possono non essere le idee delle persone intervistate;
- *fare domande libere dal contesto* - ossia che non richiedono una semplice risposta “sì/no”, ma incoraggiano la discussione (per esempio, non chiedere “Usi il sistema?”, ma piuttosto “Chi usa il sistema?”);
- *ascoltare* - dare il tempo agli intervistati di parlare;
- *non interpretare il pensiero* – spesso l’interpretazione del pensiero altrui è l’interpretazione del proprio pensiero;
- *avere pazienza*;
- *adoperare strumenti semplici* (carta e penna).

Questionari e Gruppi di lavoro

Questionari sono utili supplementi alle interviste. Il gruppo di lavoro (workshop) dei requisiti è uno dei modi più efficienti per catturare informazione. Si basa su un libero scambio di vedute (brainstorm) a cui partecipano un moderatore, un ingegnere dei requisiti, gli utilizzatori chiave ed esperti del dominio.

Ingegneria del Software

La modellazione dei casi d'uso

La modellazione dei casi d'uso è anch'essa una forma dell'ingegneria dei requisiti e procede tipicamente nel modo seguente:

- Identifica un confine candidato del sistema;
- Trova gli attori;
- Trova i casi d'uso;
 - specifica i casi d'uso;
 - identifica i flussi alternativi;
- itera i punti precedenti finché i casi d'uso, gli attori e i confini del sistema non sono stabili.

Il risultato di queste attività è il modello dei casi d'uso.

A partire dal modello del business o del dominio applicativo, da un modello dei requisiti e da una lista di caratteristiche, l'analista del sistema produce il modello dei casi d'uso ed un glossario di progetto. Esaminiamo i singoli passi in dettaglio.

Identifica il confine del sistema

Identifica che cosa è parte del sistema e che cosa non lo è. L'individuazione corretta dei confini del sistema consentono di determinare correttamente le specifiche funzionali. In UML 2 i confini del sistema sono chiamati **soggetto**. Il soggetto è definito da chi o che cosa usa il sistema (gli attori) e da quali benefici il sistema offre a questi attori (i casi d'uso).

Il soggetto è rappresentato da un rettangolo, con gli attori all'esterno e i casi d'uso all'interno. Inizialmente la modellazione dei casi d'uso verrà intrapresa avendo solo una vaga idea del soggetto: questa vaga idea diventerà sempre più chiara man mano che vengono definiti attori e casi d'uso.



Trova gli attori

L'**attore** è un ruolo che qualche entità esterna interpreta in qualche contesto quando interagisce *direttamente* con il sistema. Tipicamente è un utente, ma può essere un altro sistema, un pezzo di hardware, il tempo (es. salvataggi automatici) o qualsiasi cosa che venga a contatto con il sistema.

In UML 2, gli attori possono essere anche altri soggetti e questo permette di collegare tra loro modelli di casi d'uso.

Un ruolo è come un cappello che viene indossato in un particolare contesto. Un'entità può avere ruoli differenti e quindi essere attori differenti. Per esempio, un sistema potrebbe avere Customer e SystemAdministrator come attori. Jim potrebbe sia amministrare il sistema che usare il sistema. Quindi, Jim può interpretare sia il ruolo di Customer che di SystemAdministrator.

Gli attori sono sempre esterni al sistema. Il sistema comunque potrebbe avere una rappresentazione interna degli attori. L'attore Customer è esterno al sistema, ma il sistema potrebbe avere una classe CustomerDetails, che è una rappresentazione interna dell'attore.

Ingegneria del
Software

Per identificare gli attori, devi considerare chi o che cosa usa il sistema, e quali ruoli interpretano nelle loro interazioni con il sistema. Le domande seguenti possono essere un valido aiuto:

- Chi o cosa usa il sistema?
- Quali ruoli interpretano nell'interazione con il sistema?
- Chi installa il sistema?
- Chi o cosa avvia e termina il sistema?
- Chi mantiene il sistema?
- Quali altri sistemi interagiscono con il sistema?
- Chi o cosa riceve/fornisce informazioni da/al sistema?
- Avviene qualcosa a tempi prefissati?

In termini di modellazione, si deve tener presente che:

- gli attori sono sempre esterni;
- gli attori interagiscono direttamente con il sistema;
- un attore non è una specifica persona o cosa, ma piuttosto un ruolo che questa persona o cosa interpreta in relazione al sistema;

Ingegneria del
Software

- una persona o una cosa può interpretare molti ruoli;
- ogni attore deve essere individuato da un nome corto, significativo per il dominio applicativo;
- ad ogni attore deve essere associata una descrizione per illustrare cosa l'attore è nel dominio applicativo;
- il modello dell'attore può mostrare attributi ed eventi che l'attore può ricevere (quasi mai usati).

Trova i casi d'uso

Un **caso d'uso** è “una specifica di sequenze di azioni, incluse sequenze alternative e di errore, che un sistema, sottosistema o classe possono eseguire interagendo con attori esterni”.

Un caso d'uso è *sempre attivato* da un attore ed è *sempre scritto* dal punto di vista dell'attore.

I casi d'uso sono rappresentati come ovali, con all'interno il nome del caso d'uso.

Il miglior modo per identificare i casi d'uso è il seguente:

- esamina la lista degli attori;

Ingegneria del Software

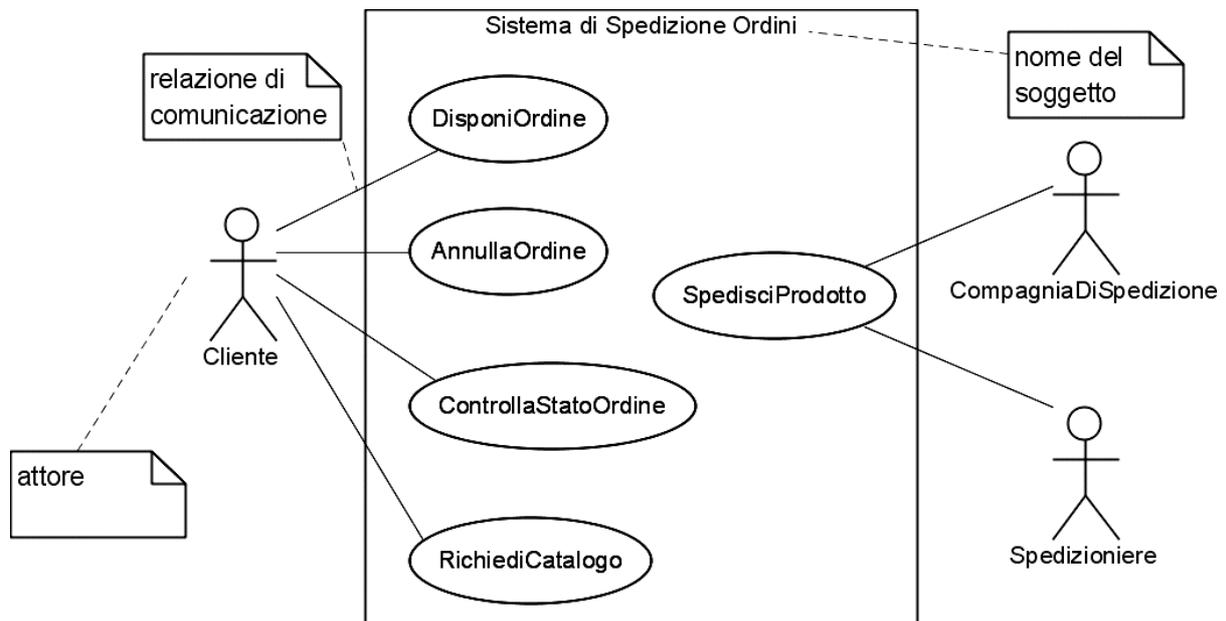
- considera come ogni attore usa il sistema;
- determina una lista di casi d'uso candidati;
- individua ogni caso d'uso attraverso una frase con un verbo (DisponiOrdine, CancellaOrdine, etc.);
- Se necessario, introduci nuovi attori.

L'individuazione dei casi d'uso è un'attività iterativa e procede attraverso un raffinamento per passi successivi. Si inizia con identificare il caso d'uso con un nome, quindi con una breve descrizione ed infine con una specifica completa.

Di seguito, viene presentata una lista di domande che possono risultare utili nell'identificare i casi d'uso.

- Quali funzioni un attore desidera che il sistema fornisca?
- Se il sistema memorizza e recupera informazioni, chi avvia tale comportamento?
- Vi sono attori a cui viene notificato il cambio di stato del sistema (avviamento, terminazione)?

- Qualche evento esterno influenza il sistema? Cosa informa il sistema di tali eventi?
- Il sistema interagisce con qualche sistema esterno?
- Il sistema genera qualche rapporto?



Il glossario di progetto

Il **glossario di progetto** è uno degli artefatti più importanti, in quanto fornisce un dizionario di termini-chiave e di definizioni usati nel dominio, comprensibili a chiunque sia coinvolto nel progetto. In pratica, il glossario cerca di catturare il linguaggio usato nello specifico dominio.

Oltre a definire i termini principali usati nel dominio, il glossario deve anche risolvere il problema dei sinonimi e degli omonimi. I **sinonimi** sono parole diverse con un'unica semantica, es. *alloggio* ed *abitazione*. Nel modello, la semantica deve essere individuata da una sola parola. Gli **omonimi** sono parole che hanno significati diversi e possono creare problemi di comunicazione. Nel modello, si dovrebbe assegnare ad ogni nome un unico significato.

Nel glossario, dovrebbero essere memorizzati i termini preferiti e inserita una lista di sinonimi sotto la loro definizione.

I termini usati nel modello UML devono essere consistenti con i termini definiti nel glossario. Tipicamente, i glossari vengono scritti in forma testuale su un file. Per progetti complessi, possono essere organizzati in semplici basi di dati. Di seguito viene dato un esempio di glossario

Dettagliare i casi d'uso

Il nome (unico nel modello) dovrebbe essere un verbo o una frase con un verbo e deve dare una chiara idea della funzione del business o del processo. Tutte le parole dovrebbero avere la prima lettera maiuscola (UpperCamelCase).	Use case: <i>PagaTassaVendita</i>
Tipicamente un numero. In casi d'uso con flussi alternativi, è utile utilizzare un sistema di numerazione gerarchico: 1.1, 1.2,...	ID: 1
Paragrafo che cattura l'obiettivo del caso d'uso	Brief description: <i>Paga la tassa di vendita all'autorità fiscale alla fine del trimestre d'esercizio</i>
Attori che avviano il caso d'uso	Primary actors: <i>tempo</i>
Attori che interagiscono con il caso dopo l'avvio	Secondary actors: <i>autorità fiscale</i>
Vincoli sullo stato del sistema prima dell'avvio del caso d'uso (condizioni booleane)	Preconditions: <i>1. È la fine del trimestre d'esercizio</i>
Passi elencati come flusso di eventi	Main flow: <i>1. Il caso d'uso inizia quando è la fine del trimestre d'esercizio</i> <i>2. Il sistema determina l'ammontare della tassa di vendita dovuta all'autorità fiscale</i> <i>3. Il sistema esegue un pagamento elettronico all'autorità fiscale</i>
Vincoli sullo stato del sistema dopo l'esecuzione del caso d'uso (condizioni booleane)	Postconditions: <i>1. L'autorità fiscale riceve il corretto importo della tassa di vendita</i>
Eventuali alternative	Alternative flows:

Ingegneria del
Software

OSSERVAZIONI:

L'identificatore numerico può rivelarsi necessario perché i nomi dei casi d'uso, seppur unici nel modello, possono cambiare durante lo sviluppo.

I casi d'uso sono **sempre avviati da un singolo attore**. Comunque, lo stesso caso d'uso può essere avviato da attori differenti ad istanti differenti. Ogni attore che può avviare il caso d'uso è un attore primario. Tutti gli altri sono secondari.

I passi di un caso d'uso sono elencati in un flusso di eventi. Ogni caso d'uso ha un flusso principale e può avere dei flussi alternativi.

Il flusso *principale*, chiamato anche lo *scenario primario*, descrive il flusso "ideale", caratterizzato dalla mancanza di errori, deviazioni, interruzioni o diramazioni, mentre i *flussi alternativi*, chiamati *scenari secondari*, catturano le anomalie che si verificano rispetto al flusso principale.

Il flusso principale è attivato dall'attore primario. Un modo classico per iniziare il flusso di eventi può essere:

1. *Il caso d'uso inizia quando un <attore><funzione>*

Il flusso di eventi è una sequenza di passi (espressi con frasi corte) di tipo dichiarativo, che sono numerati e ordinati nel tempo:

Ingegneria del
Software

<numero> Il <qualcosa><qualche azione>

Il caso d'uso deve essere una precisa dichiarazione di un frammento di funzionalità del sistema, quindi è importante riconoscere ed eliminare i processi di cancellazione, distorsione e generalizzazione individuati da Chomsky.

Per esempio, in un caso d'uso DisponiOrdine, un passo del flusso potrebbe essere:

2. Vengono inseriti i dati del cliente

Un passo scritto in forma passiva tipicamente è mal formulato, cioè lascia ambiguità su **chi, cosa, quando o dove**: chi è che inserisce tali dettagli? Dove sono inseriti ? E quali sono tali dettagli ? Ecco una forma più corretta:

1. Il caso d'uso inizia quando il cliente seleziona “disponi ordine”

2. Il cliente inserisce il proprio nome ed indirizzo nella scheda

Quando nella specifica del caso d'uso si incontrano distorsione, cancellazione o generalizzazione, queste vanno eliminate anche quando in base al contesto si possa risalire al significato. A questo proposito è utile chiedersi:

Chi specificatamente? Che cosa specificatamente? Quando specificatamente? Dove specificatamente?

UML non specifica nessun modo standard per presentare diramazioni all'interno di un flusso. In questo corso, sceglieremo di descrivere semplici varianti del flusso principale utilizzando semplici costrutti piuttosto che flussi alternativi separati.

Parola chiave **IF**

Sintassi:

- n. IF** *boolean expression*
1. *do something.*
 2. *do something else.*
 3. ...
- n+1. **ELSE**
- n+1.1 *do something.*
- n+2.

Use case: <i>GestisciCarrello</i>
ID: 2
Brief description: <i>Il cliente cambia la quantità di un oggetto nel carrello</i>
Primary actors: <i>Cliente</i>
Secondary actors: <i>Nessuno</i>
Preconditions: <i>1. Il contenuto del carrello è visibile</i>
Main flow: <i>1. Il caso d'uso inizia quando il cliente seleziona un oggetto nel carrello</i> 2. IF <i>il cliente seleziona "delete item"</i> <ol style="list-style-type: none">1. <i>Il sistema rimuove l'oggetto dal carrello</i> 3. IF <i>il cliente digita una nuova quantità</i> <ol style="list-style-type: none">1. <i>Il sistema aggiorna la quantità dell'oggetto nel carrello</i>
Postconditions: <i>Nessuna</i>
Alternative flows: <i>Nessuno</i>

Parola chiave **FOR**

Sintassi:

- n. FOR** *iteration expression*
1. *do something*
 2. *do something else*
 3. ...
- n+1.

Use case: <i>TrovaProdotto</i>
ID: 3
Brief description: <i>Il sistema trova alcuni prodotti che soddisfano i criteri di ricerca del cliente e li visualizza</i>
Primary actors: <i>Cliente</i>
Secondary actors: <i>Nessuno</i>
Preconditions: <i>Nessuna</i>
Main flow: <i>1. Il caso d'uso inizia quando il cliente seleziona "find product".</i> <i>2. Il sistema chiede al cliente i criteri di ricerca.</i> <i>3. Il cliente inserisce i criteri di ricerca.</i> <i>4. Il sistema ricerca i prodotti che soddisfano i criteri di ricerca.</i> 5. IF <i>il sistema trova dei prodotti</i> 5. FOR <i>ogni prodotto trovato</i> <ol style="list-style-type: none">1. <i>Il sistema visualizza una descrizione concisa del prodotto</i>2. <i>Il sistema mostra il prezzo del prodotto</i> 6. ELSE <ol style="list-style-type: none">5.5 <i>Il sistema comunica al cliente che nessun prodotto è stato trovato</i>
Postconditions: <i>Nessuna</i>
Alternative flows: <i>Nessuno</i>

Parola chiave **WHILE**

Sintassi:

n. **WHILE** *boolean expression*

1. *do something*
2. *do something else*
3. ...

n+1.

Use case: <i>MostraDettagliAzienda</i>
ID: 4
Brief description: <i>Il sistema mostra i dettagli dell'azienda al cliente</i>
Primary actors: <i>Cliente</i>
Secondary actors: <i>Nessuno</i>
Preconditions: <i>Nessuna</i>
Main flow: <i>1. Il caso d'uso inizia quando il cliente seleziona "show company details".</i> <i>2. Il sistema visualizza una pagina web che mostra i dettagli dell'azienda</i> 3. WHILE <i>il cliente osserva i dettagli dell'azienda</i> <i>1. il sistema suona una musica in sottofondo</i> <i>2. il sistema visualizza le offerte speciali</i>
Postconditions: <i>1. Il sistema ha visualizzato i dettagli dell'azienda</i> <i>2. Il sistema ha suonato la musica in sottofondo</i> <i>3. Il sistema ha visualizzato le offerte.</i>
Alternative flows: <i>Nessuno</i>

Ingegneria del
Software

Per gestire errori, alternative importanti o interruzioni occorrono flussi alternativi al principale. I flussi alternativi frequentemente non ritornano al flusso principale (per esempio, quando vengono gestiti errori o eccezioni). Inoltre, spesso hanno proprie postcondizioni.

I flussi alternativi possono essere documentati separatamente oppure appesi alla fine del caso d'uso. Per leggibilità è preferibile definire i flussi alternativi come casi d'uso separati. Il nome e l'identificatore del caso d'uso che implementa il flusso alternativo sono denotati come:

Alternative flow: *<nome caso d'uso con flusso principale> : <nome del flusso alternativo>*

ID: *< id caso d'uso principale> . <id del flusso alternativo>*

Ingegneria del
Software

Use case: <i>CreaNuoviAccount</i>
ID: 5
Brief description: <i>Il sistema crea un nuovo account al cliente</i>
Primary actors: <i>Cliente</i>
Secondary actors: <i>Nessuno</i>
Preconditions: <i>Nessuna</i>
Main flow: <ol style="list-style-type: none"> 1. <i>Il caso d'uso inizia quando il cliente seleziona "create new customer account".</i> 2. WHILE <i>i dati del cliente non sono validi</i> <ol style="list-style-type: none"> 1. <i>Il sistema chiede al cliente di inserire i propri dati: indirizzo e-mail, password e password di nuovo per conferma</i> 2. <i>Il sistema controlla i dati del cliente</i> 3. <i>il sistema crea un nuovo account per il cliente</i>
Postconditions: <ol style="list-style-type: none"> 1. <i>Un nuovo account viene creato</i>
Alternative flows: <i>IndirizzoEMailNonValido</i> <i>PasswordNonValida</i> <i>Cancella</i>

Alternative flow: <i>CreaNuoviAccount : IndirizzoEMailNonValido</i>
ID: 5.1
Brief description: <i>Il sistema informa il cliente che ha inserito un indirizzo di e-mail non valido</i>
Primary actors: <i>Cliente</i>
Secondary actors: <i>Nessuno</i>
Preconditions: <i>Il cliente ha inserito un indirizzo e-mail non valido</i>
Alternative flow: <ol style="list-style-type: none"> 1. <i>Il flusso alternativo inizia dopo il passo 2.2 del flusso principale.</i> 2. <i>Il sistema informa il cliente che ha inserito un indirizzo e-mail non valido</i>
Postconditions: <i>Nessuna</i>

Ingegneria del Software

NOTA BENE: un flusso alternativo non dovrebbe avere a sua volta un flusso alternativo. In caso contrario, il modello dei casi d'uso diventerebbe troppo complesso da seguire.

Il flusso alternativo può essere attivato in tre modi differenti:

1. attivato al posto del flusso principale per scelta dell'attore primario;
2. attivato dopo un passo del flusso principale – il flusso alternativo dovrebbe cominciare con:
 1. Il flusso alternativo comincia dopo il passo X del flusso principale
3. ad ogni passo del flusso principale – il flusso alternativo dovrebbe cominciare con:
 1. Il flusso alternativo può cominciare in ogni momento

Come esempio di quest'ultimo caso consideriamo il flusso alternativo Cancellata.

Alternative flow: <i>CreaNuoviAccount : Cancellata</i>
ID: 5.2
Brief description: <i>Il cliente sospende il processo di creazione</i>
Primary actors: <i>Cliente</i>
Secondary actors: <i>Nessuno</i>
Preconditions: <i>Nessuna</i>
Alternative flow: <i>1. il flusso alternativo può cominciare in ogni momento.</i> <i>2. il cliente sospende il processo di creazione dell'account</i>
Postconditions: <i>1. Nessun nuovo account è stato creato per il cliente</i>

Se il flusso alternativo deve ritornare al flusso principale, si può usare:

N. Il flusso alternativo ritorna al passo M del flusso principale.

Come trovare i flussi alternativi?

I flussi alternativi vengono identificati analizzando il flusso principale e cercando possibili alternative, errori che si possono generare nel flusso principale, interruzioni che potrebbero verificarsi in un punto particolare o in qualunque punto del flusso principale.

Quanti dovrebbero essere i flussi alternativi?

Il numero di flussi alternativi dovrebbe essere il minimo indispensabile. Ci sono due strategie:

1. Considera solo i flussi alternativi più importanti e documenta quelli;
2. Quando vi sono gruppi simili di flussi alternativi, documentane uno a scopo esemplificativo ed aggiungi delle note per spiegare le differenze con gli altri.

Ricorda sempre che i casi d'uso vengono usati per capire il **comportamento desiderato** del sistema, e non per creare un modello completo di casi d'uso. Quindi, la modellazione dei casi d'uso dovrebbe terminare non appena è stata raggiunta una soddisfacente comprensione del sistema. Ricorda in ogni caso che UP è un metodo iterativo.

Confrontare requisiti e casi d'uso

Una volta terminati il modello dei requisiti ed il modello dei casi d'uso, i due modelli vanno confrontati per capire se ogni requisito nel modello dei requisiti è coperto dai casi d'uso e viceversa. Eseguire questo confronto non è banale perché la relazione tra requisiti funzionali e casi d'uso è del tipo molti-a-molti. Il processo di confrontare i requisiti con i casi d'uso potrebbe essere fatto durante la generazione dei casi d'uso: usando i valori etichettati, è possibile associare una lista di requisiti, ognuno associato al proprio identificatore, ad ogni caso d'uso.

Un utile strumento per verificare la consistenza tra requisiti e casi d'uso è la **matrice di tracciabilità**, presentata di seguito.

		Use case			
		UC ₁	UC ₂	UC ₃	UC ₄
Requirement	R ₁	X			
	R ₂		X	X	
	R ₃			X	
	R ₄				X
	R ₅	X			

Ingegneria del
Software

La X indica che il requisito indicato nella riga ed il caso d'uso indicato nella colonna si riferiscono alla stessa specifica funzionale.

Se un requisito non corrisponde a nessun caso d'uso, allora un caso d'uso è mancante. Viceversa, se nessun requisito corrisponde ad un caso d'uso, allora il modello dei requisiti è incompleto.

Quando si applica la modellazione con i casi d'uso?

I casi d'uso rappresentano i requisiti funzionali e quindi sono adatti per sistemi con tanti utenti con diverse funzionalità, e con tante interfacce. Il modello dei requisiti va bene invece per sistemi embedded e sistemi algoritmicamente complessi, con poche interfacce e poca varietà di utenti.

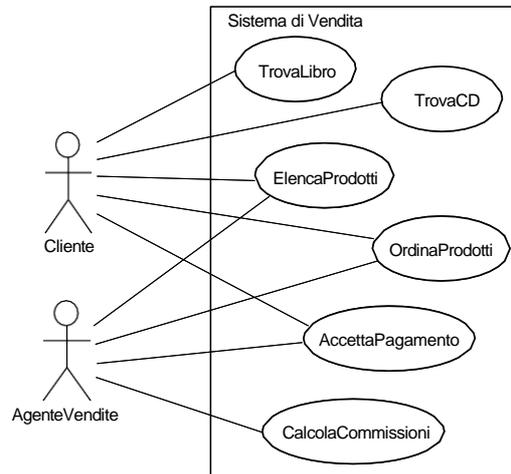
Ingegneria del
Software

La modellazione avanzata basata sui casi d'uso

Per migliorare la chiarezza e semplificare la semantica, si possono adoperare le **relazioni** tra casi d'uso o tra attori. Nel seguito, discuteremo queste relazioni.

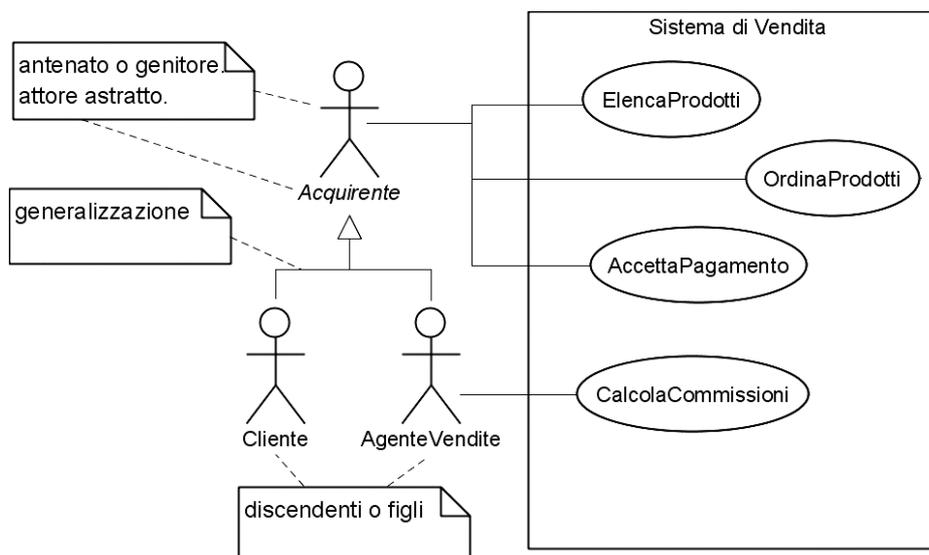
Generalizzazione degli attori

Nel diagramma seguente dei casi d'uso, si può notare come i due attori Cliente e AgenteVendite abbiano dei casi d'uso in comune.



Ingegneria del
Software

Questo comportamento comune potrebbe essere fattorizzato (eliminando così alcune intersezioni) inserendo un *attore generalizzato*. La figura seguente mostra come inserendo l'attore astratto *Acquirente* il diagramma si semplifichi notevolmente. Ovviamente, *AgenteVendite* e *Cliente* ereditano tutti i ruoli e le relazioni di casi d'uso del loro genitore.



Ingegneria del
Software

Generalizzazione dei casi d'uso

La generalizzazione dei casi d'uso è usata quando si hanno due o più casi d'uso che sono una specializzazione di un caso d'uso più generale. Le specializzazioni (i figli) ereditano tutte le caratteristiche (relazioni, punti di estensione, precondizioni, postcondizioni, flusso principale e flussi alternativi) del genitore, possono estenderle e sovrascriverle (ad eccezione delle relazioni e dei punti di estensione, che non possono essere sovrascritti).

All'interno delle specifiche, il caso d'uso padre non dovrà contenere alcuna annotazione dei casi d'uso figlio, mentre questi dovranno riferire le caratteristiche ereditate o sovrascritte.

Come estendere o modificare i passi dei flussi? Due regole:

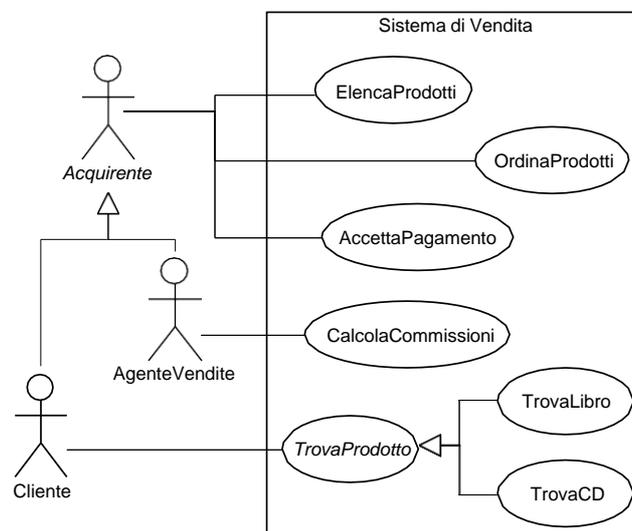
1. Ogni numero di passo nel figlio è seguito dall'equivalente numero nel genitore tra parentesi tonde. Per esempio: 1.(2.) *Questo primo passo figlio è stato ereditato dal secondo passo padre*
2. Se il passo nel figlio sovrascrive il passo nel genitore, il numero del passo nel figlio è seguito da una coppia di parentesi tonde che contengono una o (overridden) ed il numero del passo sovrascritto. Per esempio: 4.(o3) *Questo quarto passo figlio sovrascrive il terzo passo padre.*

Le cinque possibili opzioni sono riassunte di seguito:

Ingegneria del Software

La caratteristica è	esempio di numerazione
Ereditata senza modifiche	3. (3.)
Ereditata e rinumerata	6.2 (6.1)
Ereditata e sovrascritta	1. (o1.)
Ereditata, sovrascritta, e rinumerata	5.2 (o5.1)
Aggiunta	6.3

L'esempio precedente potrebbe essere reso più leggibile introducendo il caso d'uso TrovaProdotto, come generalizzazione dei casi d'uso TrovaLibro e TrovaCD.



La specifica del caso d'uso *TrovaProdotto* è già stata descritta precedente (ID: 3) ed è espressa ad un livello molto alto di astrazione. Di seguito riportiamo il caso d'uso *TrovaLibro*, ottenuto come specializzazione di *TrovaProdotto*:

Ingegneria del Software

Use case: TrovaLibro
ID: 6
Parent ID: 3
Brief description: <i>Il sistema trova alcuni libri che soddisfano i criteri di ricerca del cliente e li visualizza</i>
Primary actors: <i>Cliente</i>
Secondary actors: <i>Nessuno</i>
Preconditions: <i>Nessuna</i>
Main flow: 1. (o1.) Il caso d'uso inizia quando il cliente seleziona "find book". 2. (o2.) Il sistema chiede al cliente i criteri di ricerca, comprendenti autore, titolo, ISBN o argomento. 3. (3.) Il cliente inserisce i criteri di ricerca. 4. (o4.) Il sistema ricerca i libri che soddisfano i criteri di ricerca. 5. (o5.) IF il sistema trova dei libri 1. Il sistema visualizza il best seller corrente 2. (o5.1) Il sistema visualizza i dettagli di un massimo di 5 libri 3. FOR ogni libro sulla pagina 1. Il sistema visualizza il titolo, l'autore, il prezzo e lo ISBN 4. WHILE ci sono ancora libri 1. Il sistema dà al cliente l'opzione di visualizzare la prossima pagina di libri 2. visualizza il titolo, l'autore, il prezzo e lo ISBN 6. (6.) ELSE 1. Il sistema visualizza il best seller corrente 2. (o6.1) Il sistema comunica al cliente che nessun libro è stato trovato
Postconditions: <i>Nessuna</i>
Alternative flows: <i>Nessuno</i>

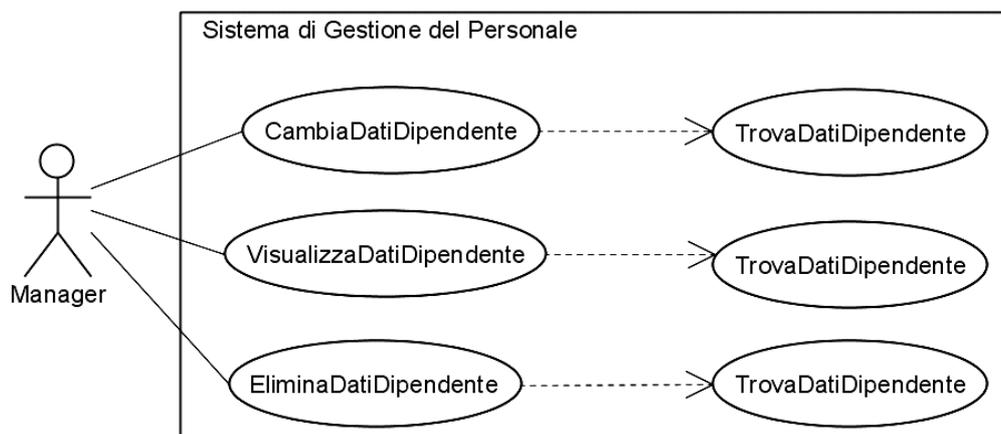
Ingegneria del Software

NOTA BENE: se il caso d'uso genitore non ha un flusso di eventi o il flusso di eventi è incompleto, allora è un caso d'**uso astratto**. In questo caso, il suo nome è scritto in corsivo.

Come si può notare dall'esempio precedente, con la notazione tipografica usata risulta difficile presentare caratteristiche ereditate e farle capire ad utenti non esperti di programmazione. Inoltre tale notazione va resa consistente manualmente ad ogni modifica (con possibilità di errore). Una soluzione a questo problema è di restringere il caso d'uso genitore alla breve descrizione della propria semantica, senza specificare un flusso.

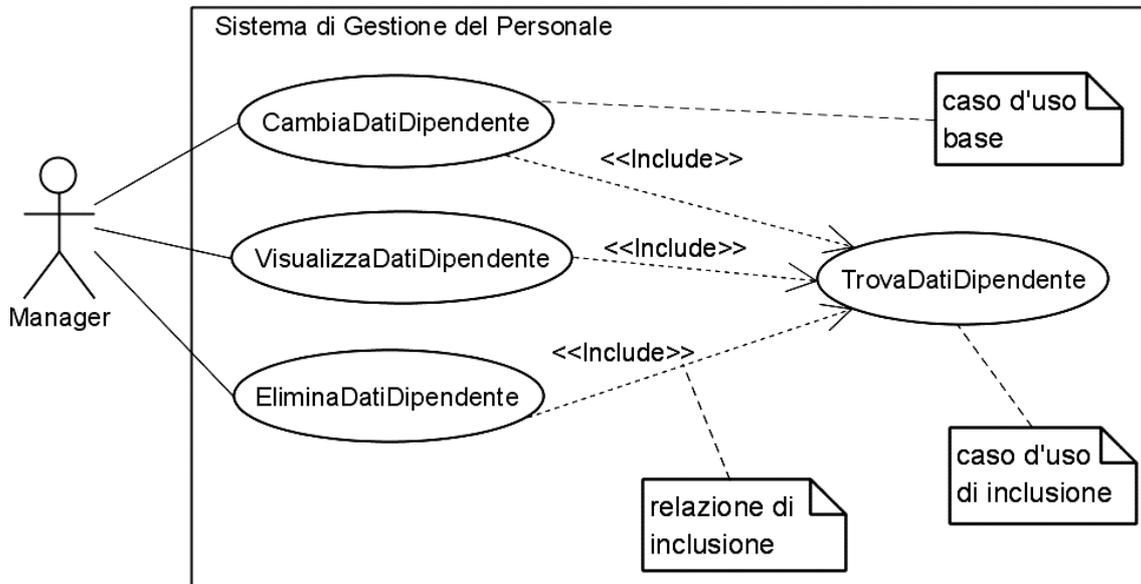
La relazione <<include>>

Supponiamo che si stia progettando un sistema di gestione del personale. Un diagramma di casi d'uso potrebbe essere il seguente.



Tutti e tre i casi d'uso richiedono di localizzare i dati di uno specifico dipendente. La relazione **<<include>>** evita di ripetere la sequenza di eventi ogni volta che servono i dati del dipendente.

La relazione **<<include>>** è uno stereotipo di relazione di dipendenza che permette di includere un caso d'uso (detto "di inclusione") in altri casi d'uso (detti "base"), consentendo a questi di condividere un comune frammento di comportamento senza ripeterlo più volte.



Nei flussi degli eventi dei casi d'uso base va specificato il punto esatto dove è richiesto il comportamento del caso d'uso di inclusione. La sintassi è simile ad una chiamata di funzione.

Ingegneria del Software

Use case: <i>CambiaDatiDipendente</i>
ID: 1
Brief description: <i>Il manager cambia i dati del dipendente</i>
Primary actors: <i>Manager</i>
Secondary actors: <i>Nessuno</i>
Preconditions: <i>Il Manager è connesso al sistema</i>
Main flow: 1. include(TrovaDatiDipendente) 2. <i>Il sistema visualizza i dati del dipendente</i> 3. <i>Il Manager cambia i dati del dipendente</i>
Postconditions: 1. <i>I dettagli del dipendente sono stati cambiati</i>
Alternative flows: <i>Nessuno</i>

Use case: <i>VisualizzaDatiDipendente</i>
ID: 2
Brief description: <i>Il Manager visualizza i dati del dipendente</i>
Primary actors: <i>Manager</i>
Secondary actors: <i>Nessuno</i>
Preconditions: <i>Il Manager è connesso al sistema</i>
Main flow: 1. include(TrovaDatiDipendente) 2. <i>Il sistema visualizza i dati del dipendente</i>
Postconditions: 1. <i>Il sistema ha visualizzato i dati del dipendente</i>
Alternative flows: <i>Nessuno</i>

Ingegneria del Software

Use case: <i>CancellaDatiDipendente</i>
ID: 3
Brief description: <i>Il manager cancella i dati del dipendente</i>
Primary actors: <i>Manager</i>
Secondary actors: <i>Nessuno</i>
Preconditions: <i>Il Manager è connesso al sistema</i>
Main flow: 1. <i>include(TrovaDatiDipendente)</i> 2. <i>il sistema visualizza i dati del dipendente</i> 3. <i>il Manager cancella i dati del dipendente</i>
Postconditions: 1. <i>I dettagli del dipendente sono stati cancellati</i>
Alternative flows: <i>Nessuno</i>

Use case: <i>TrovaDatiDipendente</i>
ID: 4
Brief description: <i>Il Manager trova i dati del dipendente</i>
Primary actors: <i>Manager</i>
Secondary actors: <i>Nessuno</i>
Preconditions: <i>Il Manager è connesso al sistema</i>
Main flow: 1. <i>Il Manager inserisce l'ID del dipendente</i> 2. <i>Il sistema trova i dettagli del dipendente</i>
Postconditions: 1. <i>Il sistema ha trovato i dati del dipendente</i>
Alternative flows: <i>Nessuno</i>

Il caso d'uso base viene eseguito fino al punto di inclusione, dove il controllo dell'esecuzione passa al caso d'uso di inclusione. Quando quest'ultimo termina, il controllo ritorna al caso d'uso base.

Ingegneria del
Software

I casi d'uso base non sono sicuramente completi senza tutti i rispettivi casi d'uso di inclusione. Anche i casi d'uso di inclusione potrebbero non essere completi, se i flussi di eventi che li definiscono hanno senso solo se inclusi in opportuni casi d'uso base.

In tal caso il caso d'uso di inclusione **non è istanziabile**, ossia non può essere avviato direttamente da un attore. Se comunque il caso d'uso di inclusione è completo può essere trattato come un normale caso d'uso e quindi può essere istanziato se necessario.

Il caso d'uso *TrovaDatiDipendente* non è completo e quindi non istanziabile.

La relazione <<extend>>

La relazione «**extend**» permette di inserire un nuovo comportamento in un caso d'uso esistente. È anch'essa una relazione di dipendenza stereotipata ma, sebbene possa essere pensata come una invocazione di funzione similmente alla inclusione, presenta differenze semantiche rispetto a quest'ultima. Innanzitutto, il caso d'uso base è completo anche senza estensioni.

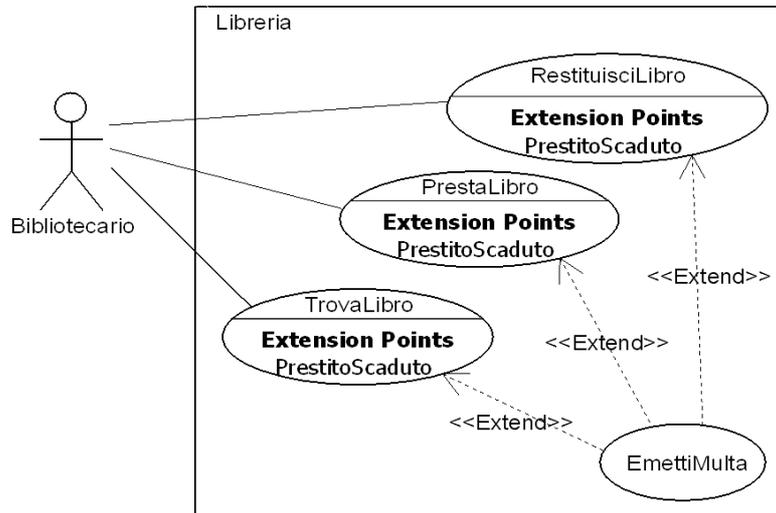
Il caso d'uso base stabilisce opportuni **punti di estensione** nel proprio flusso dove un nuovo comportamento può essere aggiunto ed il caso d'uso di estensione fornisce un insieme di segmenti che possono essere inseriti in questi punti per implementare questo nuovo comportamento.

Ingegneria del
Software

I punti di estensione non sono effettivamente inseriti nel flusso degli eventi del caso d'uso base, ma piuttosto sono aggiunti ad un livello superiore al flusso di eventi. In pratica, è come se i punti di estensione fossero scritti su un foglio trasparente che viene posto sopra il flusso degli eventi: in questo modo il flusso è indipendente dai punti di estensione.

La relazione di estensione fornisce un buon modo per trattare con casi eccezionali che non possono essere predetti in anticipo.

Un esempio di diagramma dei casi d'uso con la relazione di estensione è il seguente:



Ingegneria del Software

Di seguito, vengono descritti il caso d'uso base RestituisciLibro e il caso d'uso di estensione EmettiMulta.

Use case: <i>RestituisciLibro</i>
ID: 9
Brief description: <i>Il bibliotecario restituisce un libro prestato</i>
Primary actors: <i>Bibliotecario</i>
Secondary actors: <i>Nessuno</i>
Preconditions: <i>Il Bibliotecario è connesso al sistema</i>
Main flow: <ol style="list-style-type: none"> <i>il Bibliotecario inserisce l'identificatore della persona che restituisce il libro</i> <i>Il sistema visualizza i dati della persona con la lista dei libri che ha in prestito</i> <i>Il Bibliotecario trova il libro nella lista dei libri</i> extension point: <i>prestitoScaduto</i> <ol style="list-style-type: none"> <i>Il Bibliotecario restituisce il libro.</i>
Postconditions: <ol style="list-style-type: none"> <i>Il libro è stato restituito</i>
Alternative flows: <i>Nessuno</i>

Extension Use case: <i>EmettiMulta</i>
ID: 10
Brief description: Segment 1: <i>Il bibliotecario memorizza e stampa la multa</i>
Primary actors: <i>Bibliotecario</i>
Secondary actors: <i>Nessuno</i>
Segment 1 Preconditions: <i>Il prestito del libro restituito è scaduto</i>
Segment 1 flow: <ol style="list-style-type: none"> <i>Il Bibliotecario inserisce i dettagli della multa nel sistema.</i> <i>il sistema stampa la multa</i>
Segment 1 Postconditions: <ol style="list-style-type: none"> <i>La multa è stata memorizzata nel sistema</i> <i>il sistema ha stampato la multa</i>
Alternative flows: <i>Nessuno</i>

Ingegneria del Software

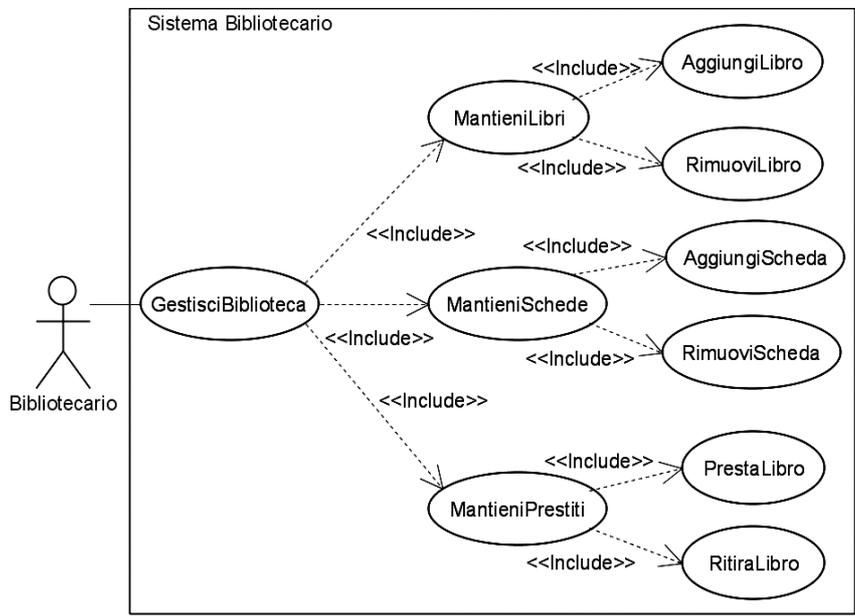
Generalmente il caso d'uso di estensione non è completo. Tipicamente consiste di uno o più frammenti di comportamento conosciuti come **segmenti di inserzione**.

Vengono applicate le regole seguenti:

- La relazione «**extend**» deve specificare uno o più punti di estensione oppure si assume che la relazione si riferisce a tutti i punti di estensione.
- Il caso d'uso di estensione deve avere lo stesso numero di segmenti di inserzione quanti sono i punti di estensione specificati nella relazione «**extend**».
- È legale che due casi d'uso di estensione estendano lo stesso caso d'uso base allo stesso punto di estensione. Se questo accade, comunque, l'ordine di esecuzione delle estensioni è indeterminato.

Nell'esempio precedente c'è un segmento di inserzione unico nel caso d'uso di estensione. Il caso d'uso di estensione può avere precondizioni e postcondizioni. Le precondizioni devono essere soddisfatte, altrimenti il segmento non viene eseguito. Le postcondizioni vincolano lo stato del sistema dopo che il segmento è stato eseguito.

Ricorda: il caso d'uso esprime **cosa** il sistema deve fare per gli attori non come deve farlo (che viene espresso nel progetto). Un altro errore comune è la “**decomposizione funzionale**” di casi d'uso a vari livelli, mediante tante relazioni di inclusione che in realtà dettagliano l'implementazione dei casi d'uso base. In questo modo, il modello descrive il sistema come un insieme di funzioni annidate ed è difficile da capire per persone coinvolte nel progetto, ma non esperte di informatica. **I livelli di inclusione non dovrebbero mai essere più di 2** e l'intero modello non dovrebbe mai avere un solo caso d'uso base.



Ingegneria del Software

Mario G.C.A. Cimino - Antonio Luca Alfeo



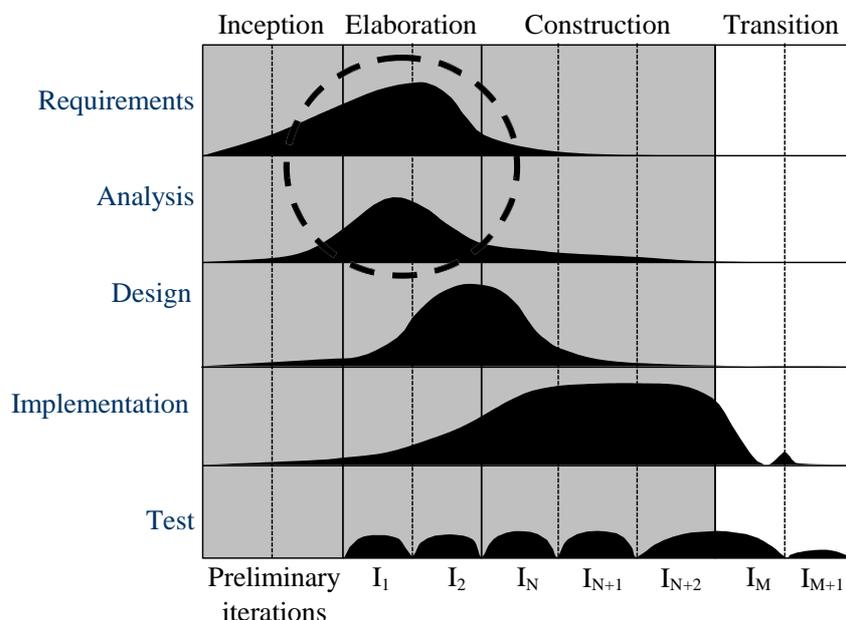
Workflow Analisi

Introduzione

L'obiettivo del workflow Analisi è di creare modelli che catturino il comportamento desiderato del sistema, cioè *che cosa* il sistema offrirà piuttosto che *come* lo farà (obiettivo del workflow Progetto).

La maggior parte dell'attività richiesta da questo workflow è realizzata durante le fasi di Inception ed Elaboration.

Il workflow Analisi si svolge in stretta concomitanza con il workflow Requisiti (vedi figura seguente): spesso per chiarire i requisiti o per scoprirne di nuovi è necessaria un'attività di analisi.



Gli artefatti prodotti dal workflow Analisi sono:

- **Classi di analisi** - modellano i concetti chiave del dominio di business;
- **Realizzazioni dei casi d'uso** – illustrano come le istanze delle classi di analisi possono interagire per realizzare il comportamento del sistema specificato da un caso d'uso.

Modello di analisi - Regole Empiriche

- Individua solo le classi che sono parte del vocabolario del dominio del problema. Evita di inserire classi derivanti dalla tecnologia, relative a problemi di comunicazione o di accesso a database.
- Descrivi il modello di analisi sempre e solo nel linguaggio del dominio.
- Crea modelli che raccontino una storia cioè descrivano parti importanti del comportamento desiderato del sistema. Concentrati nel catturare il quadro generale senza perderti nei dettagli di come il sistema lavorerà.
- Distingui sempre tra il dominio del problema ed il dominio delle soluzioni. Se si sta modellando un sistema di e-commerce, ci si aspetta di vedere classi come Cliente, Ordine, CarrelloDellaSpesa, non AccessoDatabase.
- Cerca sempre di minimizzare le dipendenze tra le classi (accoppiamento). Adopera l'ereditarietà, che è la forma più forte di accoppiamento tra classi, solo se la gerarchia di astrazioni appare naturale, non (ad esempio) per il riuso del codice.
- Mantieni il modello semplice. Pensa sempre al caso generale piuttosto che allo specifico. Ad esempio, i modelli relativi alla vendita di abitazioni, automobili e viaggi non sono modelli separati, ma piuttosto sono specializzazioni del modello di un generico "sistema di vendita".

Oggetti e Classi

UML Reference Manual: “Un oggetto è un’entità discreta con un confine ben definito che incapsula stato e comportamento; un’istanza di una classe”.

Generalmente, il solo modo per accedere ai dati di un oggetto è attraverso le funzioni che l’oggetto mette a disposizione: tali funzioni sono chiamate *operazioni*.

Ogni oggetto è istanza di una classe che definisce l’insieme di caratteristiche (attributi ed operazioni) dell’oggetto. Ad esempio, la EPSON ha prodotto migliaia di stampanti “Epson Photo 1200”. L’oggetto “*Epson Photo 1200 S/N 34120098*” appartiene alla classe “*Epson Photo 1200*”.

Proprietà comuni a tutti gli oggetti:

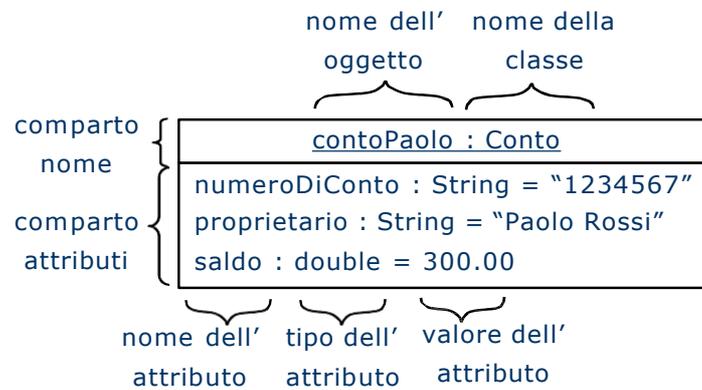
- **Identità**: individua univocamente l’oggetto nel tempo e nello spazio. Per esempio, il numero di serie *34120098* della stampante può essere utilizzato per rappresentare l’identità (**riferimento** univoco nello spazio e nel tempo) dell’oggetto.
- **Stato**: è determinato dai valori degli attributi di un oggetto e dalle relazioni che l’oggetto ha con altri oggetti ad un particolare istante.

- **Comportamento**: le operazioni che l’oggetto può compiere. Per quanto riguarda la stampante, per esempio, *accendi()*, *spegni()*, *stampa(documento)* ed *interrompi()*. L’invocazione di un’operazione su un oggetto può provocare un cambio di stato. Chiaramente lo stato dell’oggetto può influenzare il suo comportamento. Supponiamo che la stampante abbia finito l’inchiostro: l’invocazione dell’operazione *stampa(documento)* produrrebbe un segnale d’errore.

Un sistema OO è un insieme di oggetti interagenti (**scambio di messaggi**). Si può interagire con un oggetto (di cui è noto il riferimento) esclusivamente invocando le sue operazioni pubbliche ed accedendo ai suoi attributi pubblici (**incapsulamento**).

Notazione UML per gli oggetti

La notazione UML usata per identificare gli oggetti è la seguente



NOTA BENE:

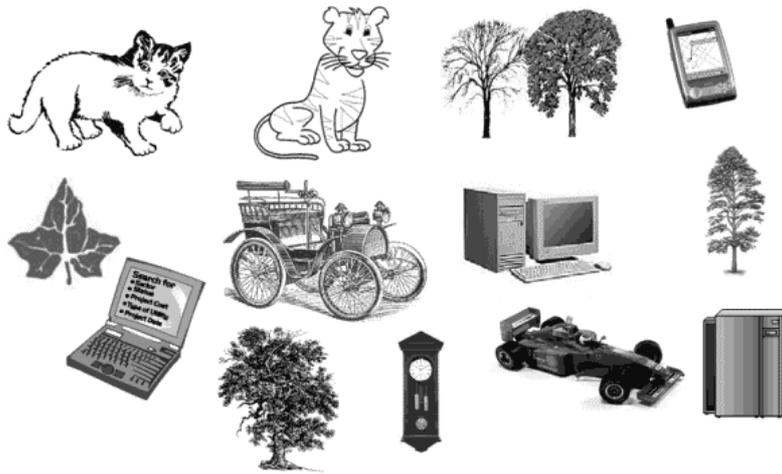
- L'**identificatore** dell'oggetto è sempre **sottolineato**.
- Il **nome dell'oggetto** può essere omissso (nell'esempio, l'identificatore diventerebbe :Conto). In questo caso, l'oggetto risulta anonimo. Gli oggetti anonimi sono utilizzati quando è usato nel diagramma solo un oggetto di una particolare classe.
- L'**identificatore** dell'oggetto può essere composto solo dal nome dell'oggetto (nell'esempio, contoPaolo). In questo modo si identifica uno specifico oggetto, ma senza individuare a quale classe appartiene. Può essere utile nelle fasi preliminari dell'analisi.

- La forma più utilizzata per l'identificatore è comunque quella rappresentata nella figura, cioè nome oggetto : nome classe.
- I nomi degli oggetti sono normalmente concatenazioni di parole scritte in *lowerCamelCase* (la prima parola inizia con una lettera minuscola e le altre con una lettera maiuscola).
- Dato che tutti gli oggetti di una stessa classe hanno le stesse operazioni, le operazioni non sono elencate nell'oggetto.
- Negli oggetti, possono essere mostrati alcuni o tutti gli attributi della classe. Gli attributi devono avere un nome e possono avere un tipo ed un valore. Il formato è il seguente: name : type = value.

Classi

UML Reference Manual: “una classe è un descrittore per un insieme di oggetti che condividono gli stessi attributi, operazioni, metodi, relazioni e comportamento”. Le classi devono descrivere le caratteristiche che ogni oggetto della classe *deve* avere senza descrivere ciascuno degli oggetti.

Osserviamo la figura seguente, quante classi si possono identificare?



In verità, non c'è una sola risposta a questa domanda. Ci sono molti modi di organizzare gli oggetti del mondo reale in categorie, e questi modi sono in numero sempre maggiore all'aumentare della varietà (numero di caratteristiche) di tali oggetti. Ad esempio, ad un primo sguardo possiamo individuare la classe dei gatti, degli alberi, delle foglie,

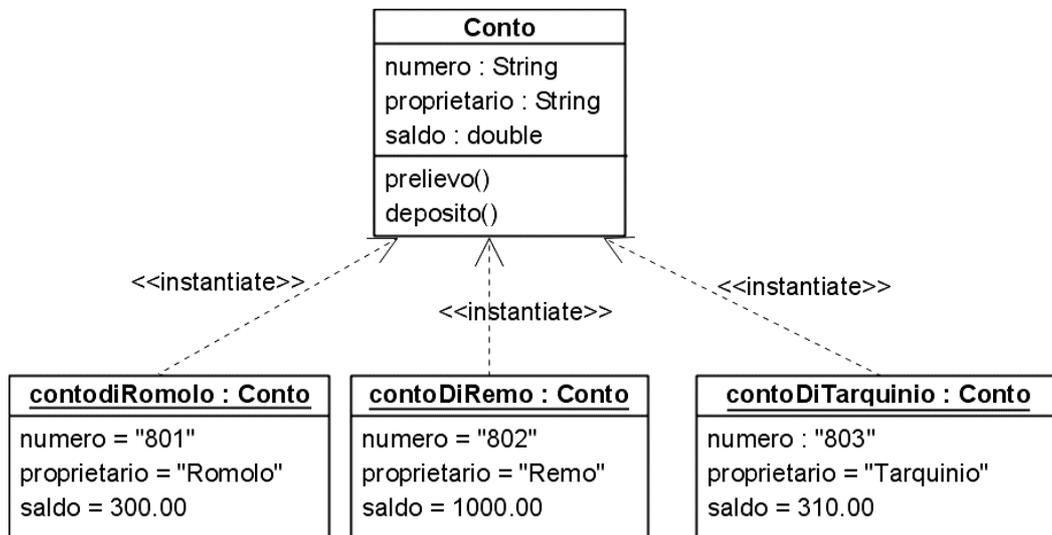
Il principale aspetto dell'analisi e progetto OO consiste proprio nello scegliere lo schema di classificazione più appropriato.

Un'analisi più dettagliata rivela che vi sono altri tipi di relazioni oltre alla istanziazione classe/oggetto. La classe *Gatto* è la generalizzazione delle classi *GattoSelvatico* e *GattoDomestico*. La classe *Foglia* ha una relazione di composizione con la classe *Albero*, poiché ogni foglia non può essere condivisa tra gli alberi e la vita di essa è interamente legata a quella dell'albero.

La classe *Periferica* ha una relazione di aggregazione con la classe *Computer*, poiché un monitor o un mouse possono essere condivisi tra computer ed hanno una vita autonoma da essi. Tuttavia questo non accade per le *PerifericheIntegrate* dei dispositivi mobili.

La relazione tra classi ed oggetti è una relazione di dipendenza stereotipata «**instatiate**». Lo *UML Reference Manual* definisce una relazione come “una connessione tra elementi del modello”.

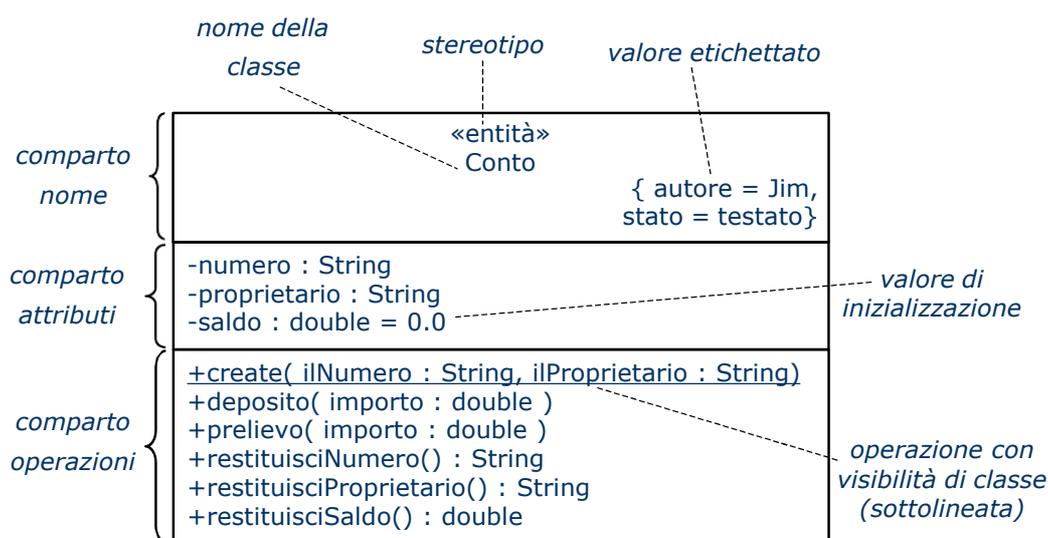
La relazione «**instatiate**» è presentata nella figura seguente:



La freccia tratteggiata indica una relazione di dipendenza a cui è stato dato un significato speciale attraverso lo stereotipo «**instantiate**». Lo *UML Reference Manual* definisce una dipendenza come “una relazione tra due elementi in cui una modifica apportata ad un elemento (il fornitore) può influenzare l’altro elemento (il cliente) o fornire ad esso informazione necessaria”. Nella figura la classe Conto è il fornitore perché determina la struttura degli altri elementi.

Notazione UML per le classi

La figura seguente mostra la sintassi visuale UML per una classe.



Solo il comparto *nome*, con il nome della classe inserito al suo interno, è obbligatorio. La scelta di quali comparti e quali ornamenti utilizzare nella notazione di classe dipende dallo scopo del diagramma.

Per l'analisi è sufficiente presentare il nome, gli attributi, le operazioni chiave e gli stereotipi (se hanno un significato per il dominio). Tipicamente, non vengono presentati i valori etichettati, i parametri dell'operazione, la visibilità e i valori di inizializzazione.

Comparto nome

- ✓ Generalmente i nomi vengono scritti nello stile *UpperCamelCase* (come *lowerCamelCase*, ma con il primo carattere maiuscolo). Dato che rappresentano “cose”, le classi dovrebbero avere nomi che sono sostantivi o sequenze di sostantivi.
- ✓ È buona norma dare dei nomi significativi, adoperando solo simboli alfanumerici senza spazi, per evitare che nella produzione automatica di codice Java/C++ o di documentazione HTML/XML vi siano conseguenze inaspettate a causa di simboli dal significato particolare.
- ✓ Evita abbreviazioni. Un nome come *DepositoContoCorrente* è sempre preferibile a *DpstCntCr*. Se comunque ci sono acronimi (per esempio, CRM – Customer Relationship Management) comunemente usati e conosciuti, usali pure.

Comparto attributo

- ✓ Gli attributi sono tipicamente nominati in *lowerCamelCase*. Sono anche essi generalmente sostantivi o sequenze di sostantivi.

- ✓ Nella fase di analisi generalmente la visibilità viene omessa (siamo interessati al *come* non al *cosa*). La tabella seguente riassume gli ornamenti, ed il loro significato, usati per indicare la visibilità in UML.

Ornamento	Nome della visibilità	Semantica
+	Public	Ogni elemento che può accedere alla classe può accedere alle caratteristiche con questa visibilità
-	Private	Solo gli operatori definiti nella classe possono accedere alle caratteristiche con questa visibilità
#	Protected	Solo operatori all'interno della classe, o all'interno di figli della classe possono accedere alle caratteristiche con questa visibilità
~	Package	Ogni elemento che è nello stesso package della classe o in subpackage annidati può accedere alle caratteristiche con questa visibilità

- ✓ Il **tipo di un attributo** può essere un'altra classe o un tipo primitivo. UML definisce quattro tipi primitivi: *Integer*, *UnlimitedNatural* (l'infinito è rappresentato con *), *Boolean* e *String*. L'OCL definisce anche il tipo *Real*. Sebbene sia possibile utilizzare i tipi

predefiniti di uno specifico linguaggio, cerca di evitarlo perché altrimenti il modello diventerebbe fortemente legato a quel linguaggio.

- ✓ Possono essere aggiunti ulteriori tipi primitivi creando una classe con lo stesso nome del tipo primitivo e lo stereotipo «**primitive**». La classe non ha né attributi né operazioni.
- ✓ La **molteplicità** permette di modellare collezioni di elementi o valori nulli. Se la molteplicità è più grande di 1, allora si sta specificando una collezione. Per esempio, `address: String[3]` indica un attributo che è una collezione di tre elementi di tipo `String`. Quando un attributo ha valore nullo (“null”) significa che l’oggetto non è stato ancora creato o ha cessato di esistere. Il valore “null” non deve essere confuso con la stringa vuota “”. Per esempio, l’attributo `emailAddress : String[0..1]` può avere un valore o avere un valore nullo. Nel caso il valore sia la stringa vuota “” significa che l’indirizzo è stato chiesto e non è stato ancora fornito. Nel caso il valore sia “null” significa che il valore non è stato ancora chiesto.
- ✓ **ATTENZIONE:** La molteplicità è poco usata in fase di analisi, essendo più un aspetto progettuale.
- ✓ Il **valore iniziale** permette di specificare il valore che un attributo prenderà quando un oggetto viene istanziato. I valori iniziali vengono usati in analisi solo se possono evidenziare vincoli importanti sul dominio.

- ✓ La specifica di un attributo può essere estesa attraverso i **valori etichettati**. Ad esempio:
indirizzo { inseritoDa = "A.Turing", dataInserimento = "20-ott-2005" }
logaritmo(x: **Real**) : **Real** { base = 2 }

Comparto operazione

La combinazione del nome, i tipi dei parametri ed il tipo dell’oggetto restituito costituiscono l’intestazione dell’operazione. **ATTENZIONE:** questa definizione è diversa dal C++.

Il formato generico di un’operazione è:

visibility name(direction parameterName: parameterType = defaultValue,..) : returnType

I nomi delle operazioni ed i nomi dei parametri sono normalmente scritti in *lowerCamelCase*.

La direzione è espressa come:

in – parametro in ingresso

inout – parametro ingresso/uscita

out – parametro uscita

Se la direzione è omessa, di default è **in**.

Alcuni esempi di operazioni vengono dati di seguito:

valoreMax(in a: Integer, in b: Integer): Integer	due parametri di ingresso, uno restituito
aggiungiSpesa(inout t: Importo)	un parametro di ingresso che viene poi letto come risultato
restituisciData(out t: Data)	un parametro di sola uscita
disegnaCerchio(centro: Punto = Punto(0,0), raggio: Real)	parametri di ingresso di cui uno inizializzato (default)

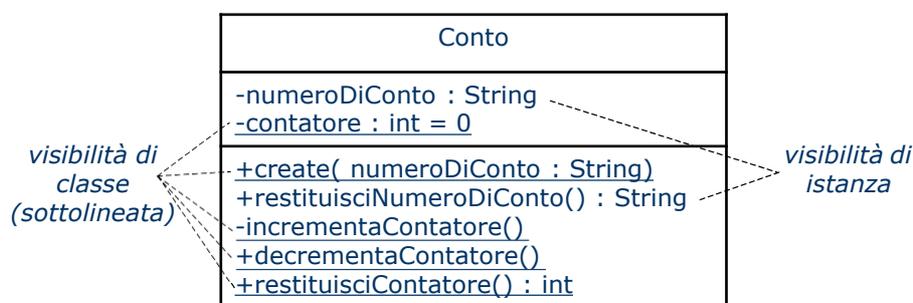
In generale, la direzione dei parametri è un problema di progetto e quindi tipicamente non ce se ne preoccupa in analisi.

I valori di default vengono usati quando l'operazione è invocata senza specificare i corrispondenti parametri attuali.

Visibilità

Tipicamente, gli attributi e le operazioni hanno visibilità di istanza, cioè ogni oggetto ha i propri valori degli attributi e le operazioni modificano questi valori.

A volte può essere necessario definire attributi che hanno un valore unico e condiviso tra le varie istanze, ed operazioni che si riferiscono a classi e non alle singole istanze (l'operazione di creazione di oggetti). In questo caso, si parla di attributi ed operazioni con visibilità di classe.



Gli attributi e le operazioni con visibilità di classe vengono sottolineati.

ATTENZIONE: operazioni con visibilità d'istanza possono accedere agli attributi ed alle operazioni con visibilità d'istanza ed anche a tutte le operazioni e agli attributi con visibilità di classe.

ATTENZIONE: operazioni con visibilità di classe possono accedere solo agli attributi ed alle operazioni con visibilità di classe. Operazioni con visibilità di classe, quindi, non possono accedere alle operazioni con visibilità di istanza perché non si saprebbe su quale istanza invocare l'operazione.

I costruttori sono operazioni speciali che creano nuove istanze di classi. Linguaggi diversi usano notazioni diverse per nominare i costruttori. Un approccio generico è quello di chiamare i costruttori `create()` ed elencare i parametri necessari. Durante l'analisi, tipicamente, non serve specificare i costruttori. Nel progetto, invece, andranno specificati completamente insieme all'eventuale distruttore.

SocioDelCircolo
<code>-numeroAppartenenza : String</code> <code>-nomeSocio : String</code> <code>-numeroSoci : int = 0</code>
<code>+create(numero : String, nome : String)</code> <code>+restituisceNumeroAppartenenza() : String</code> <code>+restituisceNomeSocio() : String</code> <code>-incrementaNumeroSoci()</code> <code>+decrementaNumeroSoci()</code> <code>+restituisceNumeroSoci() : int</code>

Trovare le classi di analisi

Cosa dovrebbe essere una classe di analisi?

1. È una classe che rappresenta un'astrazione ben definita nel dominio del problema;
2. è una classe che descrive concetti del dominio di applicazione nel mondo reale. Per esempio, Cliente, Prodotto, ContoCorrente, etc..

Le classi di analisi dovrebbero presentare un insieme di attributi, che le risultanti classi di progetto probabilmente avranno. Potremmo dire che le classi di analisi catturano attributi candidati per le classi di progetto.

Le operazioni nelle classi di analisi specificano i servizi fondamentali che la classe deve offrire. Spesso, un'operazione individuata a livello di analisi sarà suddivisa in più operazioni a livello di progetto.

Tipicamente, rispetto alla generica notazione di classe introdotta nei lucidi precedenti, la classe di analisi avrà:

- ✓ Il nome
- ✓ I nomi degli attributi (in genere vengono elencati quelli ritenuti più importanti)
- ✓ I nomi delle operazioni – i tipi dei parametri e degli oggetti restituiti sono presentati solo dove sono importanti per capire il modello
- ✓ Visibilità – non è generalmente mostrata
- ✓ Stereotipi – sono presentati solo se chiariscono maggiormente il modello
- ✓ Valori etichettati – possono essere presentati se chiariscono maggiormente il modello

La figura seguente è un esempio di una classe di analisi:



Cosa rende una classe di analisi una buona classe?

- Il suo nome riflette il suo intento. Consideriamo un sistema per il commercio elettronico. *Cliente* sembrerebbe riferire qualcosa di molto preciso del mondo reale e quindi è un buon candidato per una classe. Stessa cosa per *CarrelloDellaSpesa* (*ShoppingBasket*).

Una classe *VisitatoreSitoWeb* invece sembra avere una semantica vaga. Infatti, visitatore di un sito web sembra più un ruolo interpretato dal *Cliente* che una classe con la sua semantica.

- È un'astrazione ben definita che modella uno specifico elemento del dominio del problema ed ha una semantica chiara ed ovvia.
- Ha un insieme di responsabilità piccolo e ben definito. Ad esempio, la classe *CarrelloDellaSpesa* ci si aspetta che abbia le responsabilità “aggiungi articolo al carrello”, “rimuovi articolo dal carrello”, “visualizza gli articoli nel carrello”. Questo è un insieme coesivo di responsabilità, perché tutte le responsabilità lavorano verso lo stesso goal – gestire il carrello della spesa.
- Ha un'alta coesione. Nell'esempio della classe *CarrelloDellaSpesa* se aggiungessimo responsabilità come “valida la carta di credito” o “accetta pagamento” alla classe, perderemmo sicuramente la coesione, perché le responsabilità a questo punto si riferirebbero ad obiettivi diversi. Queste responsabilità sembrerebbero appartenere più a classi come *CompagniaCartaDiCredito* o *ControlloInUscita*.
- Ha un basso accoppiamento. L'accoppiamento è misurato come il numero di altre classi con cui una data classe ha relazioni. Una buona distribuzione delle responsabilità tra classi porterà ad un basso accoppiamento. La localizzazione del controllo o di molte

responsabilità in una singola classe tende ad incrementare l'accoppiamento di quella classe.

Regole empiriche per le classi di analisi

- Mantieni le classi semplici: attribuisce ad ogni classe un numero limitato di responsabilità (da 3 a 5, normalmente)
- Diffida delle classi onnipotenti: classi come *Sistema* o *Controllore* tenderanno ad essere sovraccaricate di responsabilità. Controlla se le responsabilità attribuite a queste classi possano essere organizzate in sottoinsiemi coesivi. È probabile che questi sottoinsiemi possano essere trasformati in classi separate.
- Evita classi *solitarie*: in una buona analisi object-oriented le classi collaborano per fornire servizi agli utenti.
- Evita molte classi con poche responsabilità: se il modello ha molte classi con una o due responsabilità diventa difficile da capire e seguire. Cerca di compattare classi affini.
- Evita “funzioidi”. Una funzioida è una normale procedura mascherata da classe.

- Evita alberi di ereditarietà profondi: ogni livello di astrazione nella gerarchia dovrebbe avere uno scopo ben definito. Evita di usare l'ereditarietà per realizzare una sorta di decomposizione funzionale. In analisi, un albero può essere considerato profondo se ha più di due livelli di astrazione. Nel progetto, il giudizio sulla profondità dipende dal linguaggio che utilizzeremo per la codifica.

Approcci per trovare le classi di analisi

Sfortunatamente, NON C'È NESSUN ALGORITMO PER TROVARE LE CLASSI DI ANALISI GIUSTE. Spesso la scelta delle classi giuste dipende dall'esperienza e dalle capacità dell'analista.

Trovare le classi usando l'analisi dei nomi e dei verbi

I nomi ed i verbi nel testo indicano rispettivamente classi e responsabilità. Questa analisi si basa sull'esame della descrizione del dominio del problema.

ATTENZIONE: tieni in considerazione sinonimi e omonimi - possono portare a classi spurie.

ATTENZIONE: stai attento che il dominio del problema non sia definito e capito male.

ATTENZIONE: cerca di individuare le classi "nascoste". Queste classi sono intrinseche al problema e quindi potrebbero non essere mai nominate esplicitamente. Per esempio, in un sistema di prenotazione di vacanze, nel dominio del problema le persone coinvolte parleranno di prenotazione, acquisto, etc., e potrebbero non menzionare mai l'astrazione più ovvia: Ordine. L'individuazione di una classe nascosta darà forma all'intero modello.

Procedura

1. Raccogli più informazione possibile. Le sorgenti di informazioni più importanti sono:
 - a. Il modello dei requisiti;
 - b. Il modello dei casi d'uso;
 - c. Il glossario
 - d. Ogni altra informazione utile (architettura, documenti iniziali, etc.)
2. Analizza l'informazione individuando:
 - a. Nomi – per esempio, “volo”;
 - b. Frasi composte da nomi – per esempio, “numero del volo”;

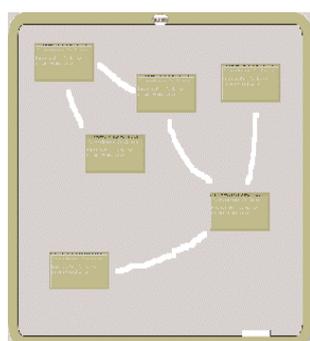
c. Verbi – per esempio, “prenota”;

d. Frasi con verbi – per esempio, “verifica la carta di credito”;

3. Se qualche termine non è chiaro, chiedi subito chiarimenti all’esperto del dominio ed aggiungi il termine al glossario;
4. Confronta i nomi, le frasi di nomi, i verbi e le frasi di verbi con i termini nel glossario in modo da rimuovere sinonimi e omonimi.
5. Una volta definita la lista di classi candidate, attributi e responsabilità, fai un’allocazione degli attributi e responsabilità alle classi candidate. Se hai qualche idea di associazione, inseriscila come associazione candidata.

Trovare le classi usando l’analisi CRC

CRC – Classe, Responsabilità, Collaboratore. Si possono usare foglietti adesivi o strumenti CASE. Un esempio è fornito nella figura seguente:



a)

Nome della classe: ContoBancario	
Responsabilità: Mantenere il saldo	Collaboratori: Banca

b)

ContoBancario	
Super Classes :	
Sub Classes :	
Description :	
Attributes :	
Name	Description
saldo	
Responsibilities :	
Name	Collaborator
mantenere il saldo	Banca

c)

a) Lavagna per analisi CRC – b) Esempio di foglietto adesivo CRC – c) Esempio in Visual Paradigm

I collaboratori sono altre classi che possono collaborare con la classe per realizzare qualche parte di funzionalità del sistema. L’analisi CRC dovrebbe essere sempre usata in congiunzione con l’analisi dei nomi e dei verbi, a meno che il sistema sia molto semplice.

Trovare le classi usando gli stereotipi di RUP

La tecnica si basa sul considerare tre differenti tipi di classi di analisi, individuate attraverso gli stereotipi: classi di confine («**boundary**»), classi di controllo («**control**») e entità («**entity**»).

1. Le classi di confine («**boundary**») mediano l'interazione tra il soggetto (sistema) ed attori esterni (ambiente). Possono essere:
 - i. classi di interfaccia con gli utenti (sistema-utente);
 - ii. di interfaccia con altri sistemi (sistema-sistema);
 - iii. di interfaccia con dispositivi (sistema-sensore/attuatore).

Considera che ogni comunicazione tra un attore ed un caso d'uso deve essere gestita da qualche oggetto nel sistema. Questi oggetti sono istanze di classi di confine. Se gli attori serviti da una classe di confine sono di tipo diverso (umani, sistemi o dispositivi), probabilmente l'analisi deve essere rivista.

Non usare troppi dettagli. Nella fase di analisi basta catturare l'esistenza di una classe che media tra il sistema e l'attore, ma non come esattamente esegue tale mediazione.

2. Le classi di controllo («**control**») sono controllori le cui istanze coordinano nel sistema il comportamento che corrisponde ad uno o più casi d'uso.

Le classi di controllo vengono trovate analizzando il comportamento descritto dai casi d'uso e pianificando come quel comportamento dovrebbe essere suddiviso tra le classi di analisi. Se il comportamento è semplice, questo può essere distribuito tra le classi di confine o le entità; altrimenti, dovrà essere introdotta una classe di controllo. Per esempio, in un sistema di processazione degli ordini, sembra adeguato inserire una classe GestioneOrdini.

ATTENZIONE: non inserire le classi di controllo artificialmente - esse devono nascere naturalmente dal dominio del problema.

ATTENZIONE: spesso le classi di controllo tendono ad essere suddivise attraverso diversi casi d'uso.

ATTENZIONE: se la classe di controllo ha un comportamento molto complesso, separa la classe in classi più semplici. Per esempio, la classe di controllo ControlloreRegistrazioneCorso potrebbe essere decomposta in Registrazione, GestioneCorso, GestionePersonale.

3. Le classi entità («**entity**») modellano le “cose” gestite dal sistema ed hanno un comportamento molto semplice. Classi che rappresentano informazioni persistenti, come Indirizzo o Persona, sono esempi di classi entità. Le classi entità:

- a. Sono coinvolte in molti casi d’uso;
- b. Sono gestite dalle classi di controllo;
- c. Forniscono informazioni alle classi di confine ed ottengono informazioni da loro;
- d. Rappresentano cose reali elaborate dal sistema;
- e. Sono spesso persistenti.

Trovare le classi usando altre sorgenti

Oltre all’analisi dei nomi e dei verbi, l’analisi CRC e gli stereotipi RUP ci sono altre sorgenti per poter trovare le classi di analisi:

- Oggetti fisici;
 - Documenti di ufficio (fatture, ordini, etc.).
-
- Interfacce verso il mondo esterno (schermi, tastiera, periferiche, etc.);
 - Entità concettuali: cose che sono cruciali al dominio, ma non sono concrete. Per esempio, ProgrammaFedeltà per un sistema di commercio elettronico.
 - Confronto con modelli del dominio (archetype pattern) già definiti nella letteratura. I modelli possono essere riusati o leggermente adattati al dominio del problema. Questa tecnica va sotto il nome di **modellazione basata sui componenti**. Per esempio, nel libro *Enterprise Patterns and MDA* viene data una lista di archetype pattern per il business: *CustomerRelationshipManagement, Inventory, Money, Order, Party, PartyRelationship, Product, Quantity, Rule*.

Il primo modello di classi di analisi

Per produrre il primo modello di classi di analisi, le uscite delle varie tecniche descritte precedentemente vanno consolidate.

- Confronta tutte le sorgenti delle classi;
- Consolida le classi di analisi, gli attributi e le responsabilità ed inseriscile in uno strumento di modellazione:
 - Usa il glossario per risolvere sinonimi e omonimi;
 - Evidenzia le differenze nei risultati delle tre tecniche;
- Controlla che le collaborazioni rappresentino relazioni tra classi;
- Migliora i nomi delle classi, degli attributi e delle responsabilità.

L'uscita prodotta da questa attività è il primo modello di analisi: un insieme di classi di analisi dove ogni classe può avere degli attributi e alcune responsabilità (da 3 a 5).

Relazioni tra classi

Le relazioni sono connessioni semantiche significative tra gli elementi del modello: sono il modo UML di connettere le entità.

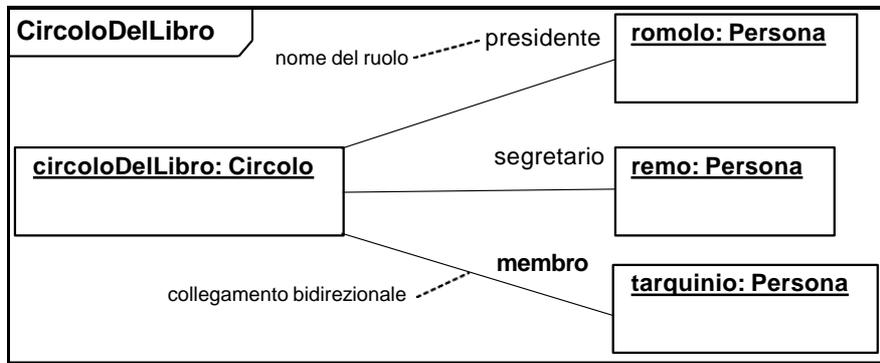
Le relazioni tra oggetti sono dette **collegamenti (link)** e quando gli oggetti lavorano insieme, diciamo che essi **collaborano**.

Le relazioni tra classi sono conosciute come **associazioni**. I collegamenti tra oggetti sono istanze di associazioni tra classi.

Che cosa è un collegamento?

Un collegamento è una connessione semantica tra due oggetti che permette di inviare messaggi tra un oggetto e l'altro. Il linguaggio C++ può implementare i collegamenti come puntatori, riferimenti o inclusione di un oggetto nell'altro. Il linguaggio Java implementa i collegamenti come riferimenti.

Un **DIAGRAMMA AD OGGETTI** è un diagramma che presenta, ad uno specifico istante nel tempo, gli oggetti e le loro relazioni. Oggetti connessi da un collegamento possono interpretare diversi ruoli reciprocamente. La figura seguente è un esempio di diagramma ad oggetti.



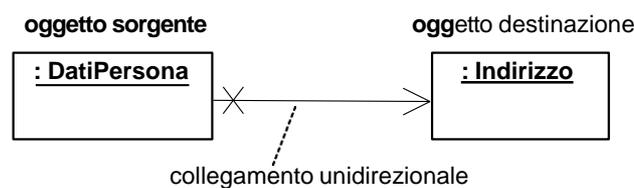
ATTENZIONE: i ruoli possono essere inseriti in entrambi gli estremi del collegamento. Nella figura, l'oggetto circoloDelLibro interpreta sempre il ruolo di circolo e quindi è stato omesso il suo ruolo.

ATTENZIONE: i collegamenti sono connessioni **dinamiche** tra oggetti e quindi non sono fissi nel tempo. Nell'esempio, il ruolo di presidente potrebbe passare da romolo a remo.

ATTENZIONE: per esserci un collegamento tra oggetti ci deve essere un'associazione tra le rispettive classi.

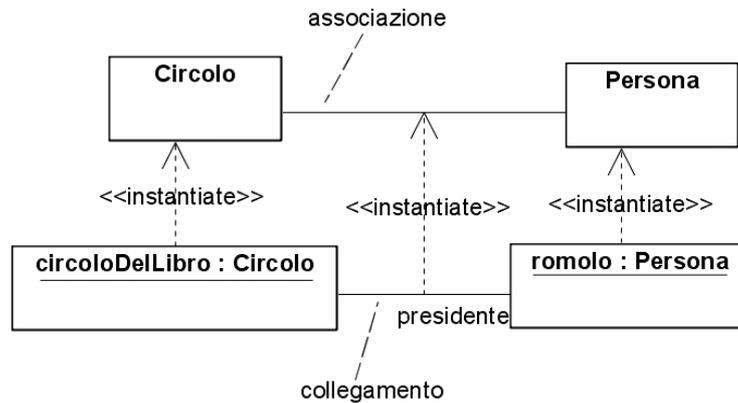
Nella figura i collegamenti sono bidirezionali. UML permette di specificare anche collegamenti unidirezionali attraverso l'uso di una freccia ad un estremo del collegamento

(navigabile) ed una croce all'altro (non navigabile). I messaggi possono solo fluire verso la freccia. La croce può essere omessa. Un esempio di collegamento unidirezionale navigabile è dato di seguito.



Che cosa è un'associazione?

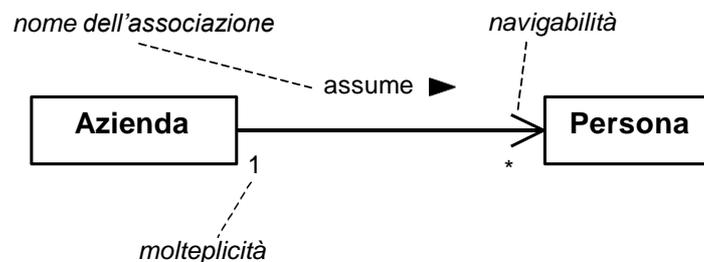
Un'associazione è una relazione tra classi. Dalla figura seguente, è chiaro che un collegamento *dipende* da un'associazione.



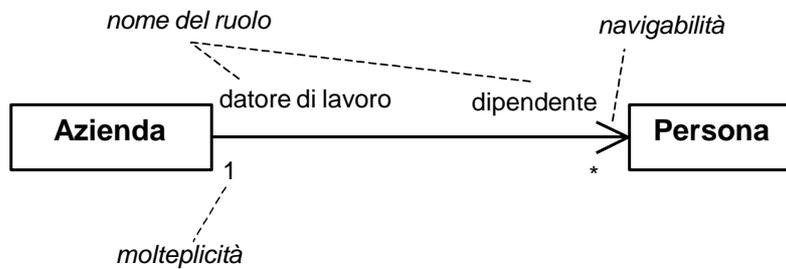
Sintassi dell'associazione

Le associazioni possono avere:

- **Un nome** – dovrebbe essere una frase con verbo perché indica un'azione che l'oggetto sorgente esegue sull'oggetto target. I nomi sono scritti in lowerCamelCase. Una freccia prefissa o postfissa al nome indica la direzione in cui il nome dovrebbe essere letto.



- **I nomi dei ruoli** – in alternativa al nome della relazione possono essere inseriti i nomi dei ruoli in uno o entrambi gli estremi. ATTENZIONE: UML permette di inserire sia i nomi che i ruoli, ma questo appesantisce la notazione senza dare informazione utile. I nomi sono tipicamente frasi di sostantivi.



- **La molteplicità**- vincola il numero degli oggetti di una classe che possono essere coinvolti in una particolare relazione ad un istante specifico. La nozione di tempo è fondamentale. Nell'esempio di prima, nel tempo una persona potrebbe essere assunta da una serie di aziende.

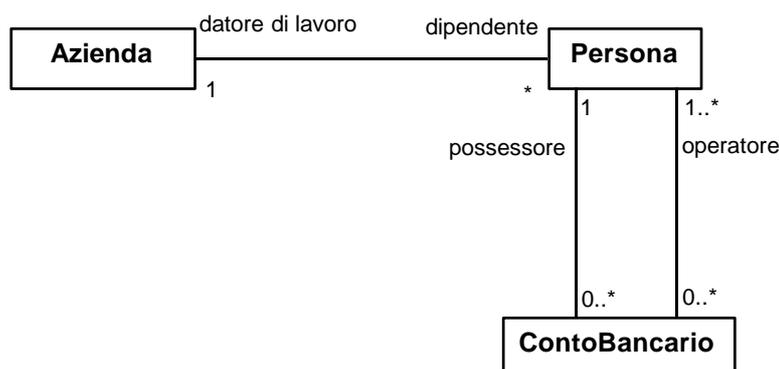
ATTENZIONE: se la molteplicità non è espressa, è indefinita.

La molteplicità è specificata da una lista di intervalli, nella forma minimo..massimo, separati da virgole.

La tabella seguente mostra il significato dei simboli:

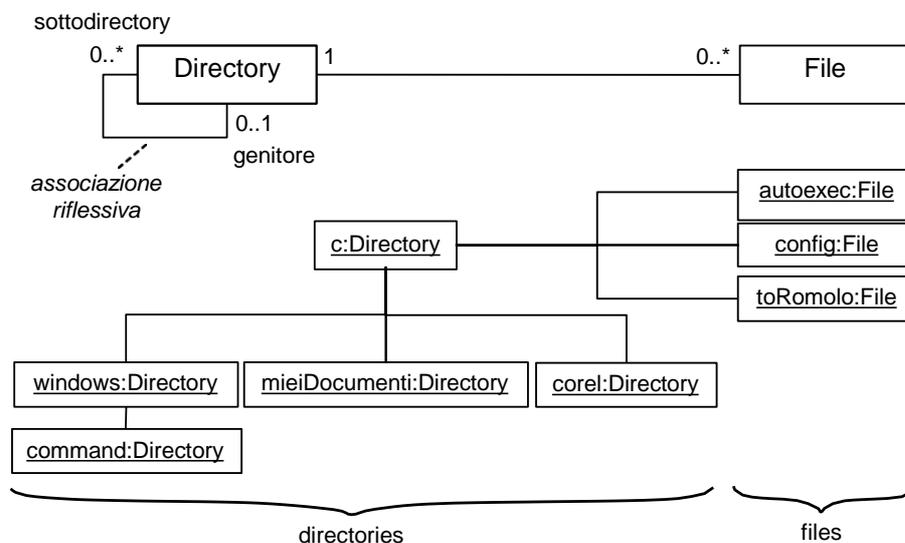
Ornamenti	Semantica
0..1	zero o uno
1	esattamente 1
0..*	zero o più
*	zero o più
1..*	uno o più
1..6	da uno a sei
1..3,7..10,15, 19..*	da 1 a 3 o da 7 a 10 o esattamente 15 o da 19 a molti

Un esempio:



NOTA BENE: la molteplicità stabilisce regole che devono essere soddisfatte nel dominio del problema e quindi è importante stabilire la molteplicità già a livello di analisi.

Analizziamo il diagramma seguente:



La relazione figlio/genitore è una associazione riflessiva (una classe ha un'associazione con se stessa), ossia oggetti della classe hanno collegamenti con oggetti della stessa classe.

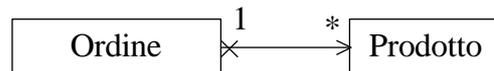
- **La navigabilità** – esprime la direzione delle associazioni. Nella tabella seguente sono riportati i tre idiomi di navigabilità suggeriti da UML 2.0. Scegli un idiomma ed usa sempre quello durante il progetto. Le celle scure stanno a significare che per l'idioma non è definita la sintassi.

Sintassi UML 2	Idioma 1: Navigabilità UML 2.0 stretta	Idioma 2: Nessuna navigabilità	Idioma 3: standard nella pratica
	Da A a B è navigabile Da B a A è navigabile		
	Da A a B è navigabile Da B a A non è navigabile		
	Da A a B è navigabile Da B a A è indefinito		Da A a B è navigabile Da B a A non è navigabile
	Da A a B è indefinito Da B a A è indefinito	Da A a B è indefinito Da B a A è indefinito	Da A a B è navigabile Da B a A è navigabile
	Da A a B non è navigabile Da B a A non è navigabile		

Il terzo idioma, sebbene utilizzato spesso nella pratica, presenta alcuni problemi:

- Non è possibile dire dal diagramma se la navigabilità è stata definita;
- Cambia il significato della freccia (da navigabile/indefinito a navigabile/non-navigabile);
- Non permette di presentare associazioni che non sono navigabili in entrambi le direzioni.

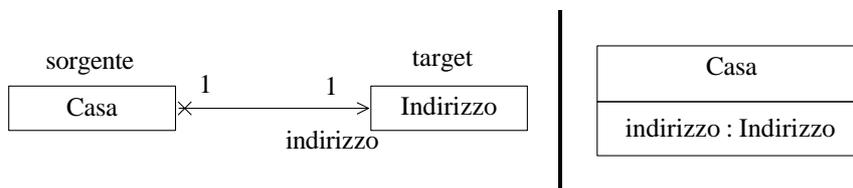
Usando la notazione UML stretta possiamo definire



La associazione ci dice che un Ordine memorizza una lista di prodotti. Non si può navigare direttamente dal Prodotto all'Ordine. Comunque, si potrebbe ancora trovare l'oggetto Ordine associato al particolare Prodotto cercando tutti gli oggetti Ordine!!!!

Associazione ed attributi

Un'associazione tra una classe sorgente ed una classe target significa in pratica che oggetti della classe sorgente possono possedere un riferimento ad oggetti della classe target. Visto diversamente, un'associazione è equivalente ad una classe sorgente che ha uno pseudo-attributo del tipo della classe target.

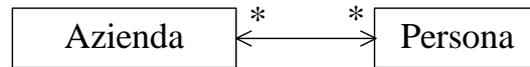


Se la molteplicità è multipla, le associazioni sono implementate o come array di attributi oppure come collezioni. Collezioni sono classi che hanno la capacità di memorizzare e recuperare riferimenti ad altri oggetti.

ATTENZIONE: usa l'associazione quando la classe target è una parte importante del modello. Altrimenti usa gli attributi. Classi importanti sono quelle che descrivono parti del dominio del problema. Classi non importanti sono quelle che si trovano nelle librerie (String, Date, Time, etc.). In genere, se la molteplicità è 1 sia per la classe sorgente che per la classe target, la scelta di rappresentare la classe target come attributo può essere la soluzione migliore.

Classi di Associazioni

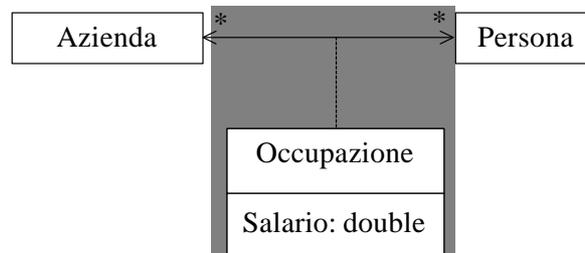
PROBLEMA: quando si ha un'associazione molti a molti tra due classi, ci possono essere degli attributi che non possono essere facilmente inseriti in nessuna delle due classi. Consideriamo l'esempio seguente:



Cosa accade se si deve inserire la regola che ogni Persona percepisce un salario da ogni Azienda da cui è assunto? Dove dovremmo inserire l'attributo salario?

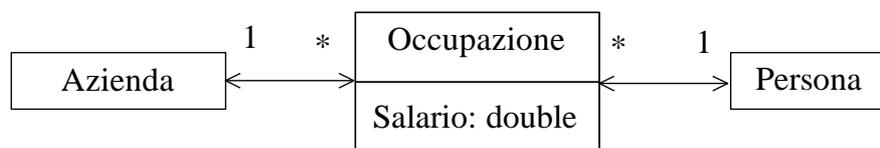
Una Persona può essere assunta da più aziende con salari diversi. Un'Azienda ha diversi dipendenti con salari diversi.

CONCLUSIONE: il salario è una proprietà dell'associazione. UML permette di modellare questo scenario con le classi di associazione. La **classe di associazione** è rappresentata da tutto ciò che è racchiuso nel rettangolo grigio (che fa parte del modello) nella figura seguente.



Le classi di associazione possono avere attributi, operazioni e altre associazioni. Istanze delle classi di associazione sono collegamenti che hanno attributi e operazioni. L'identità di questi collegamenti è determinata esclusivamente dalle identità degli oggetti ai loro estremi. Quindi, la classe di associazione può essere solo usata quando c'è un unico collegamento tra due oggetti ad uno specifico istante.

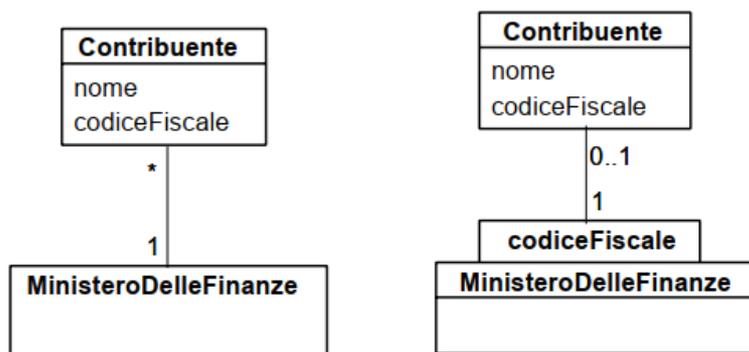
Se una persona può avere più di un lavoro con la stessa azienda, la classe di associazione non può essere usata. **SOLUZIONE:** esprimi la relazione come una classe



Associazioni qualificate

Consideriamo il diagramma a sinistra nella figura seguente. Come è possibile per il MinisteroDelleFinanze “navigare” ad uno specifico Contribuente? È necessaria una chiave unica che trasformi l’associazione n-a-molti in un’associazione n-a-uno. Questa chiave è conosciuta come **qualificatore** (diagramma a destra nella figura). **NOTA BENE:** il qualificatore appartiene all’associazione e non alla classe.

Tipicamente, i Qualificatori si riferiscono ad un attributo nella classe target, ma possono anche essere espressioni.

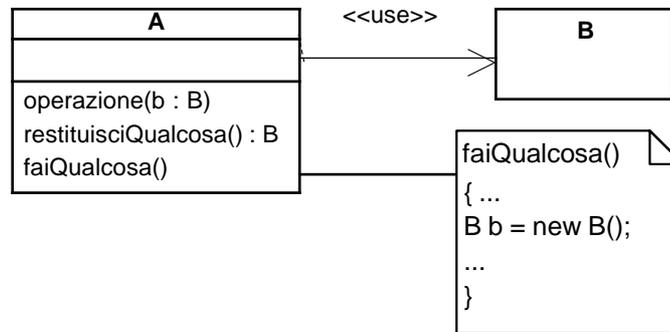


Dipendenza

Lo *UML Reference Manual* definisce una dipendenza come “una relazione tra due elementi in cui una modifica ad un elemento (il fornitore) può influenzare l’altro elemento (il cliente) o fornire ad esso informazione necessaria”. Una dipendenza è modellata da una linea tratteggiata con una freccia all’estremo. Le dipendenze possono avvenire tra classi, tra oggetti e classi, tra package e package e tra un’operazione ed una classe.

UML 2.0 specifica tre tipi di dipendenze:

- **Dipendenza di uso:** il cliente usa alcuni servizi messi a disposizione dal fornitore. Ci sono cinque dipendenze d’uso, individuate da altrettanti stereotipi:
 - «use» - il cliente usa il fornitore in qualche modo. In genere se lo stereotipo è omesso, la dipendenza è una dipendenza d’uso.



La classe A utilizza la classe B perché:

1. un'operazione della classe A ha bisogno di un parametro della classe B
2. un'operazione della classe A restituisce un oggetto della classe B
3. un'operazione della classe A usa un oggetto della classe B, ma non come attributo.

I casi 1. e 2. possono essere modellati più accuratamente da una dipendenza d'uso stereotipata «**parameter**» ed il caso 3. da una dipendenza d'uso stereotipata «**call**».

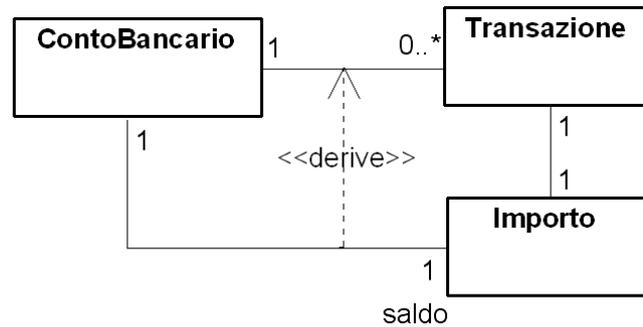
- «**call**» - un'operazione nel cliente invoca un'operazione nel fornitore.

- «**parameter**» - il fornitore è un parametro di un'operazione del cliente;
- «**send**» - il cliente è un'operazione che invia il fornitore (che deve essere un segnale) a qualche specificato target.
- «**instantiate**» - il cliente è un'istanza del fornitore.

- **Dipendenze di astrazione:** modellano dipendenze tra entità che sono a livelli di astrazione differenti (per esempio, una classe di analisi ed una classe di progetto). L'entità fornitore è più astratta dell'entità cliente. Ci sono quattro livelli di astrazione:

- «**trace**» - una dipendenza in cui il fornitore ed il cliente rappresentano lo stesso concetto ma in differenti modelli (una specifica funzionale ed il caso d'uso che la supporta, una classe di analisi ed una classe di progetto).
- «**substitute**» - indica che il cliente può essere utilizzato al posto del fornitore a tempo di esecuzione. Il cliente ed il fornitore devono mettere a disposizione lo stesso insieme di servizi.
- «**refine**» - simile a «**trace**», ma all'interno dello stesso modello. Per esempio, una classe che viene ottimizzata per motivi di prestazioni costituisce un raffinamento della classe originale.

- «**derive**» - indica che un'entità può essere derivata in qualche modo da un'altra entità. Nell'esempio seguente, la relazione di dipendenza stereotipata «**derive**» sta ad indicare che il saldo del conto corrente può essere sempre calcolato su domanda dagli importi di tutte le transazioni.



- **Dipendenze di permesso:** modellano la capacità di un'entità di accedere ad un'altra entità. Ci sono tre dipendenze di permesso:

- «**access**» - si usa tra package e permette ad un package (cliente) di accedere a tutti i contenuti pubblici di un altro package (fornitore). Ogni package definisce il proprio namespace: con «**access**» i namespace rimangono separati e quindi il cliente deve usare il percorso completo (pathname) dei nomi per accedere all'elemento nel fornitore.

- «**import**» - simile al precedente con la differenza che si ha la fusione del namespace. **ATTENZIONE:** se si hanno gli stessi nomi nei due namespace, si deve ricorrere al percorso completo.
- «**permit**» - permette una violazione controllata dell'incapsulamento quale sia la visibilità dichiarata dal fornitore. Questa dipendenza si ritrova per esempio a livello di linguaggio nelle dichiarazioni friend in C++.

Ereditarietà

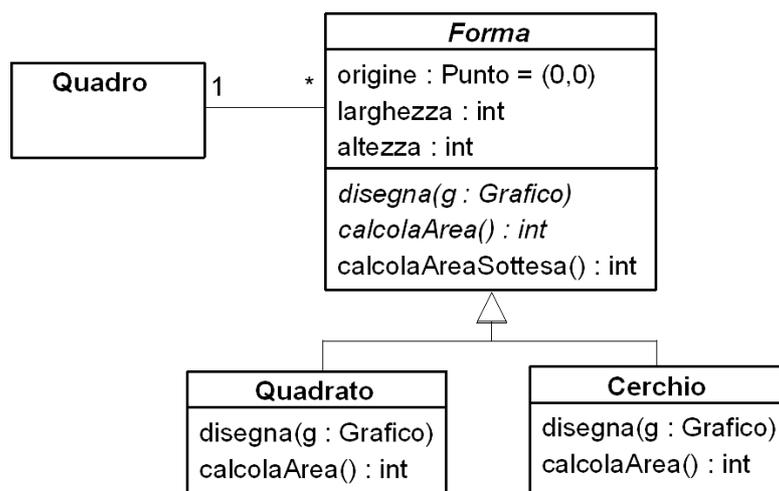
La generalizzazione è la relazione tra un elemento più generale ed uno più specifico, dove l'elemento più specifico è *interamente consistente* con il più generale.

L'elemento più specifico può essere usato ovunque sia utilizzabile l'elemento più generale senza provocare problemi al sistema.

ATTENZIONE: la generalizzazione è la forma più forte di dipendenza. La classe più specifica (*sottoclasse* o *classe figlio*) eredita tutti gli attributi, le operazioni, le relazioni ed i vincoli della classe più generale (superclasse o classe genitore). Inoltre, la sottoclasse può aggiungere nuove caratteristiche o sovrascrivere quelle ereditate.

Nell'esempio presentato di seguito, la classe *Forma* descrive una generica figura geometrica piana. L'operazione *disegna()* disegna su video la forma, l'operazione *calcolaArea()* calcola la sua area e l'operazione *calcolaAreaSottesa()* restituisce il risultato del prodotto *larghezza*altezza* del rettangolo che racchiude la forma. Mentre la definizione di questa operazione è valida per tutte le forme, la definizione delle altre due operazioni deve essere adattata alla specifica forma. Queste due operazioni devono quindi essere sovrascritte.

Per sovrascrivere un'operazione, una sottoclasse deve definire un'operazione con la stessa intestazione. ATTENZIONE: in UML, l'intestazione è specificata dal nome, il tipo dell'oggetto restituito ed i tipi dei parametri.

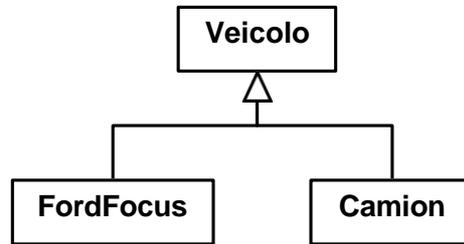


Nella figura, le operazioni *disegna()* e *calcolaArea()* della classe *Forma* sono scritte in italico. Questo non è un errore, ma indica che le due operazioni sono astratte (non implementate). Segue che la classe *Forma* è una *classe astratta* ossia non istanziabile. Le classi derivate *Quadrato* e *Cerchio* devono sovrascrivere queste operazioni, definendo il comportamento appropriato. Le classi *Quadrato* e *Cerchio* sono definite *classi concrete*.

NOTA BENE: le operazioni astratte definiscono un contratto che tutte le sottoclassi concrete devono implementare.

ATTENZIONE: mantieni un livello uniforme di astrazione ad ogni livello della gerarchia di generalizzazione. Le classi derivate da una medesima classe base devono avere lo stesso **livello di astrazione**.

Il modello seguente è corretto?



È errato derivare *FordFocus* e *Camion* da *Veicolo*, poiché la prima classe è un tipo specifico di automobile, la seconda una categoria di veicolo.

NOTA BENE: UML permette l’ereditarietà multipla. L’ereditarietà multipla è tipicamente considerata un problema di progetto.

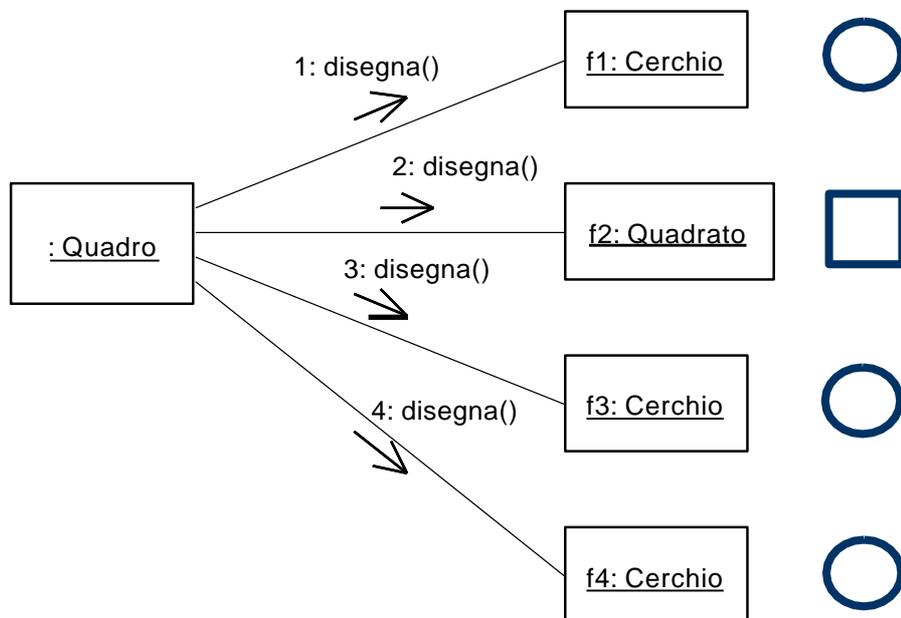
Polimorfismo

Polimorfismo significa “molte forme”. Un’operazione polimorfica è un’operazione che ha molte implementazioni. Le operazioni `disegna()` e `calcolaArea()` viste precedentemente sono esempi di operazioni polimorfiche.

La potenza del polimorfismo è che permette di inviare lo stesso messaggio ad oggetti di classi differenti ed ottenere sempre la risposta adeguata.

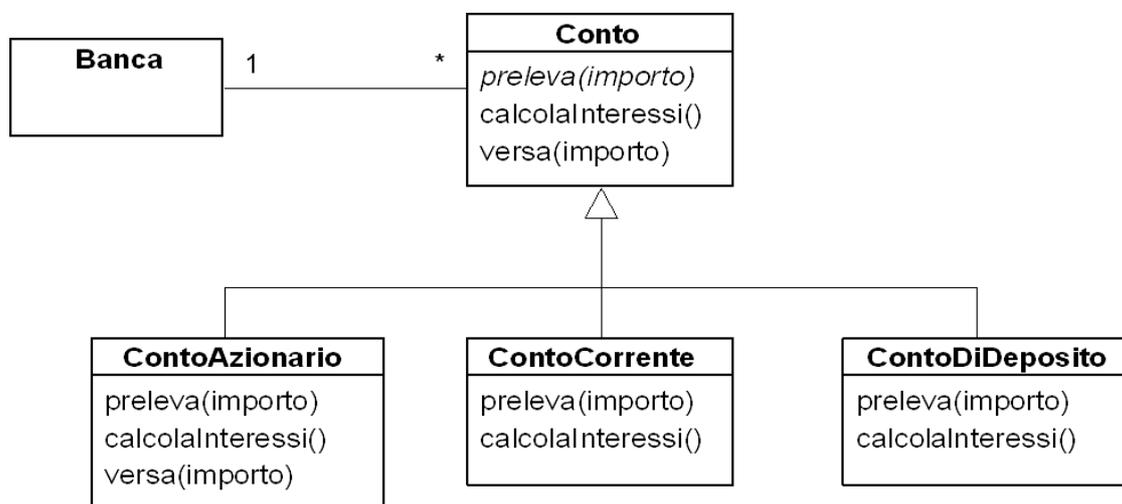
Un esempio di polimorfismo è fornito dal diagramma della classe *Forma* visto precedentemente.

Nel diagramma seguente, la classe *Quadro* mantiene una collezione di istanze di forme. Per esempio:



Cosa succede quando l'oggetto di classe Quadro iterando sugli oggetti della collezione invia ad ognuno il messaggio disegna()? Ovviamente ogni oggetto farà la cosa corretta.

Di seguito, viene dato un altro esempio di polimorfismo.



L'operazione versa() è un'operazione concreta della classe Conto che viene sovrascritta nella classe ContoAzionario, per esempio, per aggiungere regole diverse di calcolo delle commissioni a seconda dell'importo versato. Anche un'operazione concreta può quindi essere sovrascritta. ATTENZIONE: esiste già un'implementazione esistente e cambiare questa implementazione può provocare degli effetti collaterali indesiderati. Cerca sempre di utilizzare l'implementazione esistente, semplicemente aggiungendo il comportamento proprio dell'oggetto.

Realizzazione dei casi d'uso

Mentre le classi di analisi modellano la struttura statica di un sistema, le realizzazioni dei casi d'uso descrivono come le istanze delle classi di analisi interagiscono per realizzare le funzionalità del sistema (parte della vista dinamica del sistema).

Gli obiettivi delle realizzazioni dei casi d'uso sono:

- Trovare quali classi di analisi interagiscono per realizzare il comportamento specificato da un caso d'uso (è possibile scoprire nuove classi di analisi);
- Trovare quali messaggi le istanze di queste classi devono inviarsi l'un l'altra per realizzare il comportamento specificato. Questo serve per determinare:
 - Le operazioni chiave che le classi di analisi devono possedere;
 - Gli attributi chiave di ogni classe di analisi;
 - Relazioni importanti tra le classi;
 - Aggiornamenti del modello dei casi d'uso, del modello dei requisiti e delle classi di analisi.

ATTENZIONE: considera solo i casi d'uso principali e lavora alla realizzazione di questi. Alla fine avrai un modello di analisi che fornisce un quadro ad alto livello del comportamento dinamico del sistema.

Cosa sono le realizzazioni dei casi d'uso?

Consistono di insiemi di classi che realizzano il comportamento specificato in un caso d'uso. Per esempio, se abbiamo un caso d'uso PrestaLibro ed abbiamo identificato le classi di analisi Libro, Ricevuta, Utente e l'attore Bibliotecario, la realizzazione del caso d'uso dimostra come le classi e gli oggetti interagiscono. In questo modo, un caso d'uso, che è un requisito funzionale, viene trasformato in diagrammi di classi e di interazione, che sono specifiche ad alto livello del sistema.

Sebbene UML preveda un simbolo per le realizzazioni dei casi d'uso (ellisse tratteggiata), queste sono raramente modellate (un caso d'uso ha un'unica realizzazione e quindi modellare le realizzazioni non aggiunge informazione).

Le realizzazioni vengono definite aggiungendo gli elementi appropriati al modello: eventuali nuove classi, diagrammi di interazione, nuovi requisiti che vengono catturati e raffinamenti dei casi d'uso.

La realizzazione dei casi d'uso è un processo di raffinamento: un aspetto del comportamento del sistema definito in un caso d'uso o nei requisiti viene modellato attraverso le interazioni tra le istanze delle classi di analisi che sono state identificate.

Tali aspetti dinamici vengono descritti attraverso diagrammi di interazione che sono di quattro tipi: *diagrammi di sequenza* (sequence diagram), *diagrammi di comunicazione* (communication diagram), *diagrammi di visione generale dell'interazione* (interaction overview diagram) e *diagrammi di temporizzazione* (timing diagram).

Interazioni

Le interazioni sono unità di comportamento di un classificatore. Questo classificatore, conosciuto come *classificatore di contesto*, fornisce il contesto dell'interazione.

Un'interazione può usare ogni caratteristica del suo classificatore di contesto o ogni caratteristica a cui il classificatore accede.

Nella realizzazione dei casi d'uso, il classificatore di contesto è il caso d'uso e le interazioni servono a dimostrare come il comportamento descritto dal caso d'uso possa essere realizzato attraverso le istanze dei classificatori (in questo caso, le classi di analisi) che si inviano messaggi l'un l'altro.

Linee di vita

Una linea di vita (lifeline) rappresenta un partecipante in un'interazione, cioè rappresenta come un'istanza di un classificatore partecipa nell'interazione.

Ogni linea di vita ha un nome (opzionale), un tipo e un selettore (opzionale). Il selettore è una condizione Booleana che può essere usata per selezionare un'istanza che soddisfa la condizione. I selettori sono validi solo se il tipo ha una molteplicità maggiore di 1.



La linea di vita rappresenta un ruolo che un'istanza del classificatore può interpretare nell'interazione. **NOTA BENE:** la linea di vita non rappresenta nessuna istanza in particolare, ma un'istanza generica. Nei diagrammi di interazione possono comunque anche essere usate istanze specifiche (in questo caso si adotta la notazione propria dell'istanza). Quando si usano le linee di vita si produce un diagramma di interazione di *forma generica*, mentre quando si usano le istanze si produce un diagramma di interazione di *forma istanza*. L'interazione avviene attraverso lo scambio di messaggi.

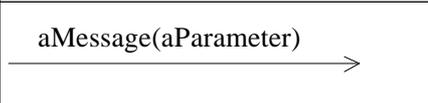
Messaggi

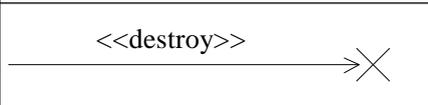
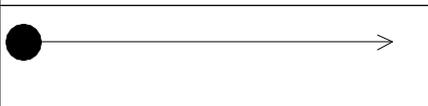
Un messaggio rappresenta una specifica di comunicazione tra due linee di vita in un'interazione. La comunicazione può essere:

- Un'invocazione di un'operazione;
- La creazione o distruzione di un'istanza;
- L'invio di un segnale.

Quando esegue un messaggio, una linea di vita è il punto focale (focus) del **controllo** o (**attivazione**). Il passaggio delle attivazioni tra le linee di vita costituisce il **flusso di controllo**.

Ci sono sette tipi di messaggi come indicato nella tabella seguente:

	Messaggio Sincrono	Il mittente aspetta che il ricevitore esegua il messaggio
	Messaggio Asincrono	Il mittente invia il messaggio e continua l'esecuzione

	Ritorno del messaggio	Il ricevitore del messaggio restituisce il controllo al mittente
	Creazione di un oggetto	
	Distruzione di un oggetto	
	Messaggio trovato	Il mittente del messaggio non è visibile nell'interazione
	Messaggio perso	Il messaggio non raggiunge mai la sua destinazione

Tipicamente, durante l'analisi non si scende nel dettaglio se un messaggio è sincrono o asincrono: in generale si considerano tutti i messaggi sincroni (caso più restrittivo).

Il messaggio trovato può essere utile quando si desidera presentare la ricezione di un messaggio per una classe, ma non si conosce da dove il messaggio è stato originato.

I messaggi persi possono essere utili durante il progetto per presentare come i messaggi potrebbero essere persi durante una condizione di errore.

Diagrammi di interazione

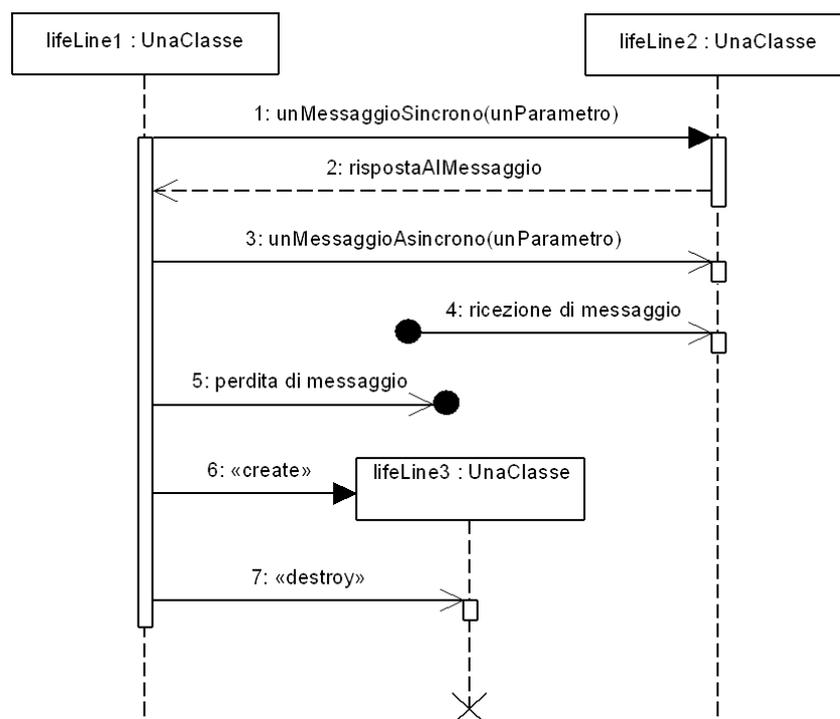
Ci sono quattro tipi differenti di diagrammi di interazione, ognuno dei quali enfatizza un aspetto differente dell'interazione

- **diagrammi di sequenza** - mostrano la sequenza, ordinata temporalmente, di messaggi scambiati tra linee di vita. In un diagramma di sequenza il tempo scorre dall'alto verso il basso, e le varie linee di vita sono affiancate orizzontalmente.
- **Diagrammi di comunicazione** – mostrano le relazioni strutturali tra oggetti. Utili in analisi per avere velocemente un'idea di una collaborazione tra oggetti.
- **Diagrammi di visione generale dell'interazione** – mostrano come un comportamento complesso è realizzato da un insieme di interazioni più semplici.
- **Diagrammi di temporizzazione** – mostrano gli aspetti real-time di un'interazione.

Diagrammi di sequenza

Di seguito, viene dato un esempio di diagramma di sequenza con l'uso dei sette tipi di messaggi. Come si può notare il tempo procede dall'alto in basso e le linee di vita da sinistra a destra. Le linee di vita sono disposte orizzontalmente in modo da minimizzare le

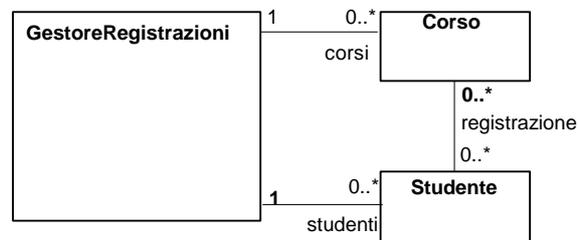
intersezioni nel diagramma e verticalmente a seconda della loro creazione. Le linee tratteggiate indicano la durata della linea di vita nel tempo. I rettangoli sottili e lunghi indicano quando una linea di vita ha il controllo del flusso.



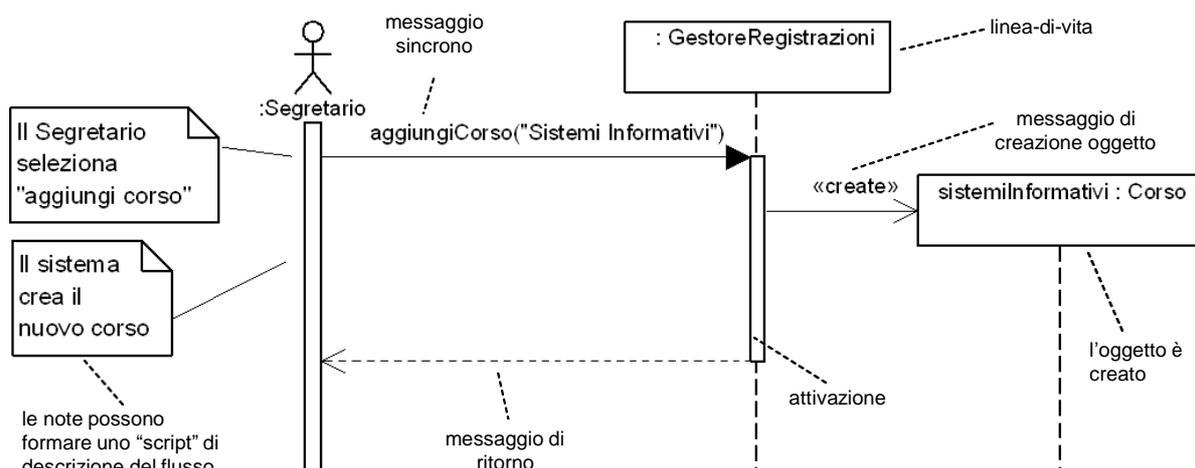
Consideriamo il caso d'uso seguente:

Caso d'uso: <i>AggiungiCorso</i>
ID: 8
Breve descrizione: <i>Aggiunge al sistema i dettagli di un nuovo corso</i>
Attori primari: <i>Segretario</i>
Attori secondari: <i>Nessuno</i>
Precondizioni: <i>1. Il Segretario ha effettuato il login nel sistema</i>
Flusso principale: <i>1. Il Segretario seleziona "aggiungi corso". 2. Il Segretario inserisce il nome del nuovo corso. 3. Il Sistema crea il nuovo corso</i>
Postcondizioni: <i>1. Un nuovo corso viene aggiunto nel sistema</i>
Flussi alternativi: <i>CorsoGiaEsistente</i>

Dall'analisi del caso d'uso si può derivare il diagramma seguente delle classi di analisi.



La figura seguente è un esempio di diagramma di sequenza che realizza il comportamento specificato dal caso d'uso *AggiungiCorso*.

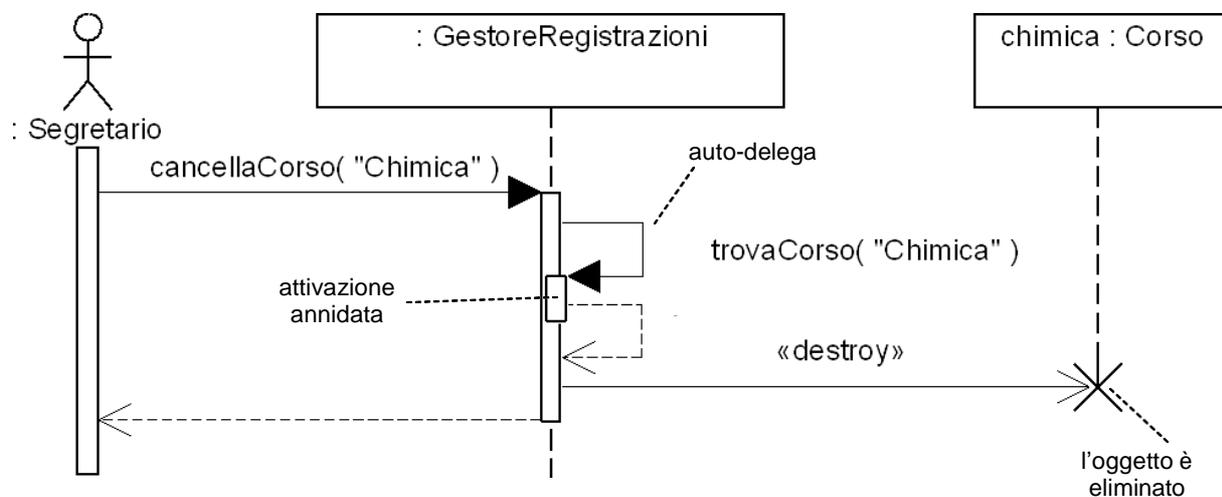


NOTA BENE: il diagramma non mostra una rappresentazione esatta di ogni passo del caso d'uso. I primi due passi prevedono una interazione con una interfaccia utente che verrà mostrata in fase di progetto.

Consideriamo il caso d'uso Cancellacorso.

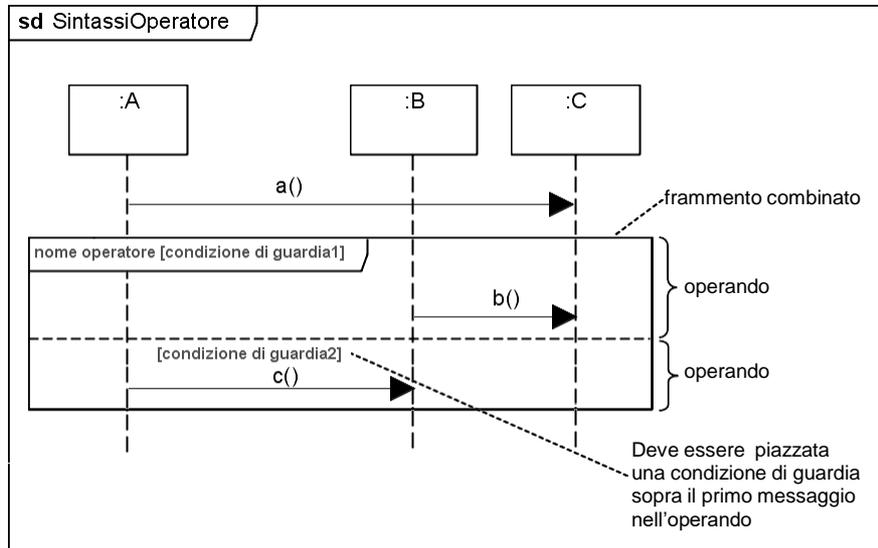
Caso d'uso: <i>Cancellacorso</i>
ID: 9
Breve descrizione: <i>Cancella un corso dal sistema</i>
Attori primari: <i>Segretario</i>
Attori secondari: <i>Nessuno</i>
Precondizioni: <i>1. Il Segretario ha effettuato il login nel sistema</i>
Flusso principale: <i>1. Il Segretario seleziona "cancella corso". 2. Il Segretario inserisce il nome del corso. 3. Il Sistema rimuove il corso</i>
Postcondizioni: <i>1. Un corso è stato rimosso dal sistema</i>
Flussi alternativi: <i>CorsoInesistente</i>

Il diagramma di sequenza seguente mostra l'interazione tra le istanze delle classi di analisi per realizzare la cancellazione del corso. Nel diagramma viene mostrata la distruzione di un oggetto e l'auto-delega (self-delegation), ossia una linea-di-vita che invoca una propria operazione, tipico nei sistemi OO. In particolare, questo crea una attivazione annidata sulla medesima linea-di-vita.



Frammenti Combinati

I diagrammi di sequenza possono essere suddivisi in “aree” dette **frammenti combinati**. Ogni frammento combinato ha un operatore, uno o più operandi e zero o più condizioni di guardia.



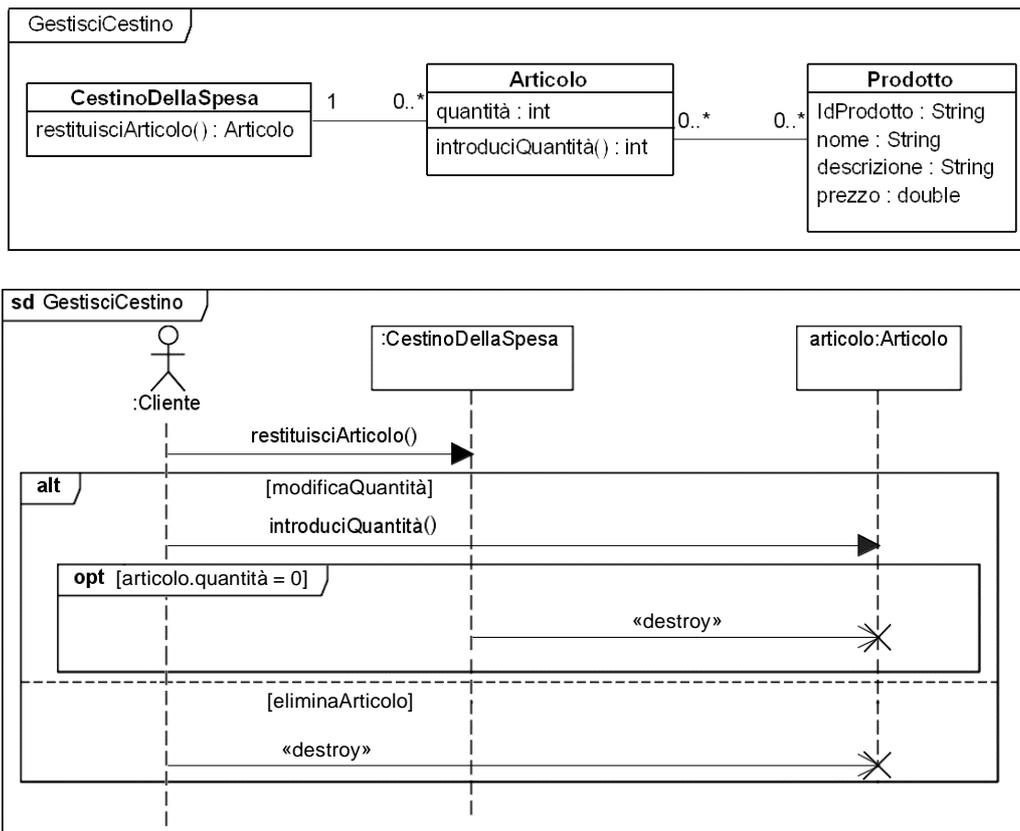
L'operatore determina *come* i suoi operandi sono eseguiti. Le condizioni di guardia determinano se l'operando può essere eseguito. La tabella seguente riassume i principali operatori.

Operatore	Nome	Semantica
opt	Opzione	Se la condizione è vera, viene eseguito un singolo operando
alt	alternative	L'operando la cui condizione è vera viene eseguito
loop	loop	Sintassi: loop min, max [condition] cicla max-min volte mentre la condizione è vera
break	break	Se la condizione di guardia è vera, l'operando è eseguito, ma non il resto dell'interazione
ref	riferimento	Il frammento combinato si riferisce ad un'altra interazione
par	parallelo	Tutti gli operandi vengono eseguiti in parallelo
critical	critico	L'operando viene eseguito atomicamente
seq	sequenza debole	Tutti gli operandi vengono eseguiti in parallelo con il vincolo che gli eventi che arrivano sulla stessa linea di vitada operandi differenti avvengano nella stessa sequenza degli operandi
strict	sequenza stretta	Gli operandi vengono eseguiti in stretta sequenza

Consideriamo il caso d'uso seguente:

Caso d'uso: <i>GestisciCestino</i>
ID: 2
Breve descrizione: <i>Il Cliente cambia la quantità di articoli nel cestino</i>
Attori primari: <i>Cliente</i>
Attori secondari: <i>Nessuno</i>
Precondizioni: <i>1. Il contenuto del cestino della spesa è visibile</i>
Flusso principale: <i>1. Il caso d'uso inizia quando il Cliente seleziona un articolo nel cestino.</i> <i>2. If il Cliente seleziona "elimina articolo"</i> <i>1. Il sistema rimuove l'articolo dal cestino</i> <i>3. If il Cliente introduce una nuova quantità</i> <i>1. Il sistema aggiorna la quantità di articoli nel cestino</i>
Postcondizioni: <i>Nessuna</i>
Flussi alternativi: <i>Nessuno</i>

Il diagramma delle classi di analisi corrispondente al caso d'uso è come segue:



L'operatore loop nella sua forma più generale `loop min, max [condition]` corrisponde a:

```
loop min times then
  while (condition is true)
    loop (max-min) times
```

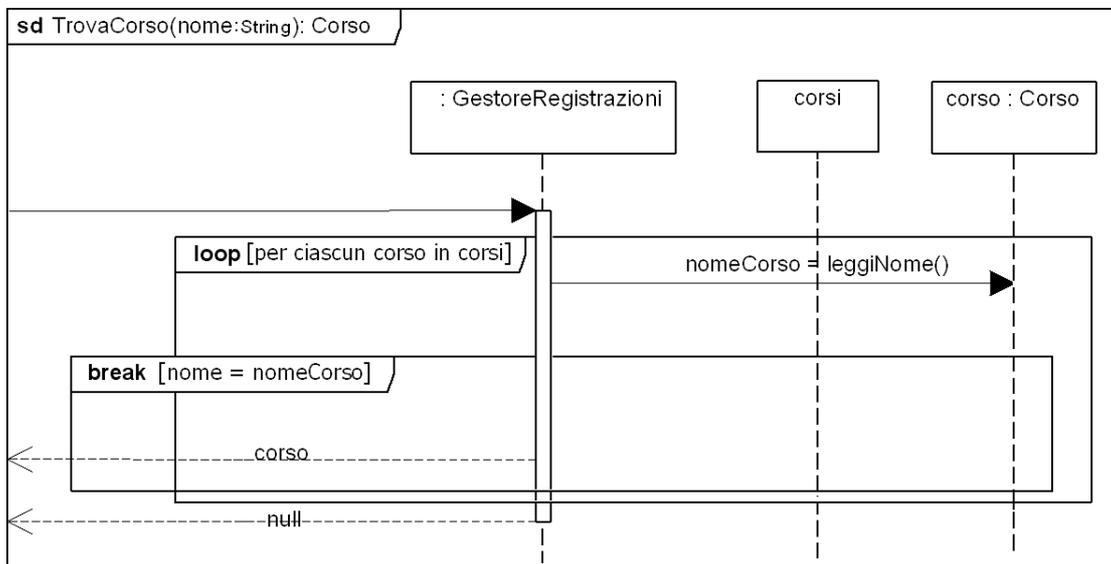
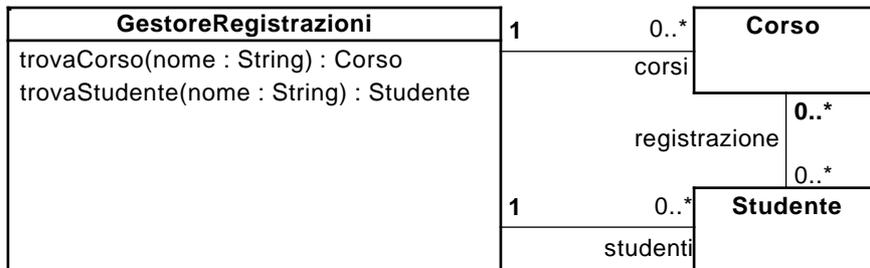
Un loop senza min, max e condition è un loop infinito. Se è dato solo min allora max = min. La forma piuttosto complessa con cui viene definito l'operatore loop consente di realizzare una vasta varietà di idiomi. Per esempio,

- `loop o loop *` \Rightarrow `while(true) {body}`
- `loop n,m` \Rightarrow `for i=n to m {body}`
- `loop [booleanExpression]` \Rightarrow `while(booleanExpression) {body}`
- `loop 1,* [booleanExpression]` \Rightarrow `repeat {body} while(booleanExpression)`
- `loop [for each object in collectionOfObjects]` \Rightarrow `forEach object in collection {body}`
- `loop [for each object in ClassName]` \Rightarrow `forEach object of class {body}`

Consideriamo il caso d'uso seguente.

Caso d'uso: <i>TrovaCorso</i>
ID: <i>11</i>
Breve descrizione: <i>Trova un corso con un dato nome</i>
Attori primari: <i>Segretario</i>
Attori secondari: <i>Nessuno</i>
Precondizioni: <i>Il Segretario ha effettuato il login nel sistema</i>
Flusso principale: <ol style="list-style-type: none"> 1. Il caso d'uso inizia quando il Segretario seleziona "trova corso". 2. Il Segretario inserisce il nome del corso da trovare. 3. IF il Sistema trova un corso con il nome inserito <ol style="list-style-type: none"> 1. Il Sistema restituisce il corso trovato 4. ELSE <ol style="list-style-type: none"> 1. Il sistema comunica che non esiste alcun corso con tale nome
Postcondizioni: <i>Nessuna</i>
Flussi alternativi: <i>Nessuno</i>

Il diagramma delle classi di analisi corrispondente al caso d'uso è come segue:



Nel diagramma, la linea di vita “corsi” rappresenta una collezione di oggetti di tipo Corso.

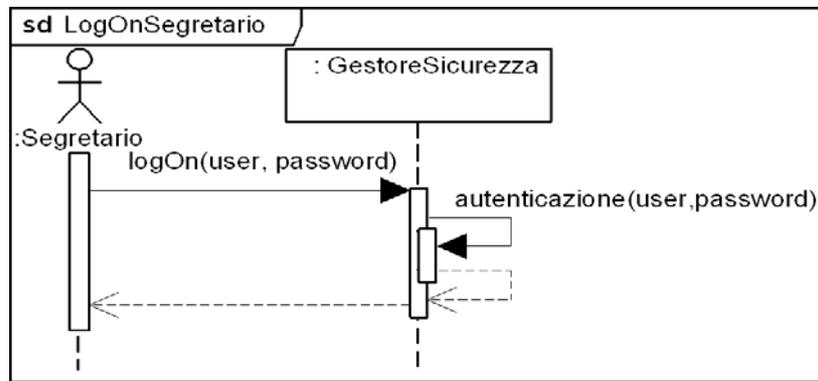
Si noti che il frammento *break* è logicamente fuori del loop: non è parte di esso poiché non fa parte del flusso di controllo che il *loop* prevede. Per questa ragione, il frammento *break* viene disegnato con una sovrapposizione parziale al frammento *loop*.

Occorrenze di Interazione

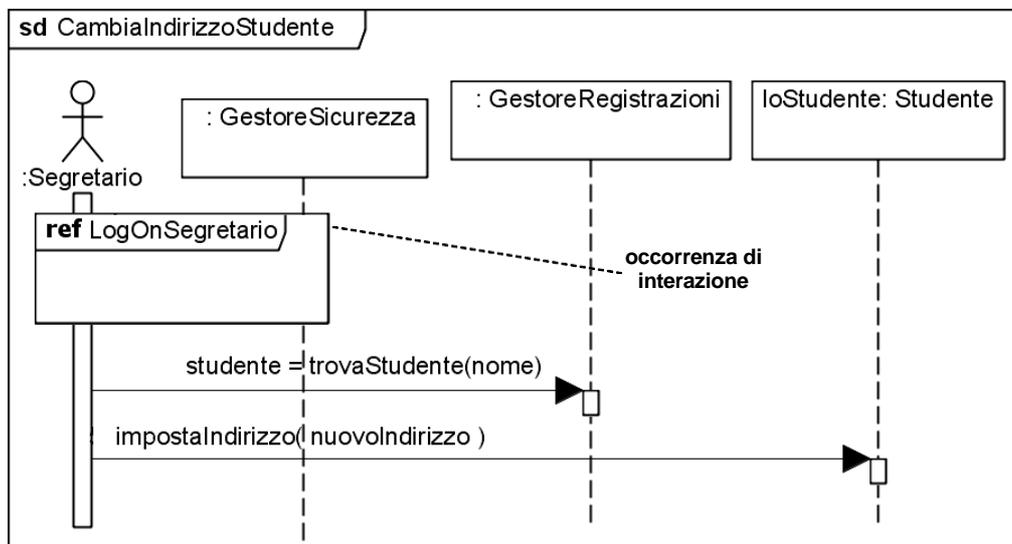
Per evitare di riusare più volte frammenti di interazioni che si ripetono in differenti diagrammi di sequenza, UML 2 mette a disposizione il riferimento ad interazione, detto **occorrenza di interazione**.

Consideriamo l’esempio della gestione dei corsi. Supponiamo che ci sia un GestoreSicurezza che controlla gli accessi. Consideriamo il caso d’uso seguente:

Caso d’uso: <i>LogOnSegretario</i>
ID: 4
Breve descrizione: <i>Il Segretario effettua il logon sul sistema</i>
Attori primari: <i>Segretario</i>
Attori secondari: <i>Nessuno</i>
Precondizioni: <i>Il Segretario non ha effettuato il logon sul sistema</i>
Flusso principale: <i>1. Il caso d’uso inizia quando il Segretario seleziona “log on”. 2. Il sistema chiede al Segretario un nome utente ed una password. 3. Il Segretario inserisce un nome utente ed una password 4. Il sistema riconosce come validi il nome utente e la password</i>
Postcondizioni: <i>1. Il Segretario ha effettuato il logon sul sistema</i>
Flussi alternativi: <i>NomeUtenteEPasswordErrate SegretarioGiàLoggato</i>



Il frammento di interazione LogOnSegretario può potenzialmente essere adoperato all'inizio di un gran numero di diagrammi di sequenza. Consideriamo, per esempio, il diagramma di sequenza seguente che mostra le interazioni necessarie per cambiare l'indirizzo di uno studente.



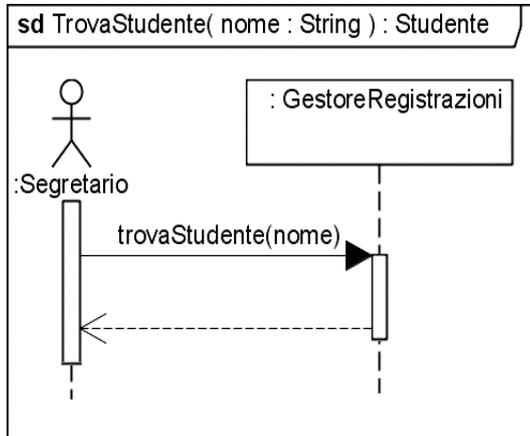
ATTENZIONE: tutte le linee di vita utilizzate nell'occorrenza di interazione devono esistere nell'interazione che la include.

ATTENZIONE: per indicare la visibilità dell'occorrenza di interazione, tracciala attraverso le linee di vita che usa.

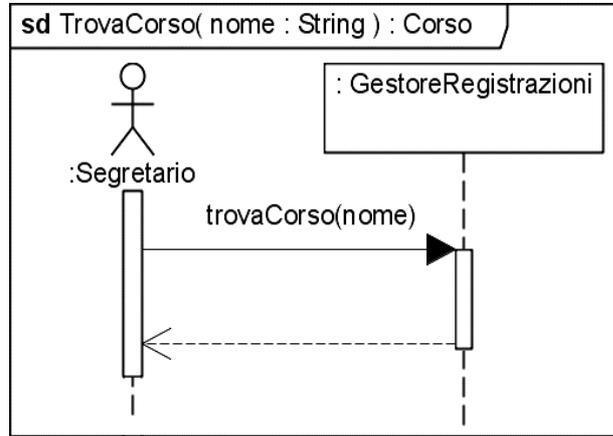
Parametri

Le interazioni possono essere **parametrizzate**. Questo permette di fornire differenti valori all'interazione in ognuna delle sue occorrenze.

Per esempio, l'interazione TrovaStudente può essere parametrizzata e riferita all'interno di un'altra interazione passando il parametro attuale necessario.



a)

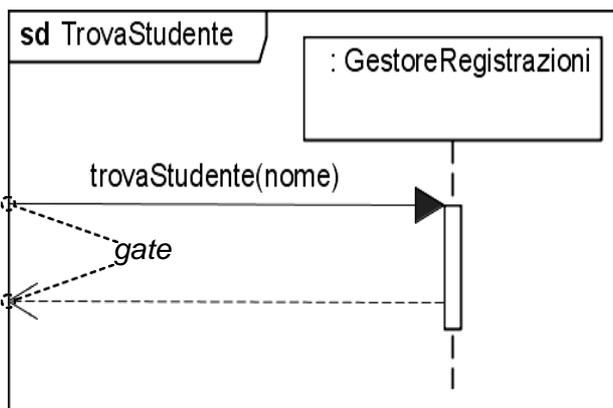


b)

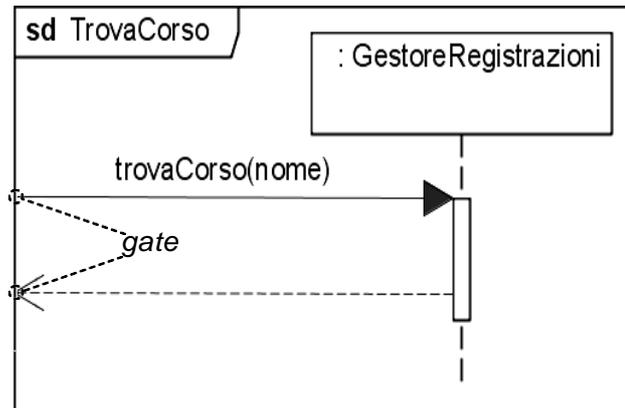
Gate

I cancelli (gate) sono gli ingressi e le uscite delle interazioni. I gate vengono usati quando un'interazione è attivata da una linea di vita che non è parte dell'interazione.

Il gate viene rappresentato come un punto nella cornice del diagramma di sequenza. Questo punto connette un messaggio fuori dalla cornice ad un messaggio dentro la cornice. I diagrammi di sequenza precedenti possono essere modificati utilizzando i gate nel modo seguente.

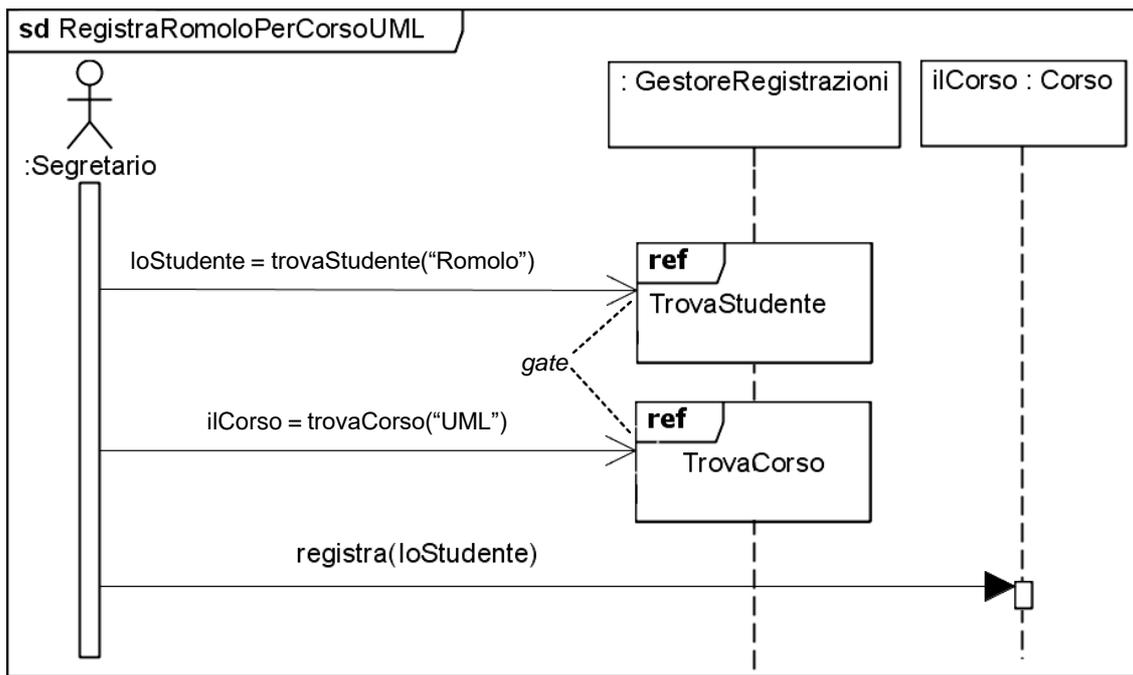


a)



b)

La figura seguente mostra l'utilizzo dei gate.



NOTA BENE: TrovaStudiante non ha più i parametri, ma ha espliciti ingressi ed uscite.

Quando conviene usare i parametri e quando i gate?

- Usa i parametri quando sono conosciute le linee di vita sorgente e destinazione di tutti i messaggi coinvolti nell'interazione.
- Usa i gate quando alcuni messaggi vengono attivati esternamente alla cornice dell'interazione e non conosci da dove effettivamente sono stati originati.

Ingegneria del Software

Mario G.C.A. Cimino - Antonio Luca Alfeo



Workflow Progetto

Ingegneria del
Software

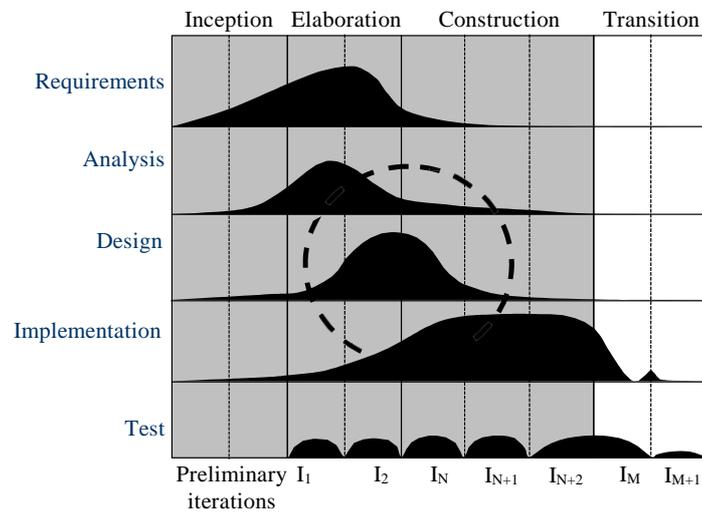
Introduzione

L'obiettivo del workflow Progetto è di specificare completamente come saranno implementate le funzionalità che il sistema dovrà fornire per soddisfare i requisiti dell'utente. Queste funzionalità sono già state modellate logicamente nel workflow Analisi.

Come abbiamo visto precedentemente, i requisiti del sistema sono tipicamente estratti dal dominio del problema e l'analisi può essere considerata come l'esplorazione del dominio dal punto di vista delle persone coinvolte nel sistema. Il progetto consiste nel costruire un modello implementabile del sistema andando a trasferire il dominio della soluzione (librerie di classi, meccanismi di persistenza, ecc.) in soluzioni tecniche.

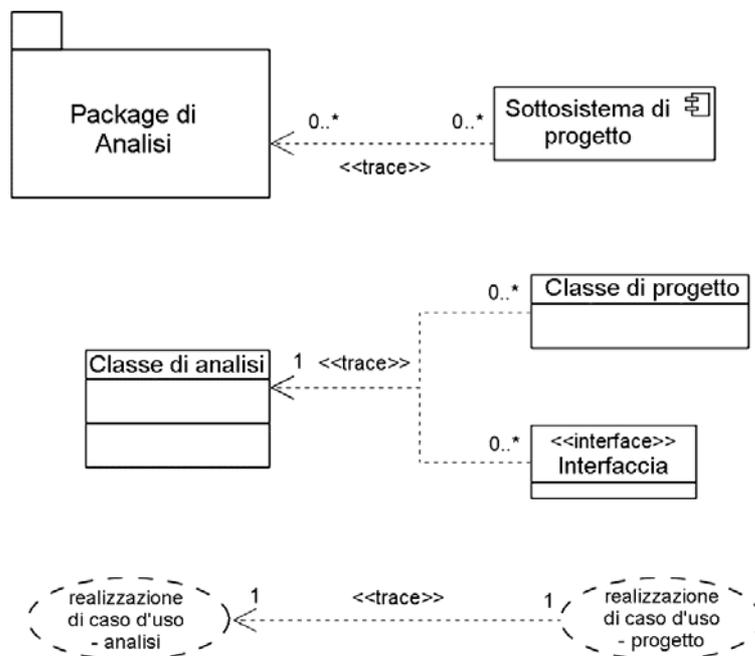
La maggior parte dell'attività richiesta dal workflow Progetto è realizzata durante le fasi di Elaboration e Construction.

Il workflow Progetto si svolge in stretta concomitanza con il workflow Requisiti e Analisi (vedi figura seguente). Ricorda che UP raccomanda che un team sia responsabile della vita di un artefatto (per esempio, un caso d'uso) lungo i workflow Requisiti, Analisi, Progetto ed Implementazione. Infatti, UP organizza il team considerando deliverable e milestone piuttosto che attività.



La figura seguente mostra un metamodello per il modello di progetto. Questo modello contiene molti sottosistemi, che a loro volta possono contenere molti elementi di modellazione. Come illustrato nella figura, l'enfasi maggiore il workflow Progetto la pone nell'identificazione delle interfacce.

Ingegneria del Software



NOTA BENE: un package di analisi potrebbe essere trasformato in più di un sottosistema per ragioni sia tecniche che architetturali.

NOTA BENE: una classe di analisi potrebbe richiedere una o più interfacce o classi di progetto.

Ingegneria del Software

NOTA BENE: le realizzazioni dei casi d'uso di progetto hanno semplicemente più dettagli rispetto a quelle di analisi. La relazione tra casi d'uso di progetto e di analisi è uno ad uno.

Esaminando la figura precedente, viene spontaneo domandarsi: visto che il modello di progetto rifinisce il modello di analisi, si devono mantenere entrambi i modelli?

TIENI PRESENTE che il modello di analisi è meno complesso e quindi più comprensibile. In particolare, il modello di analisi diventa un modello di grande valore per:

- introdurre nuove persone nel progetto;
- capire il sistema mesi o anni dopo la consegna;
- capire come il sistema soddisfi i requisiti;
- gestire la tracciabilità dei requisiti;
- pianificare la manutenzione e i miglioramenti;
- capire l'architettura logica del sistema;
- commissionare l'implementazione del sistema ad aziende esterne (outsourcing).

Ingegneria del
Software

In conclusione, è molto importante mantenere il modello di analisi quando il sistema è grande (più di 200 classi di progetto).

Le classi di Progetto

La classe di progetto completa è una classe sufficientemente dettagliata da servire come una buona base per creare codice sorgente.

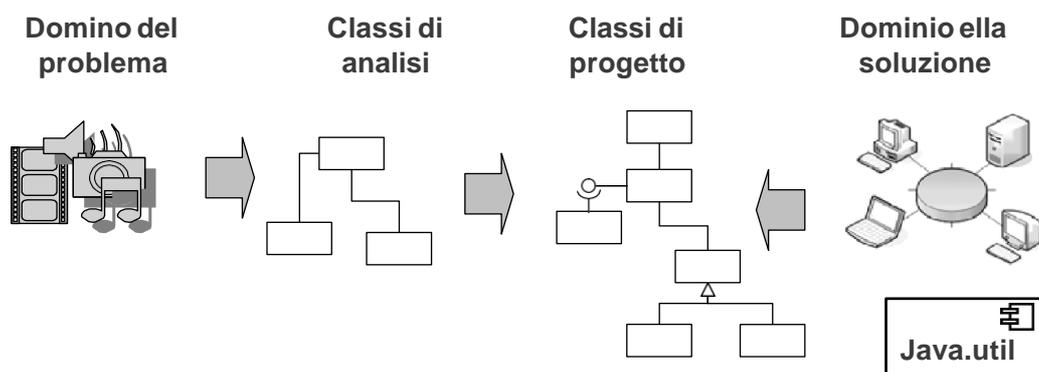
Se lo strumento utilizzato per sviluppare il sistema permette di generare il codice direttamente dal modello, le classi di progetto dovranno essere sviluppate in grande dettaglio; altrimenti potranno essere sviluppate in un dettaglio tale da renderle comprensibili ai programmatori.

Le realizzazioni dei casi d'uso nel workflow Progetto sono sviluppate in parallelo allo sviluppo delle classi di progetto e le includono come parte della loro struttura.

Che cosa sono le classi di progetto?

Le classi di progetto sono classi le cui specifiche sono state completate in modo da permetterne l'implementazione. Le classi di progetto sono generate da:

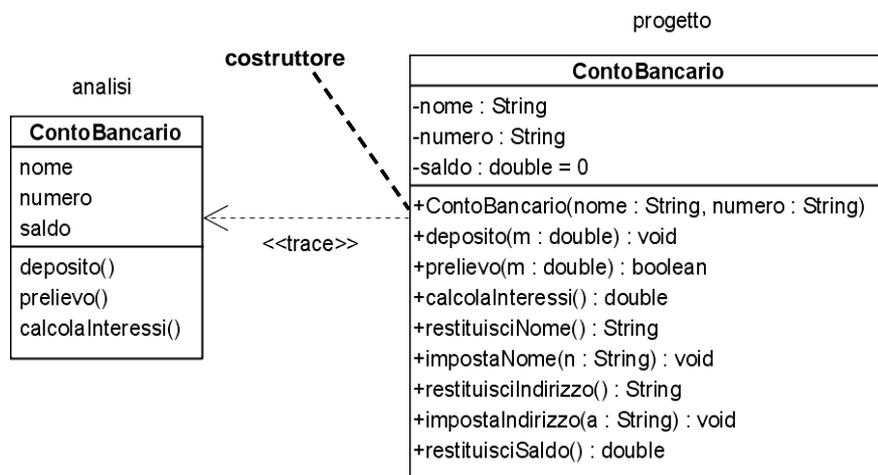
- il **dominio del problema** attraverso il raffinamento delle classi di analisi che avviene aggiungendo dettagli implementativi;
- il **dominio della soluzione** (solution domain) che è composto dall'insieme di librerie di classi, componenti riusabili, database, GUI.



Una classe di analisi può diventare più classi di progetto perché andando a definire in dettaglio le operazioni e gli attributi la classe potrebbe crescere troppo, rendendo quindi necessario suddividerla in classi più piccole. Evita lo “Swiss Army Knife”.

Con le classi di progetto viene specificato esattamente come ogni classe realizzerà le sue responsabilità. A questo proposito:

- l’insieme di attributi deve essere completo e per ogni attributo devono essere specificati il nome, il tipo, la visibilità e, eventualmente, un valore di default;
- l’insieme di operazioni deve essere completo ed in ogni operazione devono essere specificati il nome, la lista dei parametri ed il tipo dell’oggetto restituito. Le operazioni così definite sono a volte chiamate *metodi*.



Un’operazione di analisi può nella realtà essere realizzata da una o più operazioni di progetto. Consideriamo come esempio l’operazione `checkIn()` individuata in analisi. È ovvio che questa operazione quando verrà realizzata durante il progetto avrà bisogno di un insieme di operazioni di aiuto.

Quando una classe di progetto è ben-definita?

Quando possiede almeno queste caratteristiche:

- i. Completa e sufficiente** – le operazioni pubbliche della classe definiscono un contratto tra la classe ed i suoi clienti. Questo contratto deve essere chiaro, ben definito e accettabile per tutti gli interessati.

Una classe è completa se fornisce ai suoi clienti tutto quello che i clienti si aspettano, niente di più niente di meno. Per esempio, una classe ContoCorrente, se fornisce l'operazione preleva() deve anche fornire l'operazione deposita(). Analogamente, una classe CatalogoProdotti deve fornire tutte le operazioni per poter gestire il catalogo.

Una classe è sufficiente se fornisce tutte le operazioni sufficienti a realizzare l'intento della classe. Per esempio, la classe ContoCorrente non dovrebbe fornire operazioni che servono alla gestione della carta di credito o della polizza assicurativa.

- ii. Primitiva** – una classe non dovrebbe offrire implementazioni multiple dello stesso comportamento: ogni operazione dovrebbe offrire un singolo servizio primitivo e atomico. Per esempio, la classe ContoCorrente non dovrebbe avere operazioni per fare due o più depositi: basta richiamare più volte l'operazione deposita().

Le classi dovrebbero implementare l'insieme più semplice e più piccolo possibile di operazioni. A volte, per motivi di efficienza e prestazioni questa caratteristica

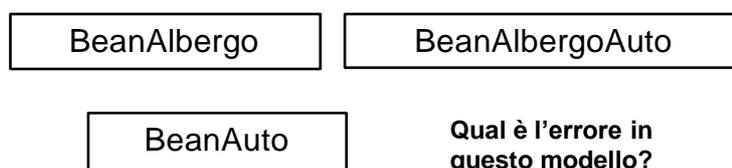
Ingegneria del Software

potrebbe essere resa meno forte. Per esempio, per ridurre i tempi si potrebbe pensare di offrire operazioni per fare più depositi.

ATTENZIONE: quando cerchi di ottimizzare le operazioni, chiediti sempre quante volte quelle operazioni verranno richiamate.

- iii. Alta coesione** – ogni classe dovrebbe modellare un singolo concetto astratto e dovrebbe avere un insieme di operazioni strettamente correlate che supportano l'intento della classe. Un'alta coesione garantisce riusabilità e manutenibilità.

Consideriamo un sistema di gestione delle vendite. Cosa è sbagliato nella figura seguente? Bean sono Enterprise Java Bean, cioè componenti software scritti in Java per piattaforma Enterprise Edition per applicazioni aziendali distribuite ed orientate alle transazioni. Un bean implementa logiche di business orientate alla riusabilità ed alla portabilità.



- Le classi sono nominate in modo sbagliato. Sarebbe più adatto PrenotazioneAlbergo e NoleggioAuto.
- Il prefisso Bean non è necessario;
- La classe BeanAlbergoAuto ha poca coesione;
- Non è né un modello di analisi (il suffisso Bean si riferisce ad aspetti implementativi) né un modello di progetto.

iv. **Basso accoppiamento** – una classe dovrebbe essere associata solamente con le classi che le permettono di realizzare le sue responsabilità.

Evita il problema dello “spaghetti code”, che rende il sistema incomprensibile e non-manutenibile.

Evita di fare connessioni tra classi solo perché una classe ha qualche codice che un'altra classe potrebbe usare. Non sacrificare l'integrità architetturale del sistema solo per risparmiare tempo di sviluppo.

Naturalmente, l'accoppiamento tra classi è desiderabile all'interno dello stesso sottosistema perché indica un'alta coesione nel sottosistema.

Ereditarietà

Nel workflow Analisi viene introdotta l'ereditarietà solo se c'è una relazione “is a” chiara e non ambigua. Nel workflow Progetto l'ereditarietà può anche essere usata per riusare codice. Nel seguito, l'ereditarietà verrà esaminata in dettaglio.

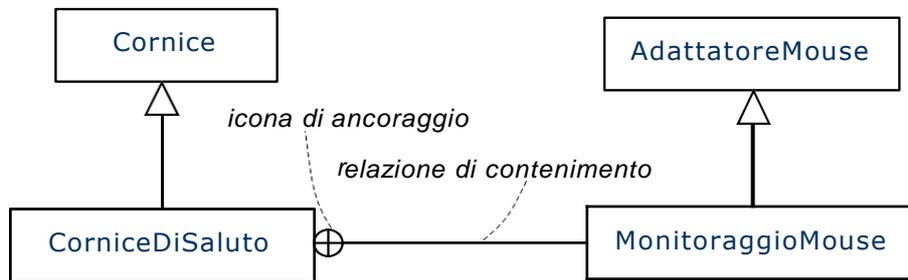
Aggregazione e Ereditarietà

Ricorda che:

- L'ereditarietà è la forma più forte di accoppiamento;
- L'incapsulamento è debole all'interno della gerarchia di classi – ogni modifica apportata alla classe base, si ripercuote su tutte le classi discendenti;
- L'ereditarietà è un tipo di relazione poco flessibile: in tutti i linguaggi di programmazione object-oriented l'ereditarietà è fissa a tempo di esecuzione.

Classi annidate

Alcuni linguaggi, quali Java, permettono di inserire una definizione di una classe nella definizione di un'altra classe, creando così una *classe annidata*. Una classe annidata è accessibile solo dalla classe che la contiene o da oggetti di tale classe.



Il monitoraggio del mouse è incapsulato all'interno della cornice di saluto.

Ogni istanza di **CorniceDiSaluto** contiene un'istanza di **MonitoraggioMouse** per processare gli eventi generati tramite il mouse.

Relazioni tra classi di progetto

NOTA BENE: non c'è nessun linguaggio di programmazione object-oriented che supporta associazioni bidirezionali, classi di associazione ed associazioni molti a molti.

Per creare un modello di progetto, va specificato come queste associazioni saranno realizzate.

Rifinire le associazioni di analisi in modo da ottenere le associazioni di progetto coinvolge diverse procedure:

- trasformare associazioni in relazioni di aggregazione o composizione dove appropriato;
- implementare associazioni uno-a-molti;
- implementare associazioni molti-a-uno;
- implementare associazioni molti-a-molti;
- implementare associazioni bidirezionali;
- implementare classi di associazioni.

Tutte le associazioni di progetto devono avere:

- navigabilità;
- molteplicità su entrambi i lati.

Aggregazione e Composizione

Aggregazione: è un tipo di relazione lieve tra oggetti – per esempio, un computer e le sue periferiche. Una periferica può essere condivisa tra computer e non è di proprietà di nessun particolare computer.

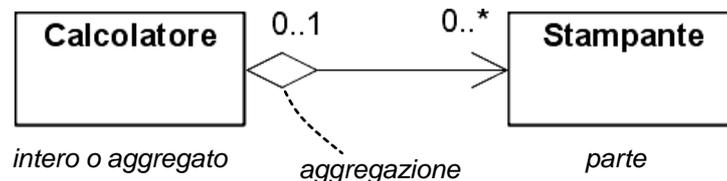
Composizione: è un tipo di relazione molto forte tra oggetti – per esempio, un albero e le sue foglie. Una foglia appartiene solamente ad un albero e quando l'albero muore anche le foglie muoiono.

Semantica dell'aggregazione

L'aggregazione è un tipo di relazione intero-parte in cui l'intero è fatto di molte parti. In questo tipo di relazioni un oggetto (l'intero) usa i servizi di un altro oggetto (la parte).

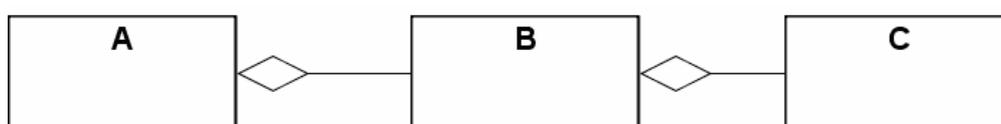
NOTA BENE: se la navigabilità è solo dall'intero alla parte, la parte non è consapevole di essere una porzione di un intero.

Ingegneria del Software



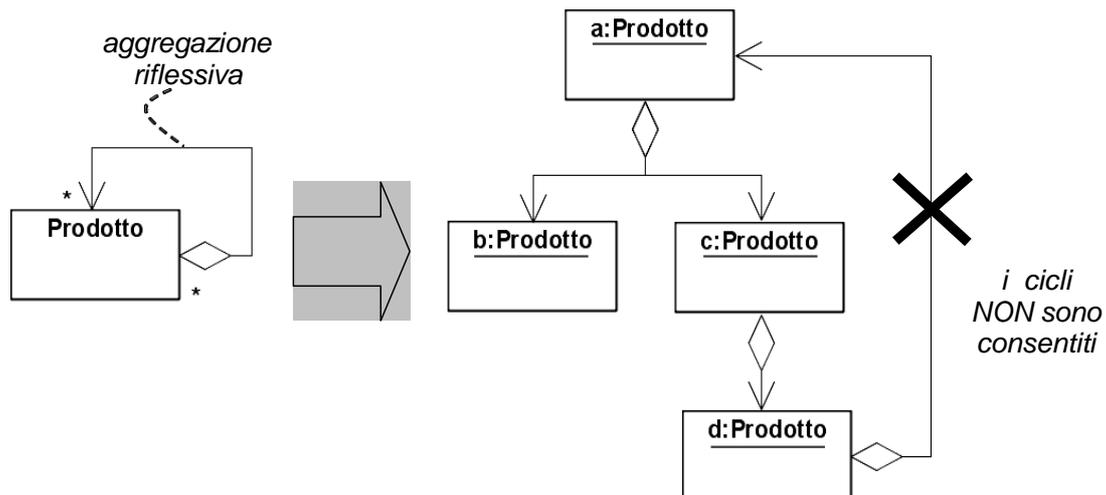
- l'aggregato può qualche volta esistere indipendentemente dalle parti, qualche volta no;
- le parti possono esistere indipendentemente dall'aggregato;
- l'aggregato è in qualche senso incompleto se qualcuna delle parti è mancante;
- è possibile che diversi aggregati condividano il possesso delle parti.

L'aggregazione è transitiva.



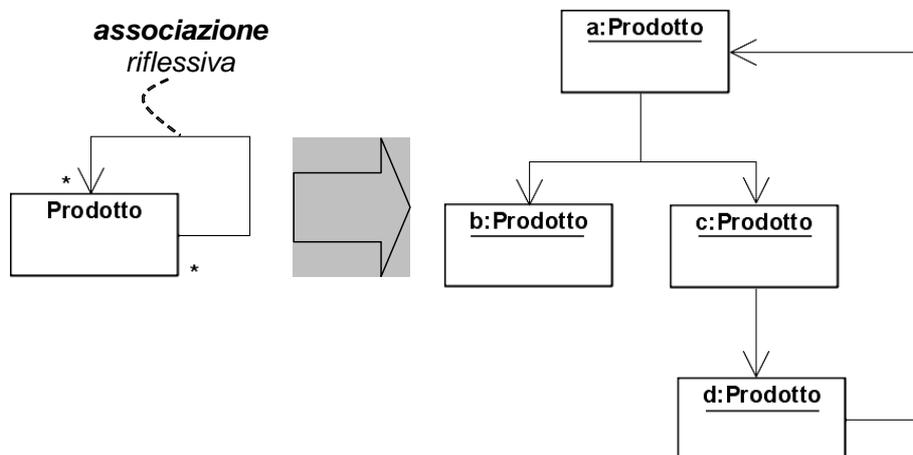
*transitività dell'aggregazione:
se C è parte di B e B è parte di A, allora C è parte di A*

L'aggregazione è asimmetrica: un oggetto non può mai o indirettamente o direttamente essere parte di sé stesso.

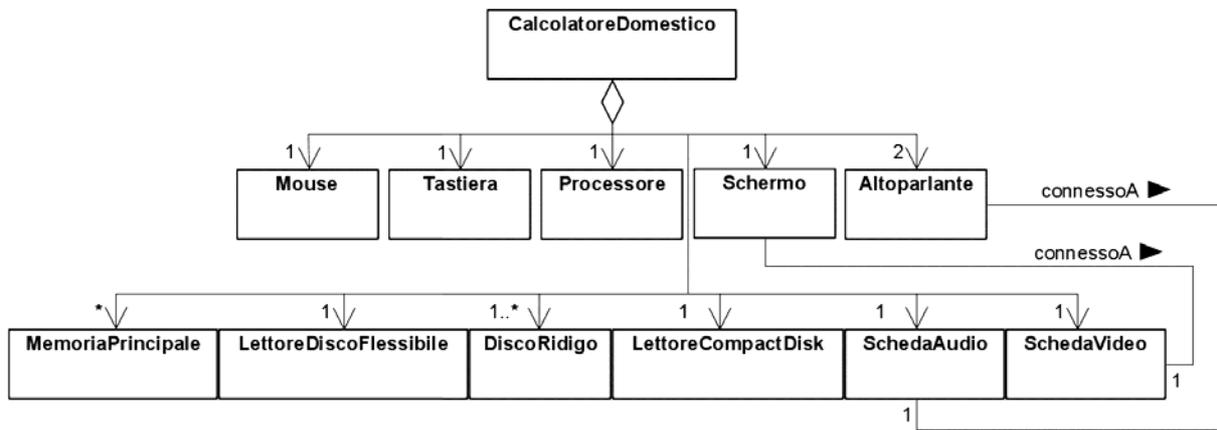


Per modellare il caso in cui l'oggetto d ha un collegamento con l'oggetto a, si può usare un'associazione riflessiva, non rifinita.

Ingegneria del Software



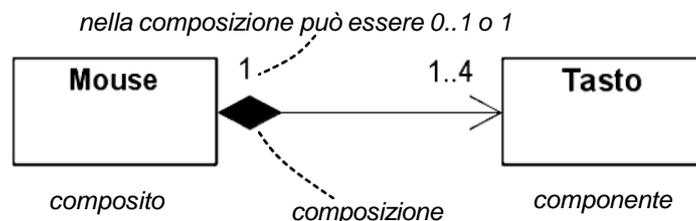
Ingegneria del Software



Semantica della composizione

La composizione è la forma più forte di aggregazione e ha una semantica simile.

Nella composizione le parti non hanno vita indipendente al di fuori dell'intero. Inoltre, ogni parte appartiene ad al più uno ed un solo intero, mentre nell'aggregazione una parte può essere condivisa tra interi.



Quindi,

- le parti possono solo appartenere ad un intero alla volta;
- l'intero ha la responsabilità della creazione e distruzione di tutte le sue parti;
- l'intero può rilasciare una parte, se la responsabilità della parte è assunta da un altro oggetto;
- se l'intero viene distrutto, deve o distruggere tutte le sue parti oppure dare la responsabilità di queste parti a qualche altro oggetto.

La semantica della composizione è molto simile alla semantica degli attributi. Perché usare gli attributi? Ci sono due ragioni

- Gli attributi possono essere tipi di dati primitivi, che non sono classi;

- Le classi come Tempo, Data, Stringa sono usate estensivamente: se dovessimo modellare una relazione di composizione per ogni classe, presto il modello sarebbe incomprensibile. È sicuramente più intuitivo modellarle come attributi che come classi.

ATTENZIONE: nel decidere se utilizzare un attributo o la composizione tieni sempre in mente la chiarezza, l'utilità e la leggibilità del modello.

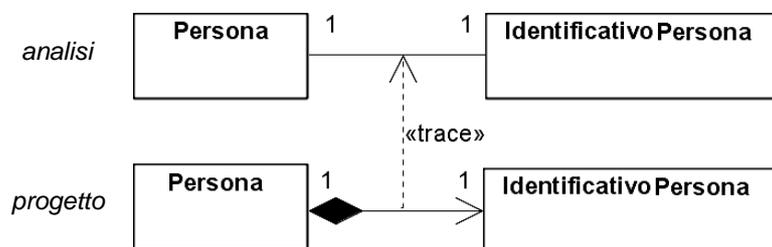
Come rifinire le relazioni di analisi.

In analisi, le relazioni sono semplicemente relazioni di associazione. Tranne nel caso in cui si viene a creare un ciclo nel grafo di aggregazione, le relazioni di associazione sono trasformate in relazioni di aggregazione o composizione. Dopo aver deciso in quale relazione trasformare una relazione di associazione

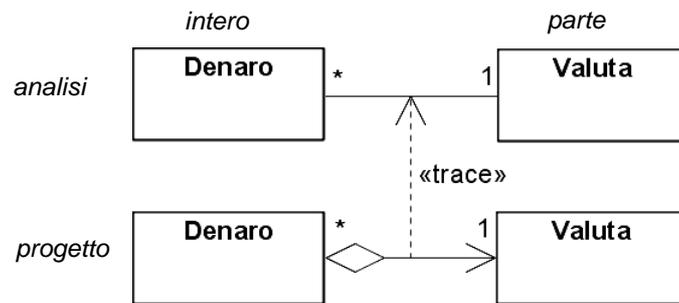
- Aggiungi le molteplicità ed i nomi dei ruoli alle associazioni, se sono assenti;
- Decidi quale lato dell'associazione è l'intero e quale la parte;
- Analizza la molteplicità del lato dell'intero – se è 0..1 o esattamente 1, puoi usare la composizione; altrimenti devi usare l'aggregazione;
- Aggiungi la navigabilità dall'intero alla parte – associazioni di progetto devono essere unidirezionali.

Ingegneria del Software

Associazioni uno-a-uno



Associazioni multi-a-uno



ATTENZIONE: controlla che non ci siano cicli nel grafo di aggregazione.

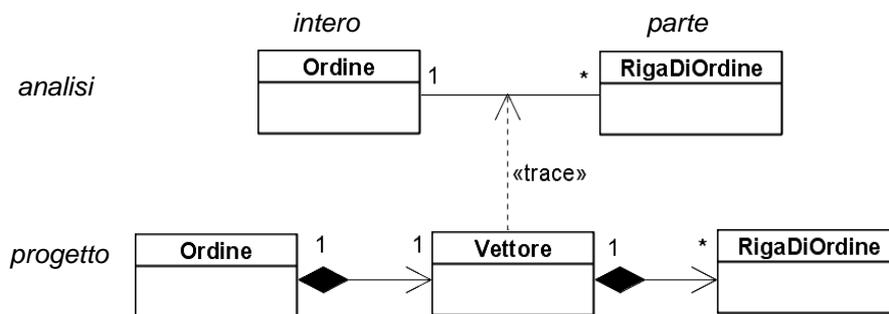
Associazioni uno-a-molti

Un'associazione uno-a-molti mostra che le parti sono una collezione di oggetti. Si dovrebbe usare o un supporto per le collezioni nativo del linguaggio o una classe collezione.

I linguaggi offrono tipicamente supporti predefiniti molto limitati (array).

Una classe collezione è una classe le cui istanze sono specializzate nel gestire collezioni di altri oggetti. Molti linguaggi propongono classi collezione nelle librerie standard.

Ingegneria del
Software



Ci sono quattro strategie fondamentali per modellare le collezioni:

1. Modellare la classe collezione esplicitamente (figura precedente).

Vantaggio: molto esplicito;

Svantaggio: molta confusione nel modello di progetto.

2. Dire allo strumento di modellazione come implementare ogni specifica associazione uno-a-molti. Molti strumenti che generano codice permettono di assegnare una specifica classe collezione ad ogni associazione uno-a-molti. Questo tipicamente viene realizzato aggiungendo dei valori etichettati all'associazione per specificare le proprietà di quella relazione per la generazione del codice.

Ingegneria del
Software

Realizzazioni dei casi d'uso nel Progetto

Rispetto all'attività corrispondente già svolta nel workflow Analisi, ci sono importanti differenze:

- Le realizzazioni dei casi d'uso nel progetto coinvolgono classi di progetto, interfacce e componenti piuttosto che classi di analisi;
- Il processo di creare realizzazioni di casi d'uso nel progetto probabilmente scoprirà requisiti non funzionali e nuove classi di progetto;
- Le realizzazioni dei casi d'uso aiutano a trovare quelli che Booch chiama i meccanismi centrali, cioè modi standard di risolvere un particolare problema di progetto che sono applicati consistentemente attraverso lo sviluppo del sistema.

Una realizzazione di un caso d'uso nel progetto è una collaborazione tra classi ed oggetti di progetto che realizzano un caso d'uso. Questa realizzazione specifica le decisioni di implementazione ed implementa i requisiti non funzionali.

Una realizzazione di un caso d'uso nel progetto consiste in **diagrammi di interazione** e **diagrammi delle classi**.

Diagrammi di interazione nel progetto

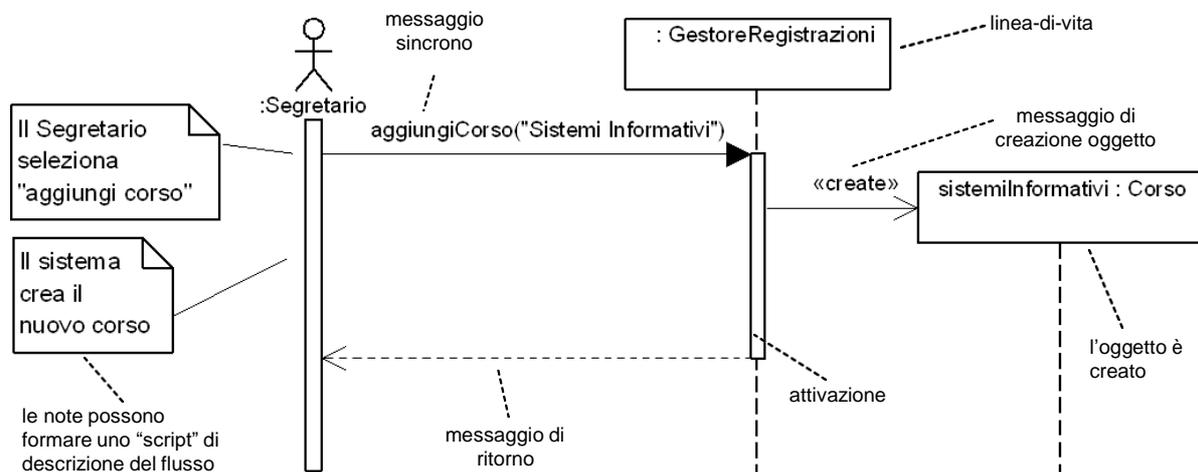
I diagrammi di interazione più usati nella realizzazione dei casi d'uso nel progetto sono sicuramente i diagrammi di sequenza.

I diagrammi di interazione nel progetto possono essere una semplice rifinitura dei diagrammi di interazione nell'analisi oppure diagrammi completamente nuovi costruiti per illustrare aspetti tecnici che sono sorti durante il progetto.

Nel progetto vengono introdotti un numero limitato di meccanismi centrali quali la persistenza degli oggetti, la distribuzione degli oggetti, le transazioni, ecc.. e vengono usati dei diagrammi di interazione per illustrare questi meccanismi.

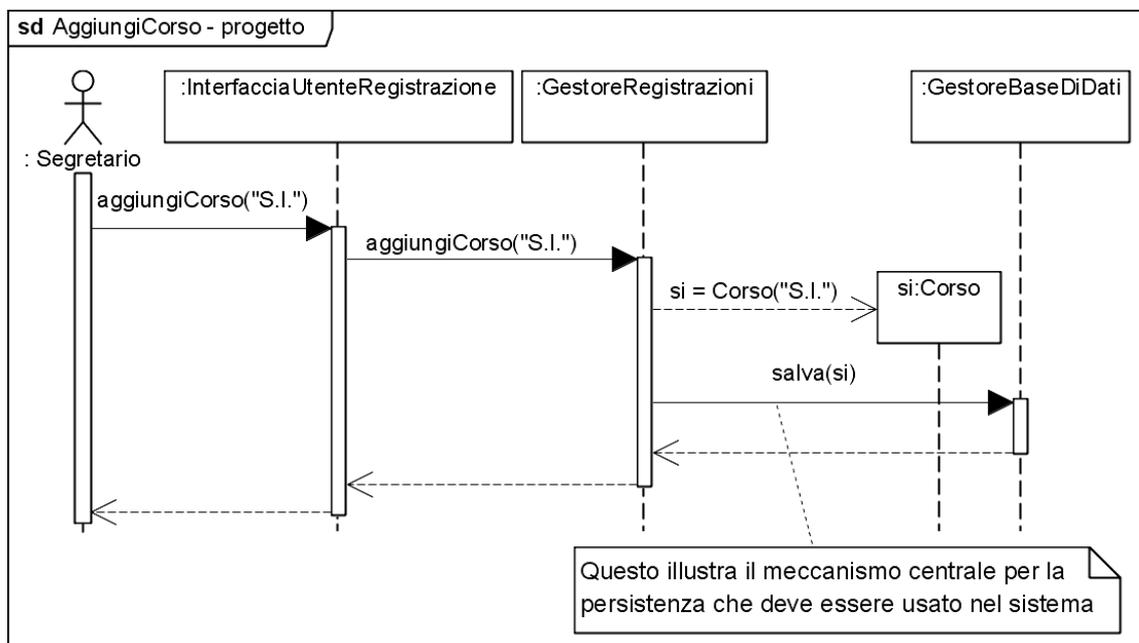
Per capire il ruolo dei diagrammi di sequenza, consideriamo il diagramma di sequenza introdotto nella analisi.

Caso d'uso: <i>AggiungiCorso</i>
ID: 8
Breve descrizione: <i>Aggiunge al sistema i dettagli di un nuovo corso</i>
Attori primari: <i>Segretario</i>
Attori secondari: <i>Nessuno</i>
Precondizioni: <i>1. Il Segretario ha effettuato il login nel sistema</i>
Flusso principale: <i>1. Il Segretario seleziona "aggiungi corso". 2. Il Segretario inserisce il nome del nuovo corso. 3. Il Sistema crea il nuovo corso</i>
Postcondizioni: <i>1. Un nuovo corso viene aggiunto nel sistema</i>
Flussi alternativi: <i>CorsoGiaEsistente</i>



Il diagramma di sequenza viene trasformato nei primi stadi del workflow Progetto come mostrato di seguito

Ingegneria del Software



Cosa è stato aggiunto?

1. l'interfaccia utente;
2. le operazioni sono state dettagliate in modo da permetterne l'implementazione (per esempio, la creazione di un oggetto);

Ingegneria del Software

3. per la persistenza degli oggetti, viene utilizzato un meccanismo molto semplice: il `GestoreRegistrazioni` utilizza un `GestoreBaseDiDati` per la memorizzazione degli oggetti. Questo meccanismo centrale, una volta definito, dovrebbe essere usato consistentemente attraverso il resto del progetto.

Ingegneria del Software

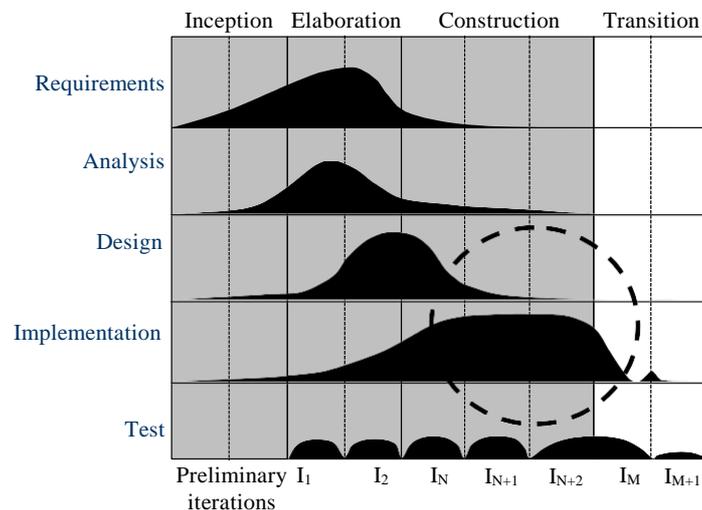
Mario G.C.A. Cimino - Antonio Luca Alfeo



Workflow Implementazione

Introduzione

L'obiettivo del workflow Implementazione è di trasformare un modello di progetto in codice eseguibile. Dal punto di vista dell'analista/progettista, lo scopo di questo workflow è di produrre un modello di implementazione, se richiesto. Questo modello richiede di allocare le classi di progetto ai componenti. Le modalità con cui questa allocazione è fatta dipendono dal linguaggio di programmazione



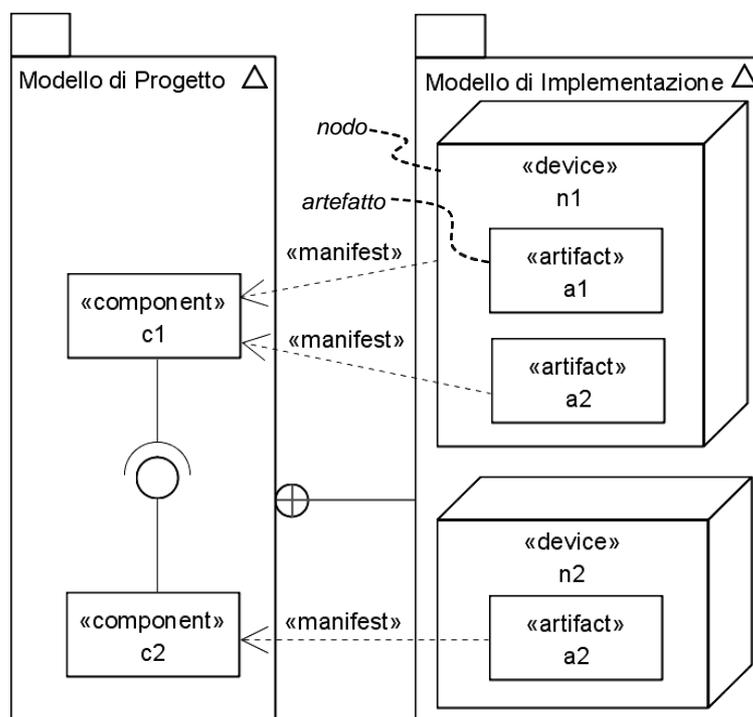
Ci sono due casi in cui è richiesta un'esplicita attività di modellazione dell'implementazione:

1. quando si vuole generare il codice direttamente dal modello;
2. se si sta realizzando uno sviluppo basato sul componente (component-based development) in modo tale da riusare componenti già disponibili: l'allocazione ai componenti di classi di progetto ed interfacce diviene un problema strategico.

La figura seguente mostra un meta-modello per il modello di implementazione.

Il modello di implementazione mostra come gli elementi di progetto sono realizzati tramite gli artefatti e come questi artefatti sono dislocati sui nodi. Gli artefatti rappresentano la specifica di cose reali quali file sorgenti, mentre i nodi rappresentano la specifica degli ambienti di esecuzione su cui quelle cose sono dislocate.

La figura seguente mostra la relazione tra modelli di progetto e modelli di implementazione.



La relazione «manifest» tra artefatti e componenti indica che gli artefatti sono rappresentazioni fisiche dei componenti.

La figura seguente mostra le attività principali del workflow Implementazione.

L'artefatto principale del workflow Implementazione è il modello di implementazione che consiste di:

- diagrammi di componenti che mostrano come gli artefatti rendono manifesti i componenti
- un nuovo tipo di diagramma, il diagramma di dislocazione (deployment diagram), che stabilisce in quali nodi gli artefatti verranno fisicamente dislocati, e le relazioni tra questi nodi.

La figura seguente mostra l'attività di implementazione architetturale. Questa attività consiste nell'identificare componenti significativi e mapparli in dispositivi hardware – in pratica consiste nella modellazione della struttura e della distribuzione fisica del sistema.

In principio, si potrebbe modellare la dislocazione fisica del sistema in maniera esaustiva. In pratica, i dettagli di dislocazione di molti componenti avranno poca importanza, a meno che si stia generando codice dal modello.

Diagramma di Dislocazione

La dislocazione è il processo di assegnare o artefatti a nodi o istanze di artefatti a istanze di nodi.

Il diagramma di dislocazione specifica i dispositivi hardware su cui il sistema sarà eseguito ed anche come il software è dislocato su questi dispositivi.

Il diagramma di dislocazione mappa l'architettura software creata nel progetto su un'architettura fisica che la esegue. In sistemi distribuiti, il diagramma modella la distribuzione del software sui nodi fisici.

Ci sono due forme di diagramma di dislocazione:

1. *forma descrittiva* (descriptor form) – contiene nodi, relazioni tra nodi ed artefatti. Un nodo rappresenta un dispositivo hardware (un PC). Un artefatto rappresenta un artefatto software tipo un file JAR (Java ARchive).
2. *forma istanza* (instance form) – contiene istanze di nodi, di relazioni e di artefatti. Un'istanza di nodo è per esempio il PC di Giovanni. Un'istanza di artefatto è per esempio una copia di FrameMaker usata per scrivere uno specifico file. Quando i dettagli di una specifica istanza non sono conosciuti, si possono usare istanze anonime.

Tipicamente, un primo diagramma di dislocazione in forma descrittiva viene creato alla fine del progetto come parte del processo di definizione dell'architettura hardware finale. Questo diagramma viene rifinito in uno o più diagrammi di dislocazione in forma istanza.

Quindi:

- nel workflow Progetto, il diagramma di dislocazione è focalizzato sui nodi e sulle relazioni tra questi nodi;
- nel workflow Implementazione, il diagramma di dislocazione è focalizzato sull'assegnare o artefatti a nodi o istanze di artefatti ad istanze di nodi.

Nodi

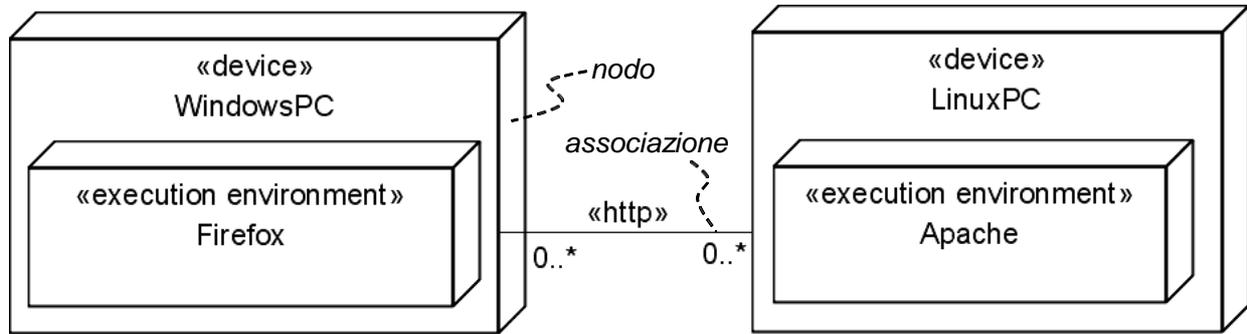
UML 2: “Un nodo rappresenta un tipo di risorsa computazionale su cui possono essere dislocati artefatti per l'esecuzione.”

Ci sono due stereotipi standard per i nodi:

- «device» - il nodo rappresenta un tipo di dispositivo fisico (per esempio, un PC);
- «execution environment» - il nodo rappresenta un tipo di ambiente di esecuzione per il software (per esempio, Apache).

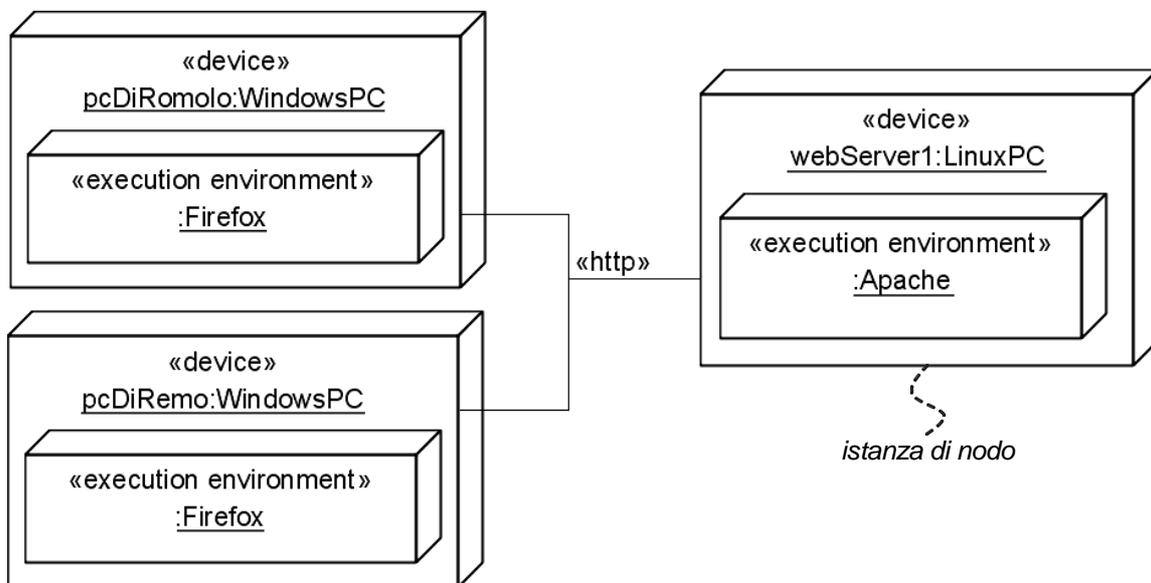
I nodi possono essere annidati in altri nodi.

La figura seguente presenta un diagramma di dislocazione in forma descrittiva. Come è pratica comune, nella figura sono stati inclusi nello stesso diagramma sia il tipo di hardware che l'ambiente di esecuzione.



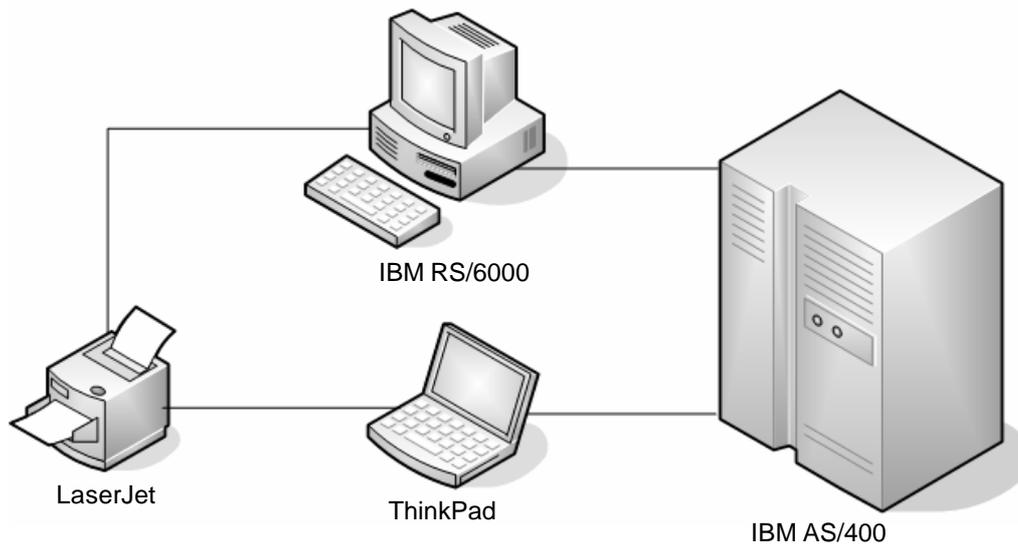
Un'associazione tra nodi rappresenta un canale di comunicazione grazie al quale l'informazione può passare avanti ed indietro.

La figura seguente mostra un diagramma di dislocazione in forma istanza. I nomi dei nodi in questo caso sono sottolineati.



I diagrammi di dislocazione sono probabilmente la parte di UML più stereotipata. La possibilità di assegnare proprie icone agli stereotipi permette di usare simboli che sono uguali ai dispositivi hardware reali, rendendo così il diagramma molto intuitivo.

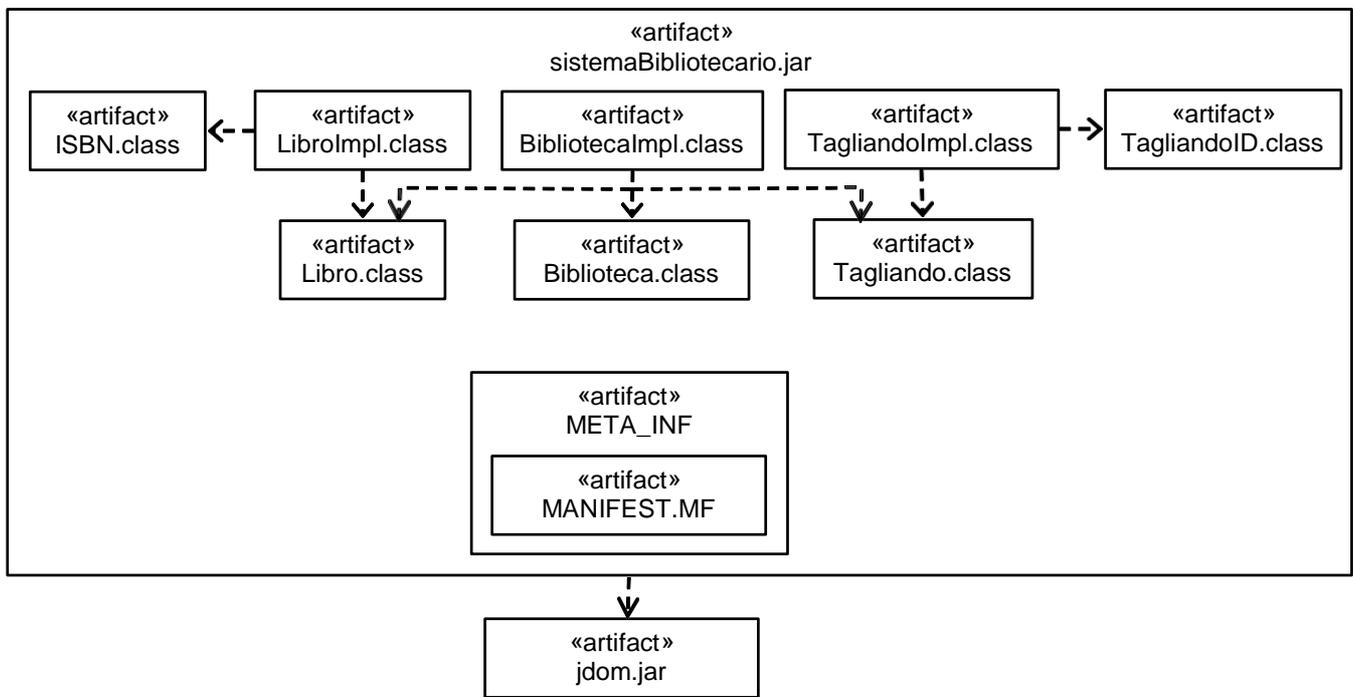
La figura seguente mostra un esempio di diagramma di dislocazione con l'uso di simboli appropriati per ogni stereotipo.



Artefatti

Un artefatto rappresenta la specifica di una cosa concreta del mondo reale quale il file sorgente `ContoCorrente.java`. Alcuni esempi di artefatti sono:

- file sorgenti
- file eseguibili
- script
- tabelle di basi di dati
- documenti
- risultati del processo di sviluppo.



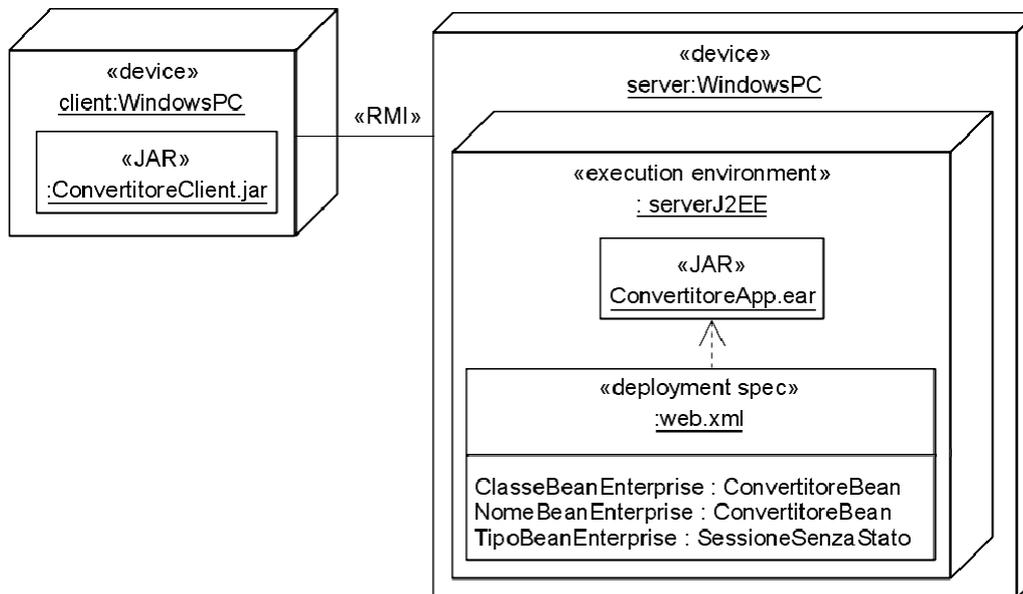
Il file chiamato MANIFEST.MF descrive i contenuti del JAR. Sebbene la figura precedente è corretta dalla prospettiva di UML, non è particolarmente descrittiva. Infatti, non è possibile capire per esempio che META_INF rappresenta un directory. A questo proposito UML mette a disposizione un piccolo numero di stereotipi di artefatto:

Stereotipi di Artefatto	Semantica
«file»	Un file fisico
«deployment spec»	Una specifica di dettagli di dislocazione
«document»	Un file generico che mantiene qualche informazione
«executable»	Un file eseguibile
«library»	Una libreria statica o dinamica
«script»	Uno script che può essere eseguito da un interprete
«source»	Un file sorgente

Questi stereotipi non sono comunque sufficienti per descrivere completamente la dislocazione. Tipicamente, per ogni specifica piattaforma si renderà necessario introdurre un profilo.

Dislocazione

La figura seguente mostra un diagramma di dislocazione in forma istanza, estratto dal tutorial Java (www.java.sun.com). Il diagramma si riferisce ad un'applicazione di conversione di denaro.



La specifica (:web.xml) attaccata all'artefatto contiene dei dettagli chiave riguardo alla dislocazione:

1. ClasseBeanEnterprise – classe che contiene la logica del bean;
2. NomeBeanEnterprise – nome che i clienti possono usare per accedere al bean;
3. TipoBeanEnterprise – tipo del bean.