

# Sistemi Informativi

F. Marcelloni – M.G. Cimino



## Unified Modeling Language e Unified Process Introduzione

Sistemi Informativi

1

### 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?

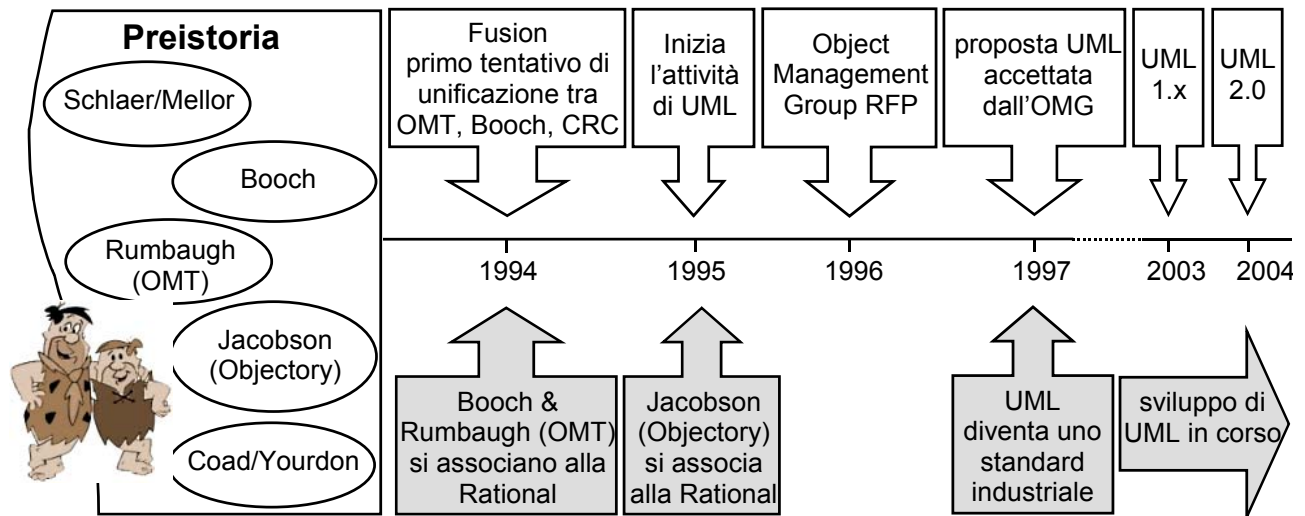
- i. UML fornisce una sintassi di modellazione visuale unica per l'intero ciclo di vita del software (dalle specifiche all'implementazione).
- ii. UML è stato usato per modellare differenti domini applicativi (da sistemi embedded a sistemi di supporto alle decisioni).
- iii. UML è indipendente dal linguaggio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

Sistemi Informativi

2

- 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. Come nasce UML?



### 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:

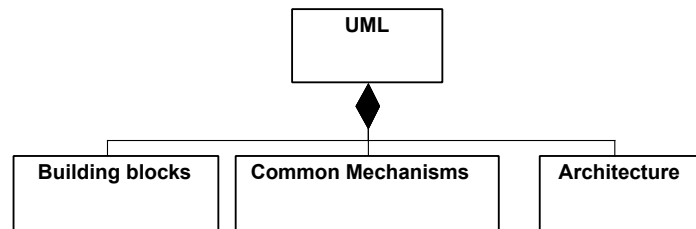
- ✓ la **struttura statica**, che descrive quali tipi di oggetti sono importanti per modellare il sistema e come essi sono connessi,
- ✓ il **comportamento dinamico**, che descrive il ciclo di vita di questi oggetti e come essi interagiscono l'un l'altro per compiere le funzionalità richieste.

# Struttura di UML

La struttura di UML consiste di:

- i. **Blocchi costituenti (building blocks)**: elementi, relazioni e diagrammi principali.
- ii. **Meccanismi comuni (common mechanisms)**: modi UML di raggiungere obiettivi specifici.
- iii. **Architettura (Architecture)**: vista UML dell'architettura di sistema.

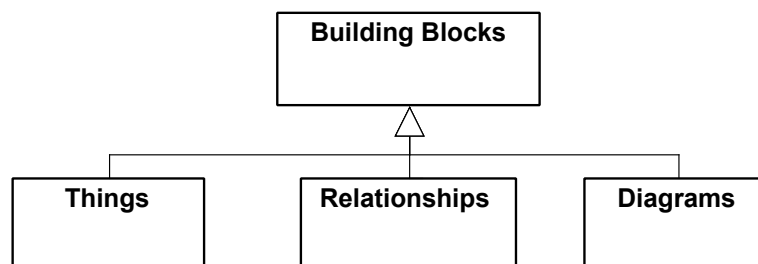
NOTA BENE: UML è stato progettato e modellato usando UML. UML è il meta-modello di UML.



## ***Blocchi costituenti***

I blocchi costituenti sono di tre tipi:








1. **entità** (things): elementi di modellazione;
2. **relazioni** (relationships): legami semantici tra entità;
3. **diagrammi** (diagrams): insieme di entità e relazioni che permettono di visualizzare “cosa fa” il sistema e “come lo fa”.



Le **entità** possono essere suddivise in:

1. **entità strutturali** (structural things): i sostantivi del modello, ossia classe, interfaccia, collaborazione, caso d'uso, oggetto, componente, nodo;
2. **entità comportamentali** (behavioral things): i verbi del modello, ossia interazioni, attività, macchine di stato;
3. **entità agglomeranti** (grouping things): il package, che è usato per raggruppare elementi del modello semanticamente correlati;
4. **entità notazionali** (annotational things): commenti che possono essere inseriti nel modello per catturare informazione ad hoc.

Le **relazioni** permettono di catturare connessioni significative tra entità. Le relazioni possono essere di dipendenza, associazione, aggregazione, composizione, contenimento, generalizzazione e realizzazione. La tabella seguente riassume i simboli utilizzati per ogni relazione.

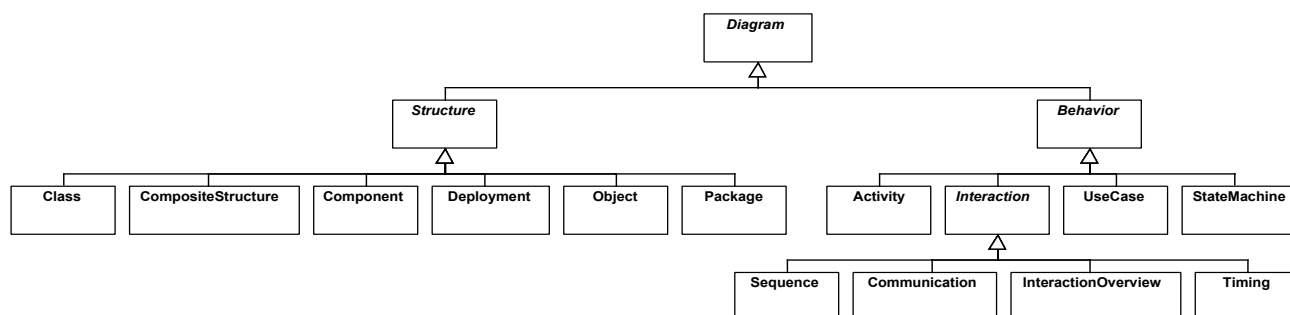
Tipo di relazione	sintassi UML sorgente destinatario	Semantica
Dipendenza		L'elemento sorgente dipende dall'elemento destinatario e può essere influenzato da cambiamenti di questo
Associazione		La descrizione di un insieme di collegamenti tra oggetti
Aggregazione		L'elemento destinatario è una parte dell'elemento sorgente
Composizione		Una forma di aggregazione forte (più vincolata)
Contenimento		L'elemento sorgente contiene l'elemento destinazione
Generalizzazione		L'elemento sorgente è una specializzazione del più generale elemento destinatario e può essere sostituito con questo
Realizzazione		L'elemento sorgente garantisce l'attuazione del contratto specificato dall'elemento destinatario

I **diagrammi** sono finestre o viste nel modello. Il modello è il contenitore di tutte le entità e relazioni che sono state create per descrivere il comportamento del sistema software che si sta cercando di progettare.

ATTENZIONE: per cancellare una entità od una relazione non basta rimuoverle da tutti i diagrammi, ma vanno rimosse dal modello.

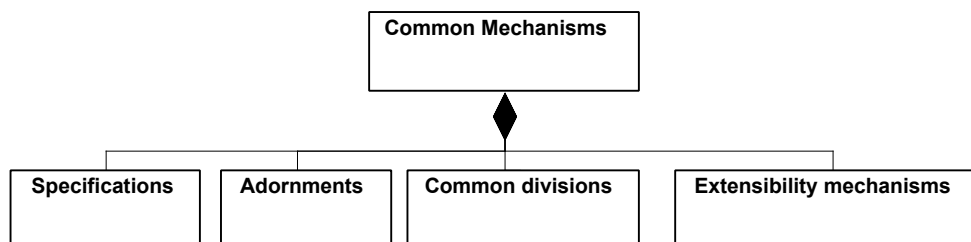
Ci sono 13 differenti tipi di diagrammi in UML, come rappresentato nella figura seguente. Nella figura, il testo in *italico* indica una categoria astratta.

In UML, non viene fissato nessun ordine specifico per creare e rifinire i diagrammi: spesso si lavora su diversi diagrammi in parallelo.



## Meccanismi comuni

UML ha quattro meccanismi comuni che vengono applicati consistentemente attraverso il linguaggio: *specifiche (specifications)*, *ornamenti (adornments)*, *suddivisioni comuni (common divisions)* e *meccanismi di estensione (extensibility mechanisms)*.



Le **specifiche** sono descrizioni testuali della semantica di un elemento. In generale, un modello UML ha almeno due dimensioni: quella grafica che permette di visualizzare il modello utilizzando diagrammi ed icone e quella testuale che è composta dalle specifiche dei vari elementi di modellazione.

Le specifiche permettono di catturare la semantica di ogni elemento, di tenere insieme il modello e di dare un significato al modello stesso. L'insieme di specifiche costituisce il supporto semantico (semantic backplane): i vari diagrammi non sono altro che viste o proiezioni visuali di questo supporto.

Un elemento UML si dice *nascosto* se compare solo a livello di supporto semantico, *fantasma* se compare solo a livello di diagrammi e non è specificato (errore comune dei principianti). Di conseguenza, si possono avere modelli:

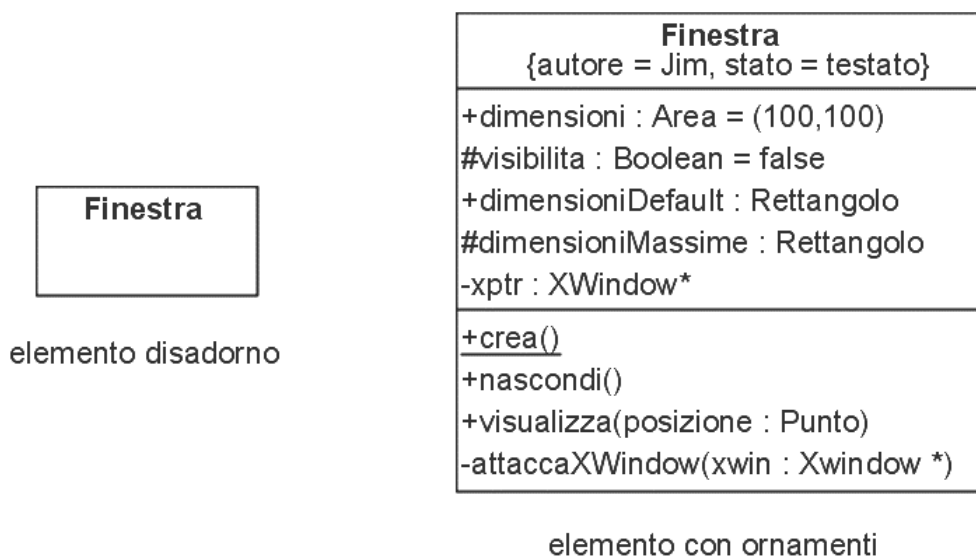
1. **elisi** (elided) – alcuni elementi sono nascosti;
2. **incompleti** (incomplete) – alcuni elementi del modello completamente mancanti;
3. **inconsistenti** (inconsistent) – il modello contiene contraddizioni.

L'obiettivo è di produrre via via modelli consistenti sempre più completi.

ATTENZIONE: “Death by diagrams” il modello contiene una quantità elevata di diagrammi, ma poche specifiche.

Gli **ornamenti** sono simboli grafici aggiuntivi rispetto al simbolo che descrive l'elemento e consentono di rendere visibili alcuni aspetti della specifica. Il meccanismo degli ornamenti permette di adattare la quantità di informazione visibile su un diagramma alle specifiche esigenze di modellazione.

NOTA BENE: Un diagramma dovrebbe contenere solo gli ornamenti strettamente necessari ad aumentarne la chiarezza e la leggibilità. Privilegia la leggibilità e la chiarezza piuttosto che la completezza.



Le **suddivisioni comuni** descrivono modi particolari di pensare ed interpretare il mondo. Due suddivisioni comuni:

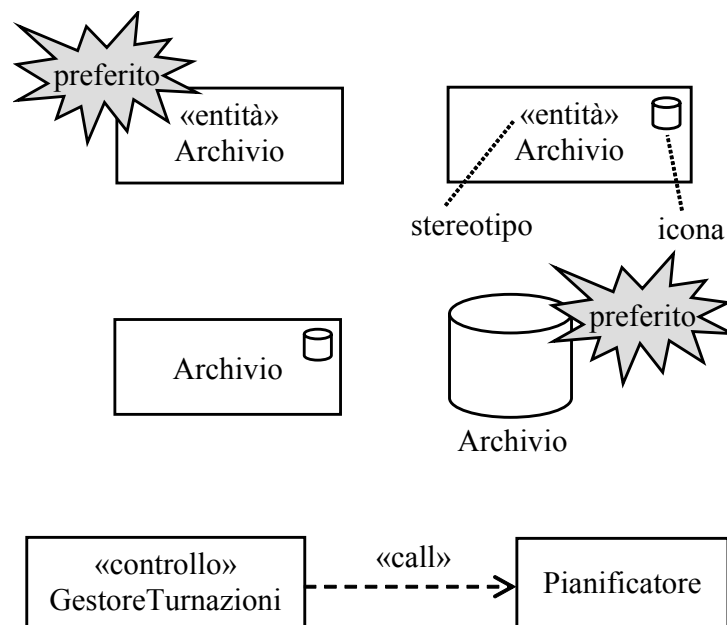
1. **classificatore/istanza.** Il **classificatore** è la nozione astratta di un tipo di elemento e l'**istanza** è uno specifico elemento concreto di tale tipo. Ad esempio, un classificatore può essere un tipo di conto corrente, mentre un'istanza è lo specifico conto corrente di Paolo Rossi. In UML, l'istanza è rappresentata con la stessa icona del classificatore, ma il nome è sottolineato. UML possiede in tutto 33 classificatori tra i quali i più comuni sono Actor, Class, Component, Interface, Node, Signal, Use Case.
2. **interfaccia/implementazione.** Questa suddivisione cerca di separare la descrizione di *cosa* un elemento è capace di fare da *come* lo fa effettivamente. L'**interfaccia** definisce l'insieme di servizi che l'elemento è in grado di fornire, mentre l'**implementazione** descrive come tali servizi vengono realizzati. In altre parole, l'interfaccia definisce il contratto a cui le specifiche implementazioni garantiscono di aderire. In object-oriented i due livelli sono nettamente separati.

I **meccanismi di estensione** consentono di adattare UML alle necessità presenti e future. UML incorpora tre semplici meccanismi di estensione: i vincoli, gli stereotipi ed i valori etichettati.

1. Un **vincolo** (constraint) è una stringa di testo racchiusa tra parentesi graffe che specifica alcune condizioni o regole che devono essere soddisfatte. In altre parole, vincola alcune caratteristiche dell'elemento. Tipicamente, i vincoli sono espressi usando lo Object Constraint Language, un linguaggio definito in UML.
2. Uno **stereotipo** (stereotype) rappresenta una variazione di un elemento di modellazione che mantiene la stessa forma (attributi e relazioni) ma assume un intento modificato. Gli stereotipi quindi permettono di introdurre nuovi elementi di modellazione basati su elementi esistenti. Questo può essere ottenuto appendendo il nome dello stereotipo racchiuso tra doppie parentesi angolari («nome dello stereotipo») all'elemento.

Ogni stereotipo può definire vincoli o valori etichettati.

È possibile associare una icona allo stereotipo, mantenendo preferibilmente il contorno nero e il riempimento bianco (evita l'uso dei colori). Dato che gli stereotipi introducono nuovi elementi di modellazione con differenti intenti, deve essere possibile definire la semantica di questi elementi. Tipicamente, la semantica è espressa mediante note.



3. Un **valore etichettato** (tagged value) permette di aggiungere delle proprietà (oltre a quelle predefinite) agli elementi di modellazione. La sintassi per i valori etichettati è la seguente: {tag1 = value1, tag2 = value2, ..., tagN=valueN}. Alcuni tag rappresentano informazioni supplementari applicate ad un elemento del modello; altri indicano proprietà di un nuovo elemento di modellazione definito da uno stereotipo.

Un insieme di vincoli, stereotipi e valori etichettati costituisce un **profilo**. I profili consentono di adattare UML a domini specifici. Esempio:

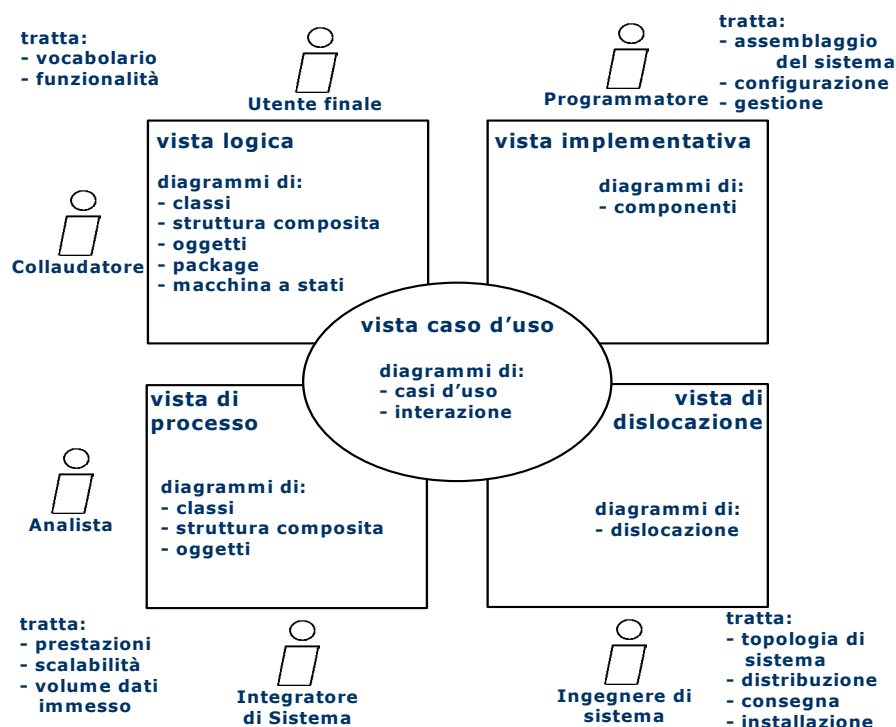
Stereotipo	Tag	Vincoli	Estende	Semantica
«NETComponent»	nessuno	nessuno	Component	rappresenta un componente nella piattaforma .NET
«NETProperty»	nessuno	nessuno	Property	rappresenta una proprietà di un componente
«NETAssembly»	nessuno	nessuno	Package	un impacchettamento a tempo di esecuzione di componenti .NET
«MSI»	nessuno	nessuno	Artifact	un file auto-installante di componenti
«DLL»	nessuno	nessuno	Artifact	un eseguibile portabile di tipo DLL
«EXE»	nessuno	nessuno	Artifact	un eseguibile portabile di EXE



## Architettura

L'architettura è la struttura organizzativa di un sistema, comprensiva della sua decomposizione in parti, la connettività tra queste parti, l'interazione, i meccanismi ed i principi guida che guidano il progetto del sistema. Gli aspetti essenziali di un'architettura sono catturati dalle "4+1" viste descritte da Philippe Kruchten:

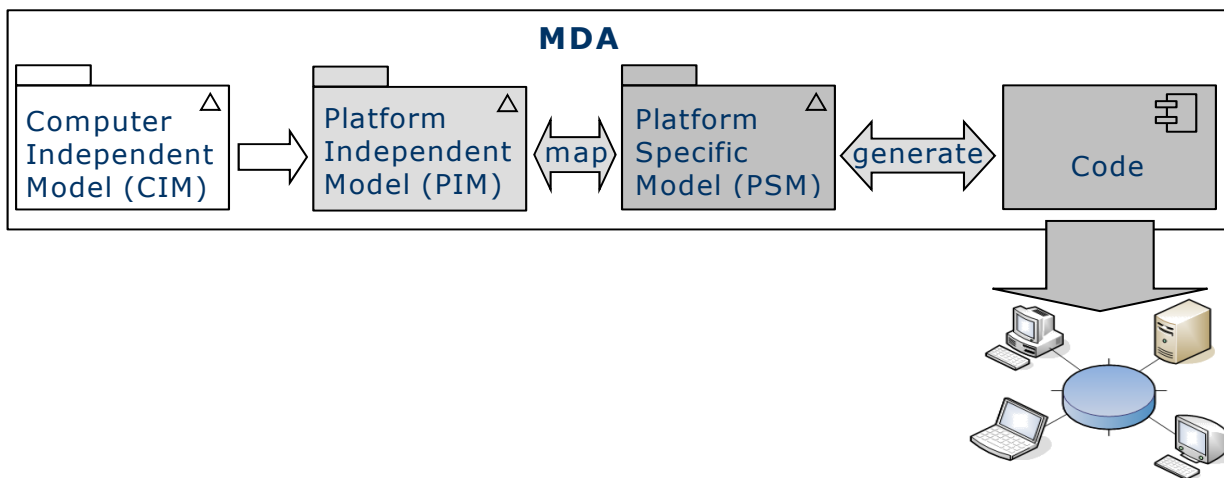
- **vista logica**, che cattura il vocabolario del dominio del problema come un insieme di classi ed oggetti, ponendo l'enfasi su come classi ed oggetti implementino il comportamento richiesto;
- **vista di processo**, che modella i thread ed i processi eseguibili in un sistema come classi attive (classi che hanno il proprio thread di controllo), fornendo una variante "process-oriented" della vista logica e mantenendo i medesimi artefatti;
- **vista implementativa**, che modella i file ed i componenti che costituiscono il codice fisico del sistema, illustrando le dipendenze tra componenti e la gestione delle configurazioni di insiemi di componenti per definire una versione del sistema;
- **vista di dislocazione**, che modella la dislocazione fisica di artefatti su un insieme di nodi fisici computazionali quali computer e periferiche.
- **vista caso d'uso**, che cattura i requisiti base del sistema come un insieme di casi d'uso, che forniscono la base per la costruzione delle altre viste.



La modellazione UML nella metodologia UP è un processo a rifinitura incrementale che, attraverso l'architettura "4+1" cattura le informazioni strettamente necessarie a costruire il sistema.

## Quale sarà il futuro di UML?

Il futuro di UML sarà Model Driven Architecture (MDA). MDA definisce una visione per come il software può essere sviluppato basato sui modelli: l'essenza di questa visione è che i modelli guidano la produzione di architetture software.



**Computer Independent Model (CIM):** modello ad un livello di astrazione molto alto che cattura le richieste chiave del sistema e il vocabolario del dominio del problema in un modo indipendente dal tipo di calcolatore. Modello della parte del business che ci si propone di automatizzare.

**Platform-Independent Model (PIM):** modello che esprime la semantica di business del sistema software indipendentemente dalla piattaforma sottostante (EJB, .NET, etc.).

**Platform-Specific Model (PSM):** PIM arricchito con informazioni specifiche della piattaforma di sistema in modo tale che possa essere generato codice eseguibile. Un PSM sufficientemente completo consente, in principio, di poter generare il 100% del codice sorgente e artefatti come documentazione, test, file e descrizioni d'uso. Per raggiungere questo obiettivo, il modello UML deve essere computazionalmente completo, cioè la semantica di tutte le operazioni deve essere specificata in un appropriato linguaggio (*action language*).

Alcuni strumenti di supporto a MDA forniscono già alcuni *action language* che sono ad un livello di astrazione più alto di linguaggi quali Java o C++.

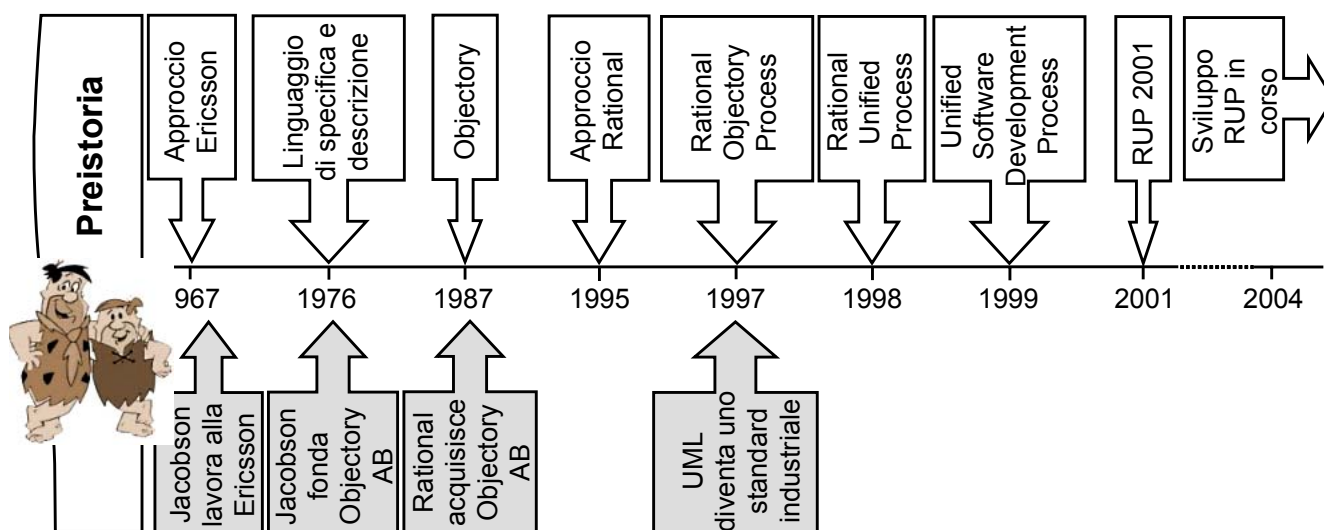
Nella visione MDA, il codice sorgente (Java, C++, C#) è proprio il codice macchina che risulta dalla compilazione di modelli UML. Questo codice è generato dal PSM.

## 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.



### Un po' di storia:



RUP è la versione di UP commercializzata da IBM, che acquisì la Rational Corporation nel 2003. Sia UP che RUP modellano il **chi**, il **cosa** ed il **quando**, ma lo fanno in maniera leggermente differente. La tabella seguente riassume come UP modella il chi, il cosa e il quando come degli stereotipi UML.

elemento UP	Semantica
 Worker	<b>Chi</b> – Un ruolo nel progetto, sostenuto da un individuo o un gruppo.
 Activity  Artifact	<b>Cosa</b> – Una unità di lavoro eseguita da un <i>Worker</i> (ruolo) o un artefatto prodotto nel progetto. Gli artefatti sono rappresentati come un documento generico: possono avere icone diverse a seconda di quello che rappresentano.
 Workflow Workflow Detail	<b>Quando</b> – Una sequenza di attività collegate che apportano valore al progetto. Possono essere spezzati in altri workflow, riferiti solo attraverso un nome.

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.

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:

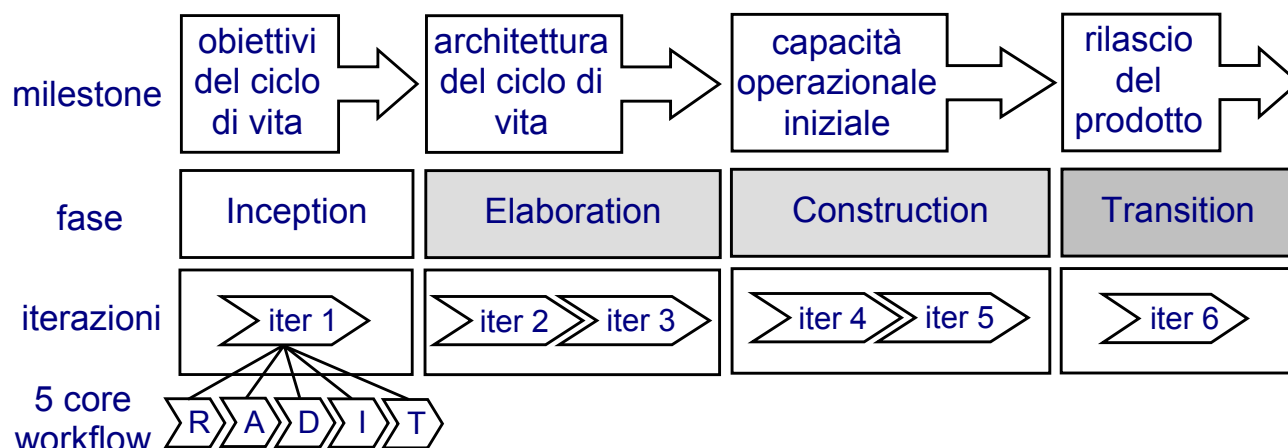
1. workflow requisiti (*requirements*): cattura che cosa il sistema dovrebbe fare;
2. workflow analisi (*analysis*): raffina e struttura i requisiti;
3. workflow progetto (*design*): realizza i requisiti attraverso un'architettura di sistema;
4. workflow implementazione (*implementation*): genera il software;
5. 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

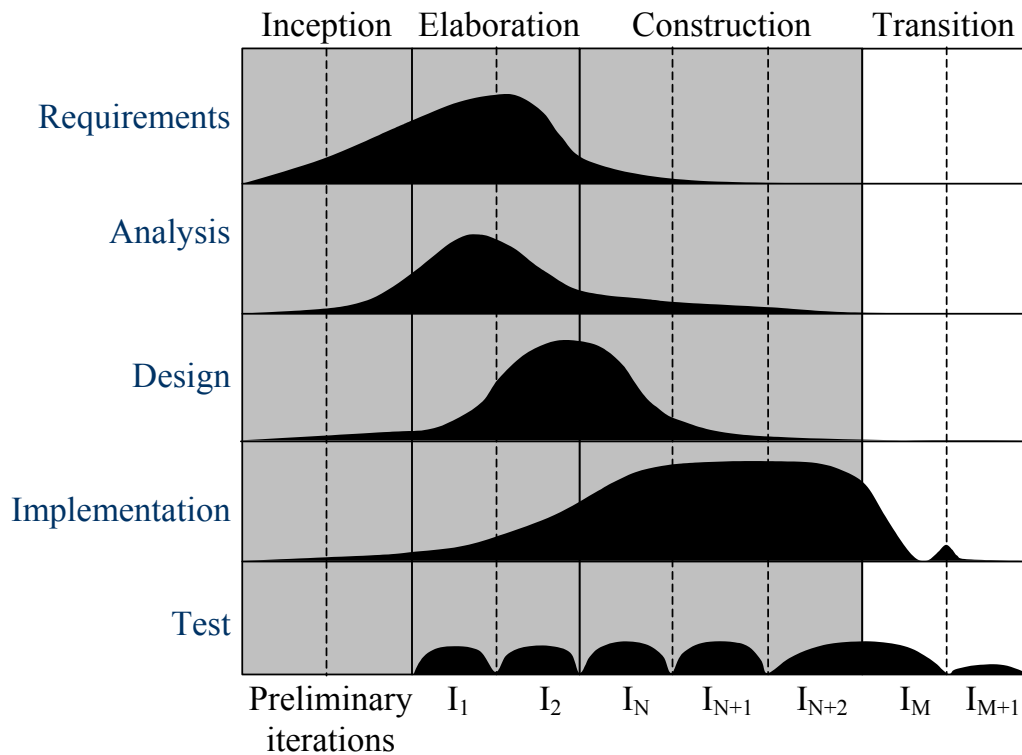
Il ciclo di vita del progetto è diviso in quattro fasi: Principio (*Inception*), Elaborazione (*Elaboration*), Costruzione (*Construction*) e Transizione (*Transition*)



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).

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.

<b>Condizioni</b>	<b>Deliverable</b>
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

## ***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.



**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.

<b>Condizioni</b>	<b>Deliverable</b>
È 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
È 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
È 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. È stato verificato il caso di business con riferimento al piano di progetto.	Un piano aggiornato di progetto  Caso di business
È 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.

**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.

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.

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

# Sistemi Informativi

F. Marcelloni – M.G. Cimino



## Il workflow Requisiti

### 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.

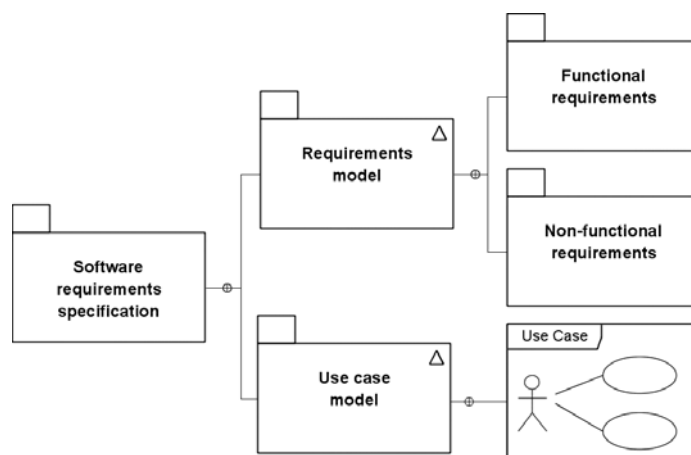
La maggior parte dell'attività richiesta da questo workflow è realizzata durante le fasi di Inception and Elaboration.

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.

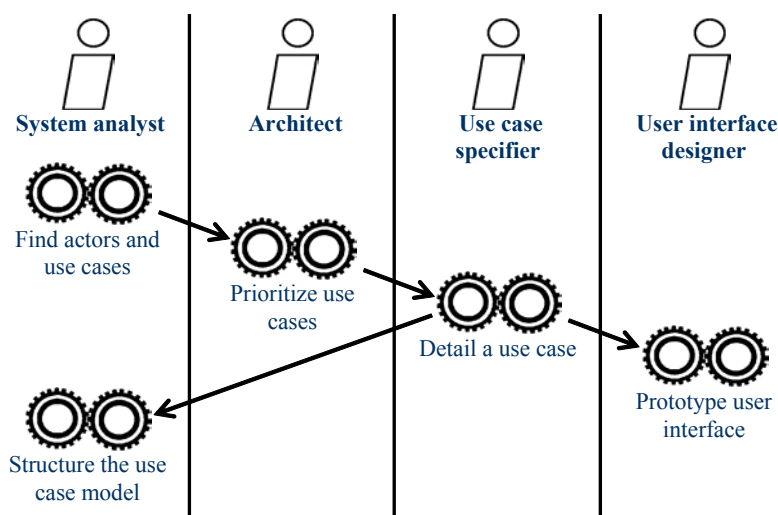
Vi sono in generale due tipi di requisiti: **funzionali** (che cosa il sistema farà) e **non funzionali** (vincoli sul sistema dovuti a prestazioni, affidabilità, etc.). I casi d'uso possono solo catturare i requisiti funzionali.

Il metamodello dell'approccio all'individuazione dei requisiti sia funzionali che non-funzionali è descritto nella figura seguente. Il metamodello presenta che la specifica dei requisiti si basa sia su un modello dei requisiti che su un modello dei casi d'uso.

I rettangoli denotano package. Il piccolo triangolo indica che il package contiene un modello.



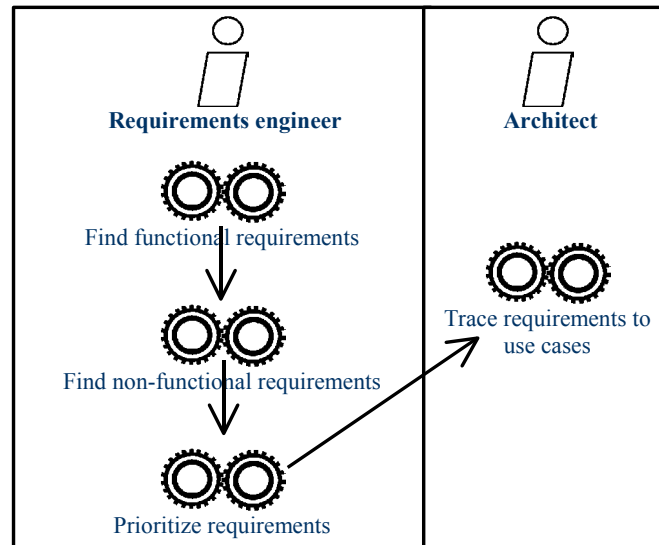
La figura seguente dettaglia le specifiche attività per il workflow Requirements nello UP standard. Le frecce indicano il flusso tipico di lavoro. Questa è comunque solo un'approssimazione (nella realtà alcune attività potrebbero essere fatte in un ordine differente).



In questo corso, siamo interessati alle seguenti attività:

- ✓ Trova attori e casi d'uso
- ✓ Dettaglia un caso d'uso
- ✓ Struttura il modello dei caso d'uso

Per trattare i requisiti non funzionali, faremo un'estensione al workflow Requirements dello UP, aggiungendo le attività ed il nuovo worker presentati nella figura seguente.



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.

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:**

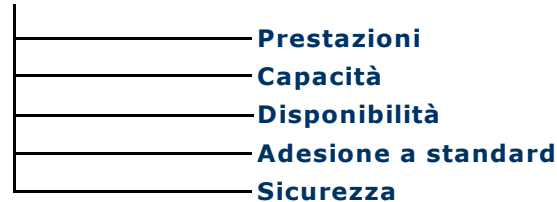
1. Il sistema ATM deve essere scritto in C++
2. Il sistema ATM deve comunicare con la banca usando un codice cifrato a 256-bit
3. Il sistema ATM deve convalidare la carta entro 3 secondi
4. Il sistema ATM deve convalidare il PIN entro 3 secondi

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**



**Requisiti non funzionali**



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
<b>Must have</b>	Requisito fondamentale per il sistema
<b>Should have</b>	Requisito importante che però può essere omesso
<b>Could have</b>	Requisito opzionale (da realizzare se c'è tempo).
<b>Want to have</b>	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).

RUP definisce l'insieme di attributi riportati nella tabella seguente:

<b>Attributo</b>	<b>Semantica</b>
Stato	Può avere uno dei seguenti valori: <i>Proposto</i> : requisito ancora in fase di discussione <i>Approvato</i> : requisito approvato per l'implementazione <i>Respinto</i> : requisito non approvato per l'implementazione <i>Incorporato</i> : requisito implementato in una particolare release
Beneficio	Può avere uno dei seguenti valori: <i>Critico</i> : il requisito <i>deve</i> essere implementato, altrimenti il sistema non sarà accettato dai committenti <i>Importante</i> : il requisito può essere tralasciato, ma ciò va contro l'usabilità del sistema ed il soddisfacimento dei committenti <i>Vantaggioso</i> : il requisito può essere tralasciato senza un impatto significativo sull'accettabilità del sistema
Sforzo	Una stima del tempo e delle risorse necessarie ad implementare il requisito, misurata in giorni-uomo o altre unità.
Rischio	<i>Alto, Medio o Basso</i> : il rischio insito nell'aggiungere questa caratteristica
Stabilità	<i>Alta, Media o Basso</i> : una stima della probabilità che il requisito cambi in qualche modo
Release	La versione che dovrebbe implementare il requisito

## ***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.

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

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.

## 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

Questionari sono utili supplementi alle interviste. ATTENZIONE: i questionari non dovrebbero essere utilizzati prima di fare delle interviste altrimenti ci si potrebbe trovare nella situazione paradossale di dover predisporre un questionario senza sapere quali siano le

domande corrette. I questionari sono particolarmente adatti ad ottenere risposte a domande specifiche.

Possono essere utilizzati per validare la propria comprensione delle specifiche.

### **Gruppi di lavoro**

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. La procedura è la seguente:

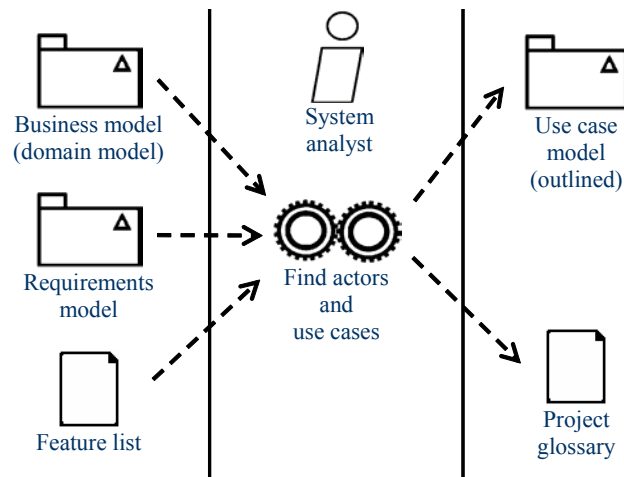
- Spiega che è un brainstorm
  - tutte le idee sono accettate come buone, registrate e non dibattute;
- Chiedi ai membri del gruppo di dare un nome ad ogni requisito del sistema
  - Scrivi ogni requisito su un foglietto adesivo;
  - Appendi i foglietti ad una lavagna;
- Discuti ogni requisito con il gruppo, inserisci gli attributi e confrontali.

## **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. Concentriamoci sull'attività "Find actors and use case" del workflow Requirements, descritta nella figura seguente.



Il modello del business o del dominio applicativo può essere o non essere presente. Se presente è sicuramente una sorgente eccellente per i casi d'uso.

Il modello dei requisiti è stato descritto precedentemente. I requisiti funzionali suggeriranno casi d'uso ed attori, mentre i requisiti non-funzionali suggeriranno informazioni da tenere in mente quando verranno definiti i casi d'uso.

La lista delle caratteristiche è un insieme di requisiti candidati raccolti nel documento preliminare di visione d'insieme stilato all'inizio del workflow Requirements.

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.

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;

- 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;

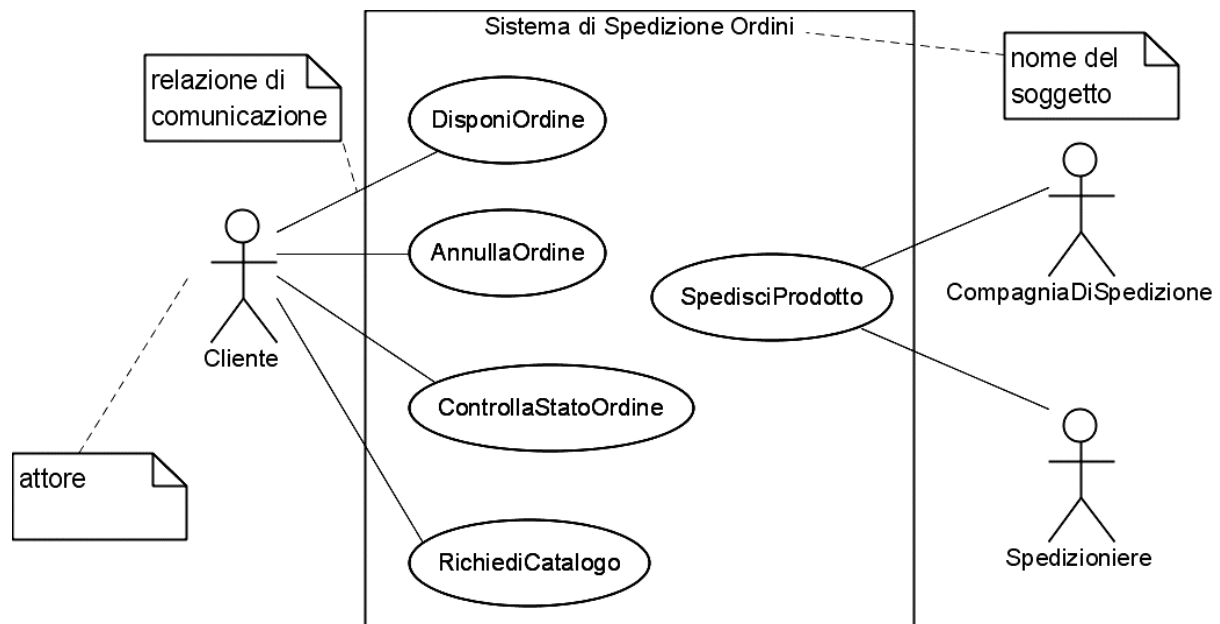
- considera come ogni attore usa il sistema;
- determina una lista di casi d'uso candidati;
- individua ogni caso d'uso attraverso una frase con un verbo (DisponiOrdine, CancellaOrdine, etc.);
- Se necessario, introduci nuovi attori.

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.

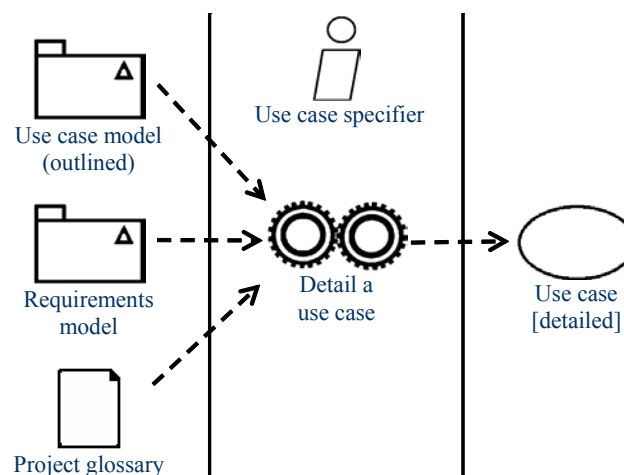
I termini usati nel modello UML devono essere consistenti con i termini definiti nel glossario. Tipicamente, i glossari vengono scritti in forma testuale su un file. Per progetti complessi, possono essere organizzati in semplici basi di dati. Di seguito viene dato un esempio di glossario



<b>Catalogo</b>	elenco illustrato dei prodotti in vendita sinonimi: <i>listino, tariffario</i> omonimi: nessuno
<b>Cliente</b>	persona fisica o giuridica che compra prodotti o servizi sinonimi: <i>compratore, acquirente, consumatore</i> omonimi: nessuno
<b>Carta di credito</b>	una carta che può essere usata per pagare i prodotti sinonimi: <i>carta</i> omonimi: nessuno ...

## Dettagliare i casi d'uso

Dopo aver creato i diagrammi dei casi d'uso ed identificato gli attori ed i casi d'uso principali, ogni caso d'uso deve essere specificato in dettaglio. La figura seguente descrive come l'attività richiesta per dettagliare i casi d'uso viene svolta. Il risultato di questa attività sono casi d'uso più dettagliati (almeno il nome e la specifica).



UML non definisce nessun formato standard per la specifica dei casi d'uso. È importante comunque che un'organizzazione abbia un formato standard. Il modello seguente è usato comunemente.

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).	<b>Use case:</b> <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,...	<b>ID:</b> 1
Paragrafo che cattura l'obiettivo del caso d'uso	<b>Brief description:</b> <i>Paga la tassa di vendita all'autorità fiscale alla fine del trimestre d'esercizio</i>
Attori che avviano il caso d'uso	<b>Primary actors:</b> <i>tempo</i>
Attori che interagiscono con il caso dopo l'avvio	<b>Secondary actors:</b> <i>autorità fiscale</i>
Vincoli sullo stato del sistema prima dell'avvio del caso d'uso (condizioni booleane)	<b>Preconditions:</b> <i>1. È la fine del trimestre d'esercizio</i>
Passi elencati come flusso di eventi	<b>Main flow:</b> <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)	<b>Postconditions:</b> <i>1. L'autorità fiscale riceve il corretto importo della tassa di vendita</i>
Eventuali alternative	<b>Alternative flows:</b>

## 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:

*<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*  
    n.1 *do something.*  
    n.2 *do something else.*  
    n.3 ...
- n+1. **ELSE**  
    n+1.1 *do something.*
- n+2.

Use case: <i>GestisciCarrello</i>
<b>ID:</b> 2
<b>Brief description:</b> <i>Il cliente cambia la quantità di un oggetto nel carrello</i>
<b>Primary actors:</b> <i>Cliente</i>
<b>Secondary actors:</b> <i>Nessuno</i>
<b>Preconditions:</b> <i>1. Il contenuto del carrello è visibile</i>
<b>Main flow:</b> <i>1. Il caso d'uso inizia quando il cliente seleziona un oggetto nel carrello</i> <i>2. IF il cliente seleziona "delete item"</i> <i>2.1 Il sistema rimuove l'oggetto dal carrello</i> <i>3. IF il cliente digita una nuova quantità</i> <i>3.1 Il sistema aggiorna la quantità dell'oggetto nel carrello</i>
<b>Postconditions:</b> <i>Nessuna</i>
<b>Alternative flows:</b> <i>Nessuno</i>

## Parola chiave FOR

### Sintassi:

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

Use case: <i>TrovaProdotto</i>
<b>ID:</b> 3
<b>Brief description:</b> <i>Il sistema trova alcuni prodotti che soddisfano i criteri di ricerca del cliente e li visualizza</i>
<b>Primary actors:</b> <i>Cliente</i>
<b>Secondary actors:</b> <i>Nessuno</i>
<b>Preconditions:</b> <i>Nessuna</i>
<b>Main flow:</b> <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> <i>5. IF il sistema trova dei prodotti</i> <i>5.5 FOR ogni prodotto trovato</i> <i>5.5.1 Il sistema visualizza una descrizione concisa del prodotto</i> <i>5.5.2 Il sistema mostra il prezzo del prodotto</i> <i>6. ELSE</i> <i>5.5 Il sistema comunica al cliente che nessun prodotto è stato trovato</i>
<b>Postconditions:</b> <i>Nessuna</i>
<b>Alternative flows:</b> <i>Nessuno</i>

## Parola chiave **WHILE**

Sintassi:

- n. **WHILE** *boolean expression*  
    n.1 *do something*  
    n.2 *do something else*  
    n.3 ...  
n+1.

Use case: <i>MostraDettagliAzienda</i>
<b>ID:</b> 4
<b>Brief description:</b> <i>Il sistema mostra i dettagli dell'azienda al cliente</i>
<b>Primary actors:</b> <i>Cliente</i>
<b>Secondary actors:</b> <i>Nessuno</i>
<b>Preconditions:</b> <i>Nessuna</i>
<b>Main flow:</b> <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> <i>3. WHILE il cliente osserva i dettagli dell'azienda</i> <i>3.1 il sistema suona una musica in sottofondo</i> <i>3.2 il sistema visualizza le offerte speciali</i>
<b>Postconditions:</b> <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>
<b>Alternative flows:</b> <i>Nessuno</i>

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>*

<b>Use case:</b> <i>CreaNuoviAccount</i>
<b>ID:</b> 5
<b>Brief description:</b> <i>Il sistema crea un nuovo account al cliente</i>
<b>Primary actors:</b> <i>Cliente</i>
<b>Secondary actors:</b> <i>Nessuno</i>
<b>Preconditions:</b> <i>Nessuna</i>
<b>Main flow:</b> 1. <i>Il caso d'uso inizia quando il cliente seleziona "create new customer account".</i> 2. <b>WHILE</b> <i>i dati del cliente non sono validi</i> 2.1 <i>Il sistema chiede al cliente di inserire i propri dati: indirizzo e-mail, password e password di nuovo per conferma</i> 2.2 <i>Il sistema controlla i dati del cliente</i> 3. <i>il sistema crea un nuovo account per il cliente</i>
<b>Postconditions:</b> 1. <i>Un nuovo account viene creato</i>
<b>Alternative flows:</b> <i>IndirizzoEMailNonValido</i> <i>PasswordNonValida</i> <i>Cancella</i>

<b>Alternative flow:</b> <i>CreaNuoviAccount : IndirizzoEMailNonValido</i>
<b>ID:</b> 5.1
<b>Brief description:</b> <i>Il sistema informa il cliente che ha inserito un indirizzo di e-mail non valido</i>
<b>Primary actors:</b> <i>Cliente</i>
<b>Secondary actors:</b> <i>Nessuno</i>
<b>Preconditions:</b> <i>Il cliente ha inserito un indirizzo e-mail non valido</i>
<b>Alternative flow:</b> 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>
<b>Postconditions:</b> <i>Nessuna</i>

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.

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

Requirement	Use case				
		UC <sub>1</sub>	UC <sub>2</sub>	UC <sub>3</sub>	UC <sub>4</sub>
	R <sub>1</sub>	X			
	R <sub>2</sub>		X	X	
	R <sub>3</sub>			X	
	R <sub>4</sub>				X
	R <sub>5</sub>	X			

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.

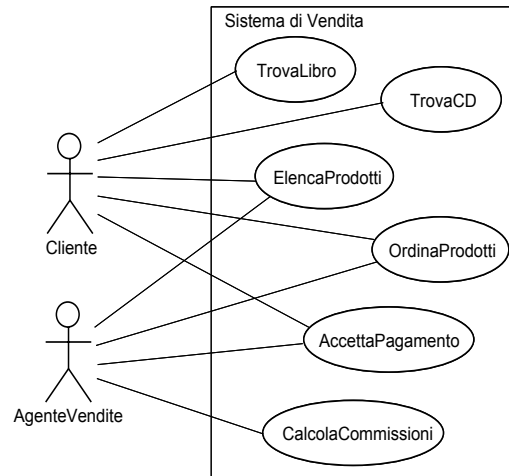


## La modellazione avanzata basata sui casi d'uso

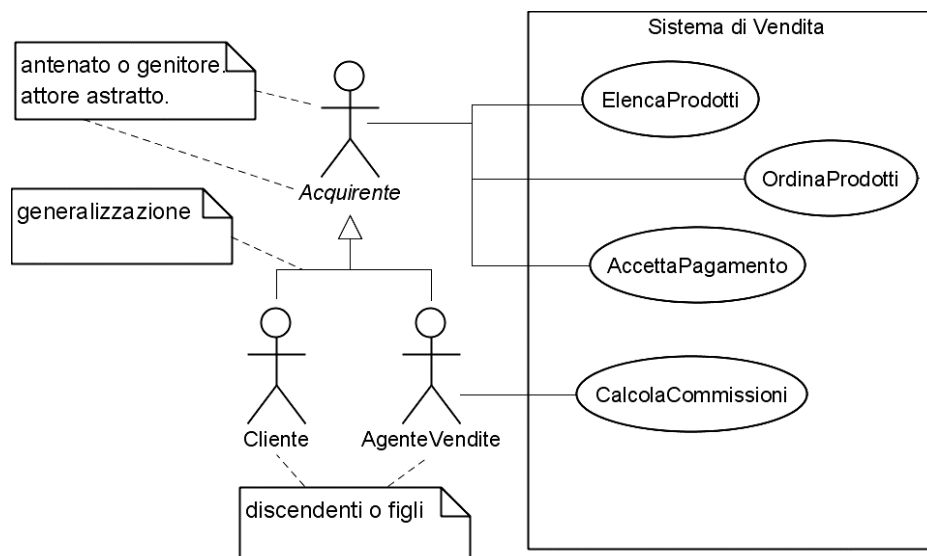
Per migliorare la chiarezza e semplificare la semantica, si possono adoperare le **relazioni** tra casi d'uso o tra attori. Nel seguito, discuteremo queste relazioni.

### 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.



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.



## 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.

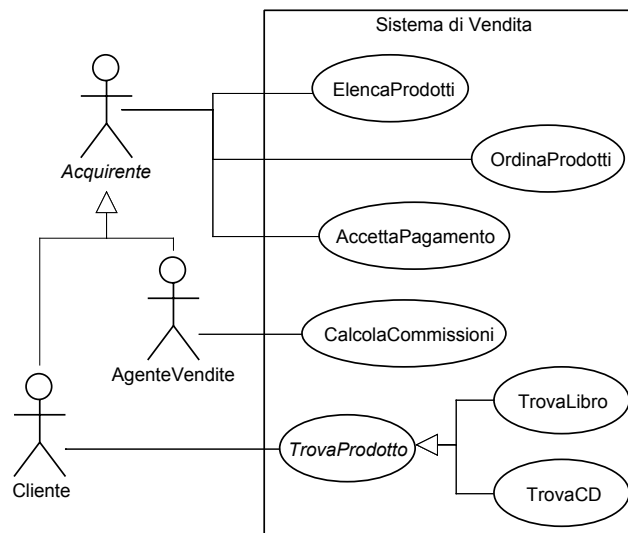
Come estendere o modificare i passi dei flussi? Due regole:

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:

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*:

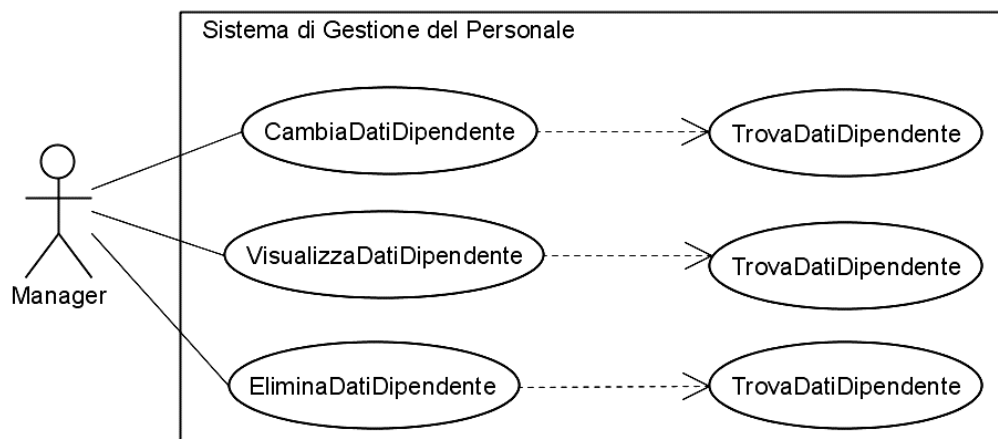
Use case: <i>TrovaLibro</i>
<b>ID:</b> 6
<b>Parent ID:</b> 3
<b>Brief description:</b> <i>Il sistema trova alcuni libri che soddisfano i criteri di ricerca del cliente e li visualizza</i>
<b>Primary actors:</b> <i>Cliente</i>
<b>Secondary actors:</b> <i>Nessuno</i>
<b>Preconditions:</b> <i>Nessuna</i>
<b>Main flow:</b> <ol style="list-style-type: none"> <li>(o1.) Il caso d'uso inizia quando il cliente seleziona "find book".</li> <li>(o2.) Il sistema chiede al cliente i criteri di ricerca, comprendenti autore, titolo, ISBN o argomento.</li> <li>(3.) Il cliente inserisce i criteri di ricerca.</li> <li>(o4.) Il sistema ricerca i libri che soddisfano i criteri di ricerca.</li> <li>(o5.) <b>IF</b> il sistema trova dei libri               <ol style="list-style-type: none"> <li>Il sistema visualizza il best seller corrente</li> <li>(o5.1) Il sistema visualizza i dettagli di un massimo di 5 libri</li> <li><b>FOR</b> ogni libro sulla pagina                   <ol style="list-style-type: none"> <li>Il sistema visualizza il titolo, l'autore, il prezzo e lo ISBN</li> </ol> </li> <li><b>WHILE</b> ci sono ancora libri                   <ol style="list-style-type: none"> <li>Il sistema dà al cliente l'opzione di visualizzare la prossima pagina di libri</li> <li>visualizza il titolo, l'autore, il prezzo e lo ISBN</li> </ol> </li> </ol> </li> <li>(6.) <b>ELSE</b> <ol style="list-style-type: none"> <li>Il sistema visualizza il best seller corrente</li> <li>(o6.1) Il sistema comunica al cliente che nessun libro è stato trovato</li> </ol> </li> </ol>
<b>Postconditions:</b> <i>Nessuna</i>
<b>Alternative flows:</b> <i>Nessuno</i>

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.

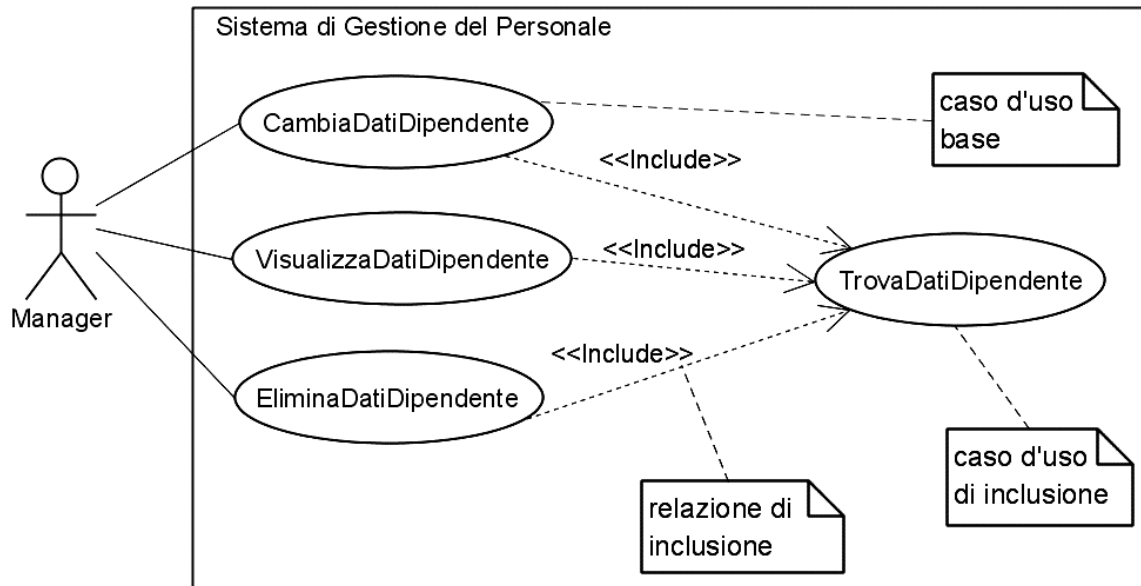
### **La relazione <<include>>**

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.

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

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

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

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

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>>**

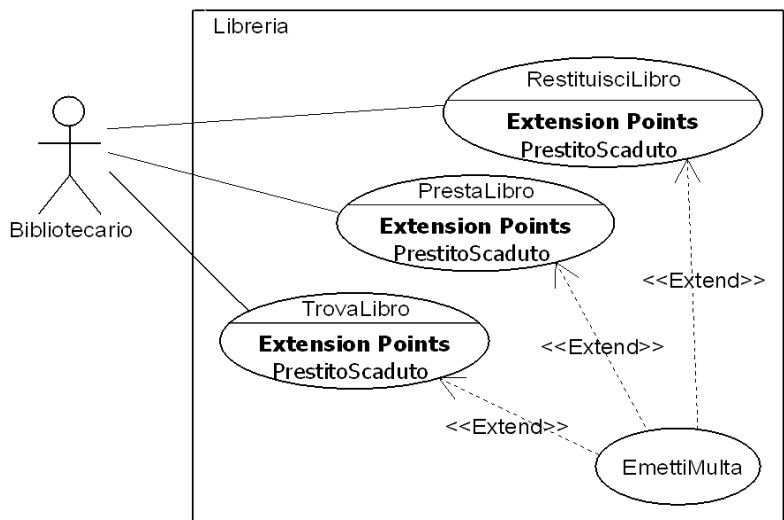
La relazione «**extend**» permette di inserire un nuovo comportamento in un caso d'uso esistente. È anch'essa una relazione di dipendenza stereotipata ma, sebbene possa essere pensata come una invocazione di funzione similmente alla inclusione, presenta differenze semantiche rispetto a quest'ultima.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

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:



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

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

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

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

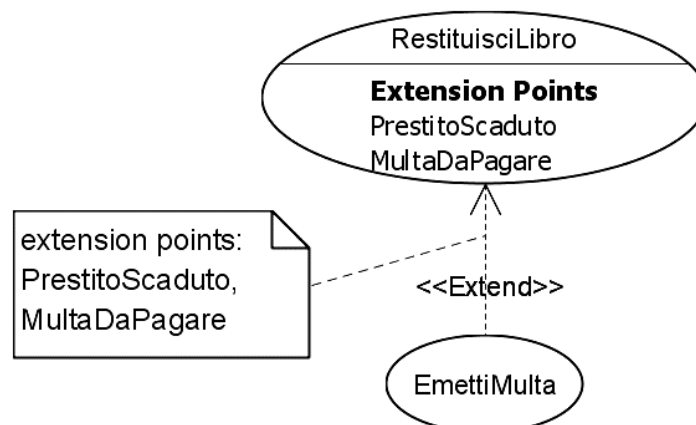
Vengono applicate le regole seguenti:

- 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.

### ***Segmenti di inserzione multipli***

I segmenti di inserzione multipli possono rendersi necessari quando l'estensione non può essere catturata in modo pulito con un solo segmento in quanto per esempio dopo la prima estensione si deve tornare al flusso principale del caso d'uso base. Per esempio, nell'esempio del caso d'uso RestituisciLibro, si può supporre che dopo aver stampato la multa si torni al flusso principale per verificare se ci siano altri libri con prestito scaduto e quindi dare alla persona la possibilità di pagare totalmente le multe emesse. La figura seguente mostra il diagramma dei casi d'uso in questo caso.



Il caso d'uso di estensione viene modificato nel modo seguente:

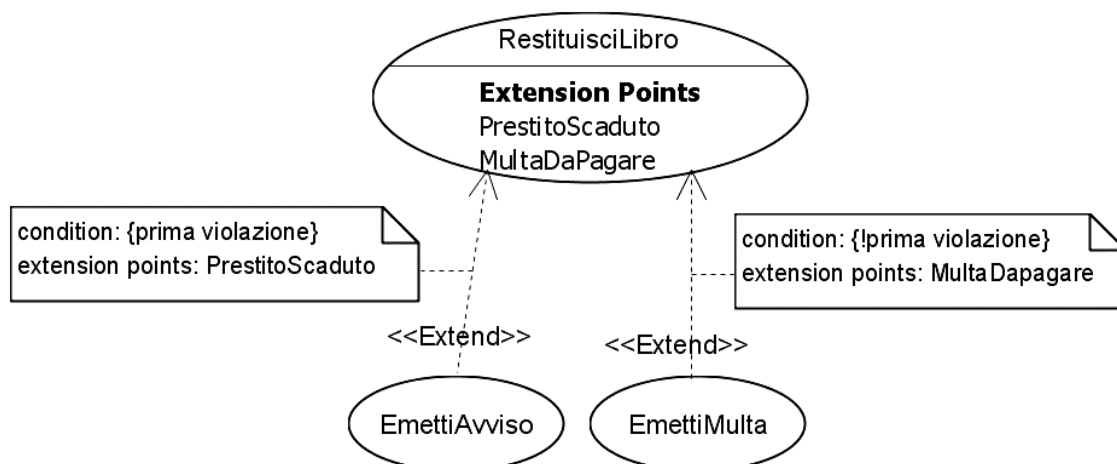


Extension Use case: <i>EmettiMult</i>
<b>ID:</b> 10
<b>Brief description:</b> Segment 1: <i>Il Bibliotecario memorizza e stampa la multa.</i> Segment 2: <i>Il Bibliotecario accetta il pagamento per la multa.</i>
<b>Primary actors:</b> <i>Bibliotecario.</i>
<b>Secondary actors:</b> <i>Nessuno.</i>
<b>Segment 1 Preconditions:</b> <i>Il prestito del libro restituito è scaduto.</i>
<b>Segment 1 flow:</b> 1. <i>Il Bibliotecario inserisce i dettagli della multa nel sistema.</i> 2. <i>il sistema stampa la multa.</i>
<b>Segment 1 Postconditions:</b> 1. <i>La multa è stata memorizzata nel sistema.</i> 2. <i>il sistema ha stampato la multa.</i>
<b>Segment 2 Preconditions:</b> <i>Il pagamento di una multa deve essere effettuato.</i>
<b>Segment 2 flow:</b> 1. <i>Il Bibliotecario accetta il pagamento per la multa.</i> 2. <i>il Bibliotecario inserisce che la multa è stata pagata nel sistema.</i> 3. <i>il sistema stampa una ricevuta.</i>
<b>Segment 2 Postconditions:</b> 1. <i>la multa è stata memorizzata come pagata.</i> 2. <i>il sistema ha stampato la ricevuta.</i>

Il numero dato ai segmenti determina l'ordine con cui i segmenti vengono inseriti nei punti di inserzione.

### Estensioni condizionali

La figura seguente presenta una biblioteca più benevola. In caso di prestito scaduto, prima viene dato un avvertimento e quindi viene emessa la multa. Questo scenario può essere implementato con le estensioni condizionali. La condizione è un'espressione booleana e l'inserzione avviene se e solo se tale espressione è vera.



Di seguito viene data la specifica del caso d'uso di estensione EmettiAvviso

Extension Use case: <i>EmettiAvviso</i>
<b>ID:</b> 11
<b>Brief description:</b> Segment 1: <i>Il Bibliotecario emette un avviso.</i>
<b>Primary actors:</b> <i>Bibliotecario.</i>
<b>Secondary actors:</b> <i>Nessuno.</i>
<b>Segment 1 Preconditions:</b> <i>Il prestito del libro restituito è scaduto.</i>
<b>Segment 1 flow:</b> 1. <i>Il Bibliotecario inserisce i dettagli dell'avviso nel sistema.</i>
<b>Segment 1 Postconditions:</b> 1. <i>l'avviso è stato memorizzato nel sistema.</i>

### *Quando conviene utilizzare le caratteristiche avanzate?*

In generale, è meglio adoperare le relazioni di generalizzazione, inclusione ed estensione, solo se semplificano e aumentano la chiarezza del modello. Bisogna inoltre considerare che molti utenti dei casi d'uso non sono esperti di informatica, per cui possono avere difficoltà a comprendere queste estensioni. Quindi, **non badare alla eleganza del modello, ma alla chiarezza.**

Non di rado i progetti vengono specificati da una eccessiva quantità di documentazione (“**bisogno di carta**”). Una buona norma è far rientrare il flusso principale di un caso d'uso in un foglio (mediamente mezzo foglio). Altrimenti o il testo è troppo complicato (tipico della lingua italiana), o il caso d'uso si può spezzare in più casi d'uso, oppure si possono creare più flussi alternativi.

**Ricorda:** il caso d'uso esprime **cosa** il sistema deve fare per gli attori non come deve farlo (che viene espresso nel progetto). Nell'esempio seguente, lo scrittore aveva già immaginato un qualche tipo di interfaccia (una scheda con un bottone OK), per cui il caso d'uso non è una pura dichiarazione di requisiti, ma un progetto primitivo:

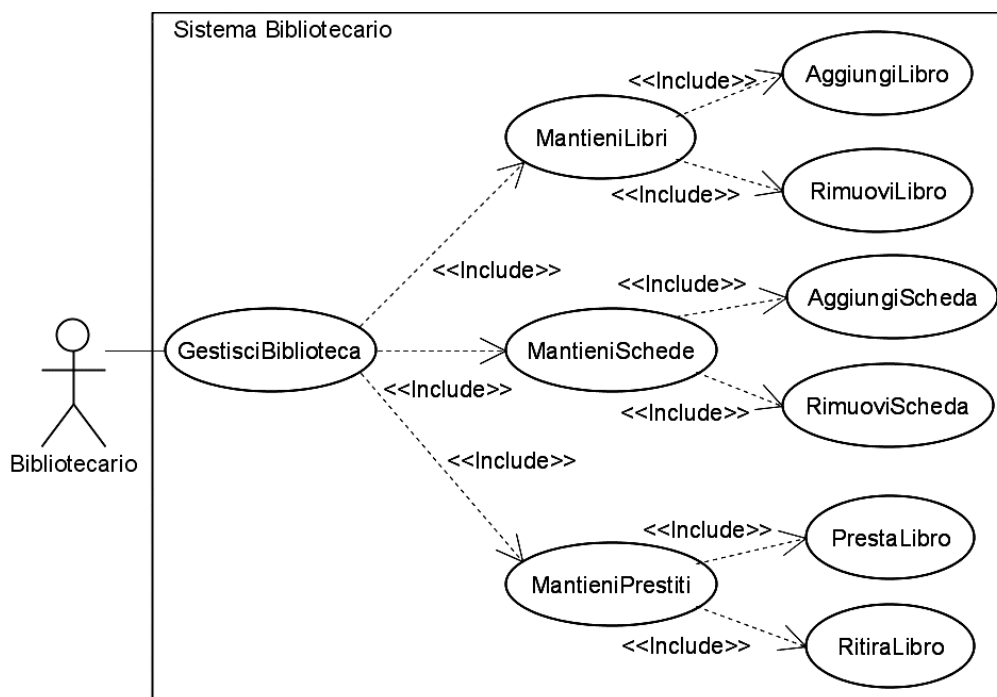
4. Il sistema chiede al Cliente di confermare l'ordine
5. Il cliente preme il bottone OK

Un modo migliore sarebbe stato:

5. ~~Il cliente preme il bottone OK~~ → Il cliente accetta l'ordine

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. Questo è tipico delle vecchie tecniche di programmazione procedurale.

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.



# Sistemi Informativi

F. Marcelloni – M.G. Cimino



## Workflow Analisi

101

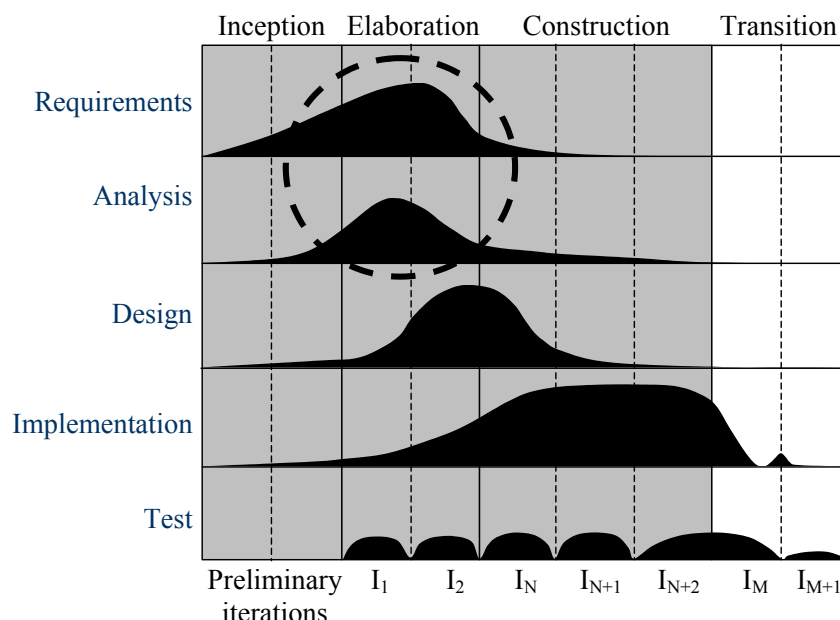
### Introduzione

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.

102

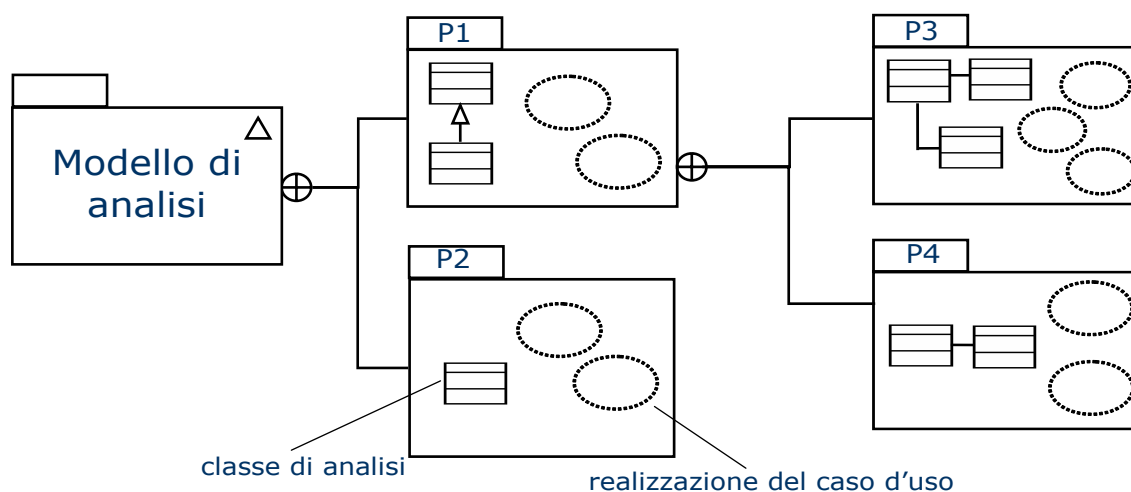


Gli artefatti prodotti dal workflow Analisi sono:

- **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.

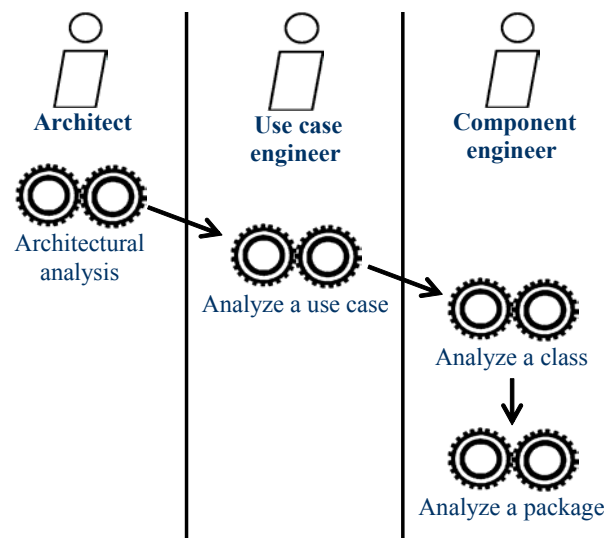
103

La figura seguente presenta il metamodello del modello di analisi. Il modello contiene package di analisi che racchiudono classi e realizzazioni di casi d'uso.



La figura seguente mostra le attività principali che compongono il workflow Analisi.

104



## Modello di analisi - Regole Empiriche

- ✓ Individua solo le classi che sono parte del vocabolario del dominio del problema. Evita di inserire classi di progetto (ossia classi derivanti dalla tecnologia, relative a problemi di comunicazione o di accesso a database): tutte le scelte implementative riguardano i workflow Progetto e Implementazione.
- ✓ Descrivi il modello di analisi sempre e solo nel *linguaggio del dominio*.

105

- ✓ 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 *Menu* o *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.
- ✓ Chiediti “Il modello è utile per tutte le persone coinvolte, soprattutto per gli utenti del dominio?”
- ✓ 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”.

106

# 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.

107

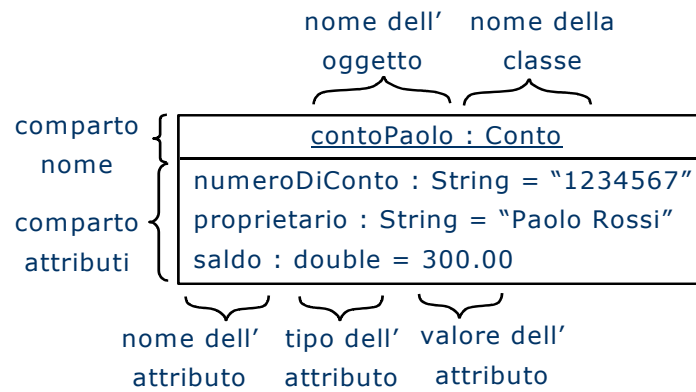
- **Comportamento**: le operazioni che l’oggetto può compiere. Per quanto riguarda la stampante, per esempio, *accendi()*, *spegni()*, *stampa(documento)* ed *interrompi()*. L’invocazione di un’operazione su un oggetto può provocare un cambio di stato. Chiaramente lo stato dell’oggetto può influenzare 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

108



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.

109

- 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.

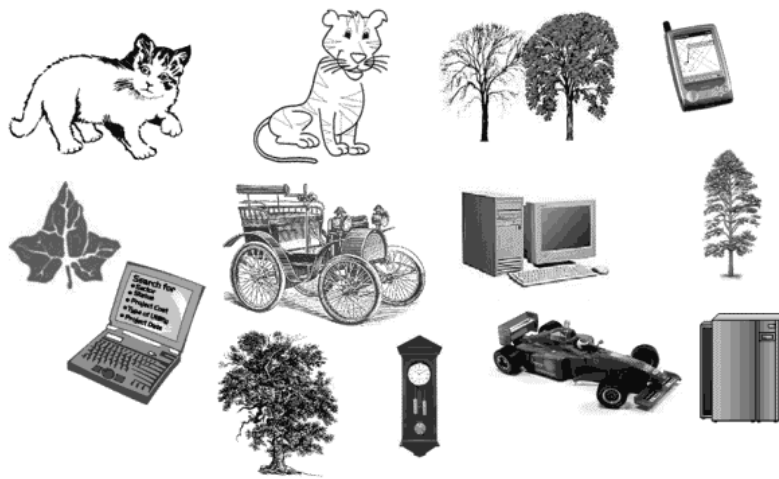
## Classi

*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?

110





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.

111

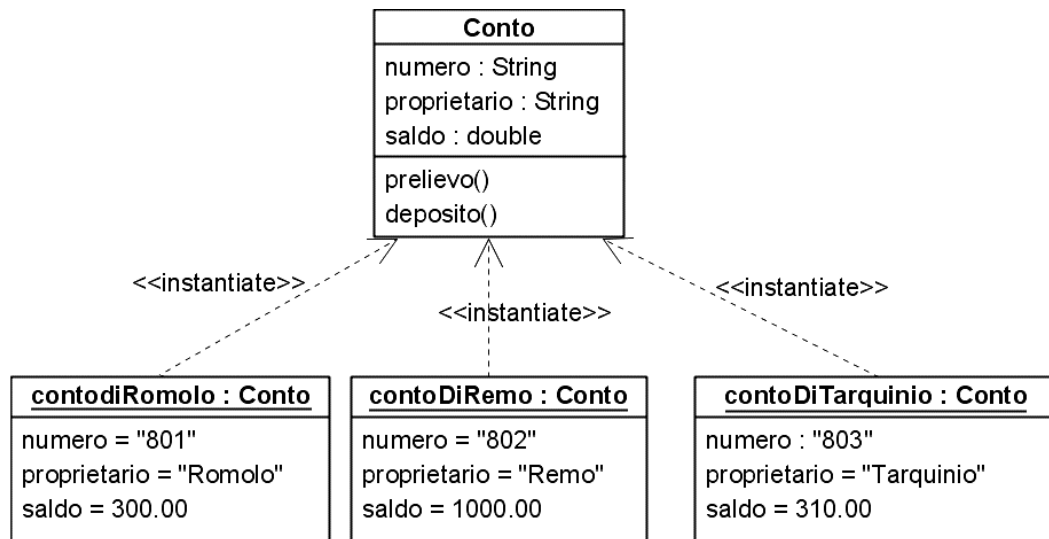
Un'analisi più dettagliata rivela che vi sono altri tipi di relazioni oltre alla istanziazione classe/oggetto. La classe *Gatto* è la generalizzazione delle classi *GattoSelvatico* e *GattoDomestico*. La classe *Foglia* ha una relazione di composizione con la classe *Albero*, poiché ogni foglia non può 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:

112

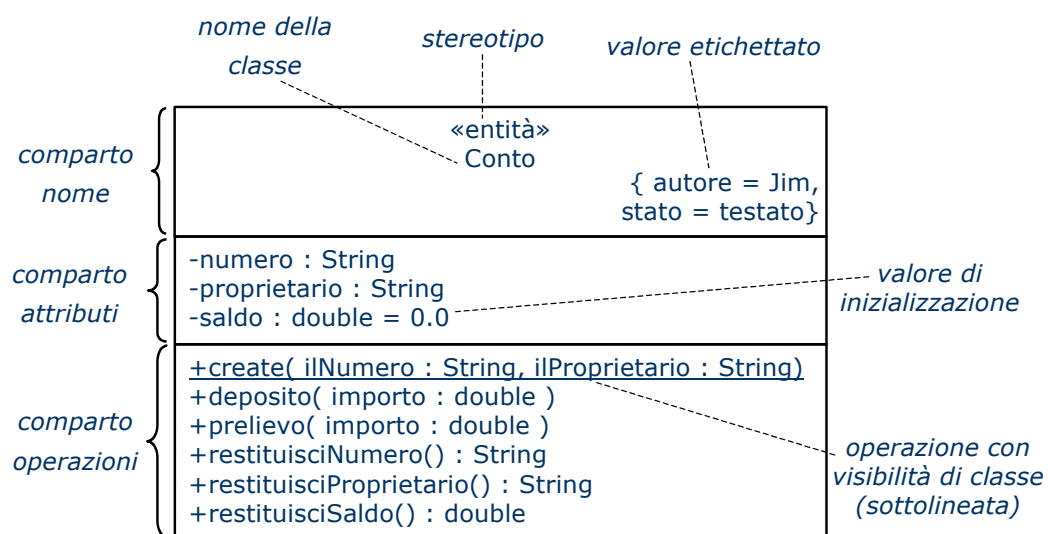


La freccia tratteggiata indica una relazione di dipendenza a cui è stato dato un significato speciale attraverso lo stereotipo «**instantiate**». Lo *UML Reference Manual* definisce una dipendenza come “una relazione tra due elementi in cui una modifica apportata ad un elemento (il fornitore) può influenzare 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.

113

## 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.

114

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 *DpstCntCrt*. 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.

115

- ✓ 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

116

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.

117

- ✓ 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**.

118

Alcuni esempi di operazioni vengono dati di seguito:

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

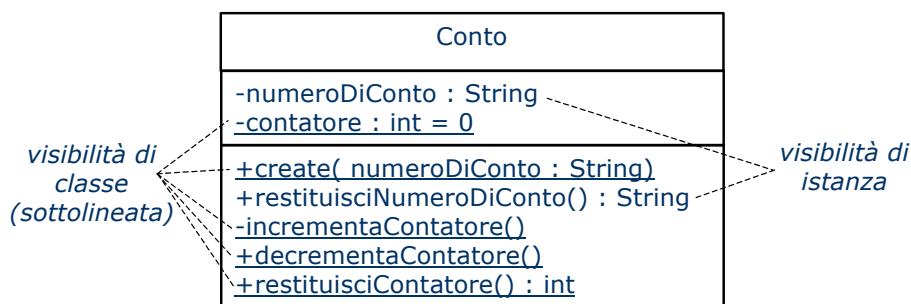
I valori di default vengono usati quando l'operazione è invocata senza specificare i corrispondenti parametri attuali.

119

## 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 sottomessi.

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.

120

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.

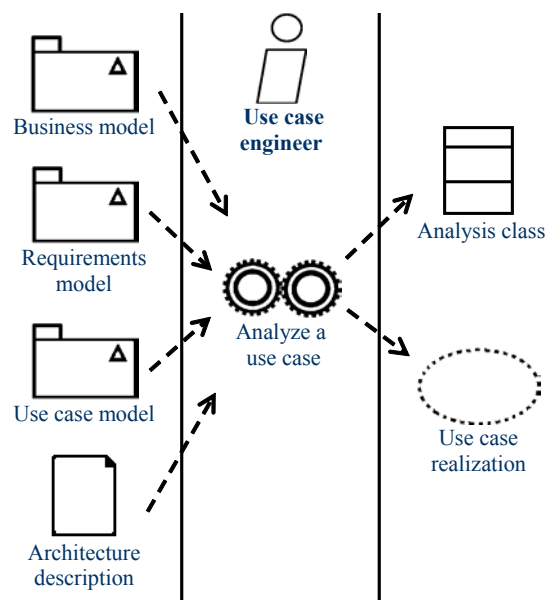
I costruttori sono operazioni speciali che creano nuove istanze di classi. Linguaggi diversi usano notazioni diverse per nominare i costruttori. Un approccio generico è quello di chiamare i costruttori `create()` ed elencare i parametri necessari. Durante l'analisi, tipicamente, non serve specificare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



121

## Trovare le classi di analisi

L'attività "Analizza un caso d'uso" ha in ingresso, oltre ai modelli e requisiti definiti nel workflow Requisiti, il modello, se esiste, del dominio e la descrizione dell'architettura, e produce classi di analisi e realizzazioni di casi d'uso. La descrizione dell'architettura è un'istantanea degli aspetti significativi dell'architettura stessa: può includere estratti di modelli UML inseriti in un testo descrittivo.



122

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.

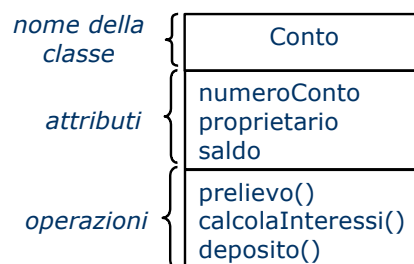
Tipicamente, rispetto alla generica notazione di classe introdotta nei lucidi precedenti, la classe di analisi avrà:

- ✓ Il nome
- ✓ I nomi degli attributi (in genere vengono elencati quelli ritenuti più importanti)

123

- ✓ 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*).

124



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

125

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 sovraccariche 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 funzioide è una normale procedura mascherata da classe.

126



- 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.

127

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”;

128

c. Verbi – per esempio, “prenota”;

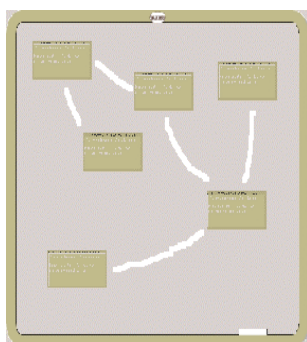
d. Frasi con verbi – per esempio, “verifica la carta di credito”;

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.

129

### *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	
<b>Responsabilità:</b> Mantenere il saldo	<b>Collaboratori:</b> 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.

130

## **Procedura**

### **1. FASE 1 – Libero scambio di vedute (brainstorm).**

Partecipanti: analisti OO, utenti ed esperti del dominio ed un moderatore.

Scopo: acquisire informazioni.

Procedura:

- a. Spiega che è un brainstorm
  - i. tutte le idee sono accettate come buone, registrate e non dibattute;
- b. Chiedi ai membri del gruppo di dare un nome alle cose del loro dominio del problema
  - i. Scrivi ogni cosa su un foglietto adesivo;
  - ii. Appendi i foglietti ad una lavagna
- c. Chiedi ai membri del gruppo di stabilire le responsabilità che le cose potrebbero avere ed inseriscile nel comparto Responsabilità.

131

- d. Con il gruppo, tenta di individuare le classi che potrebbero collaborare. Sistema i foglietti in modo che riflettano questa organizzazione e traccia delle linee tra loro. Alternativamente, memorizza le classi che collaborano con la classe in un comparto apposito del foglietto.

### **2. FASE 2 – Analisi dell'informazione.**

Partecipanti: analisti OO ed esperti del dominio.

Scopo: analizzare l'informazione raccolta nella prima fase.

Procedura:

- a. Analizza che le classi candidate rispettino le caratteristiche elencate precedentemente che descrivono classi di analisi di buona qualità.
- b. Se una classe candidata sembra essere una parte di un'altra, probabilmente è un attributo e non una classe.
- c. Se una classe candidata sembra essere poco importante o non mostrare un comportamento interessante, trasformala in attributo di un'altra classe.
- d. Nel dubbio, definisci una classe piuttosto che un attributo.

132

## *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.

133

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.

134

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 riutilizzati 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).

137

## 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**.

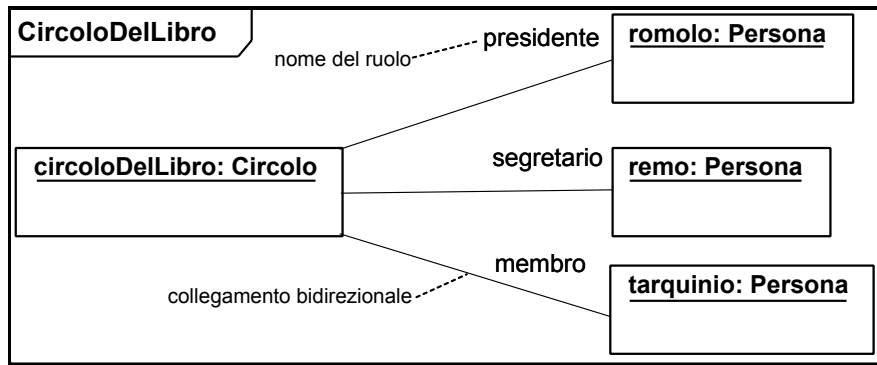
Le relazioni tra classi sono conosciute come **associazioni**. I collegamenti tra oggetti sono istanze di associazioni tra classi.

### *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.

138



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.

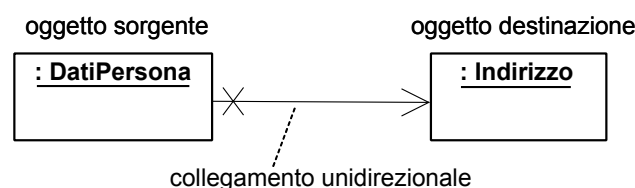
ATTENZIONE: i collegamenti sono connessioni **dinamiche** tra oggetti e quindi non sono fissi nel tempo. Nell'esempio, il ruolo di presidente potrebbe passare da romolo a remo.

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

139

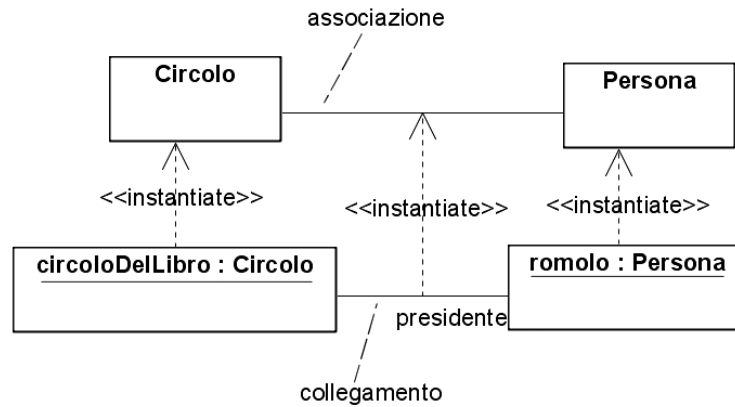
(navigabile) ed una croce all'altro (non navigabile). I messaggi possono solo fluire verso la freccia. La croce può essere omessa. Un esempio di collegamento unidirezionale navigabile è dato di seguito.



### ***Che cosa è un'associazione?***

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

140

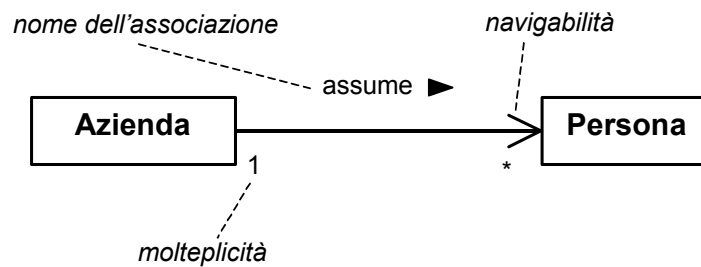


## Sintassi dell'associazione

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.

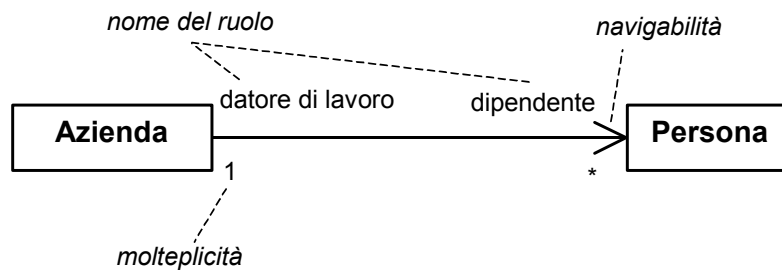
141



- **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.

142





- **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.

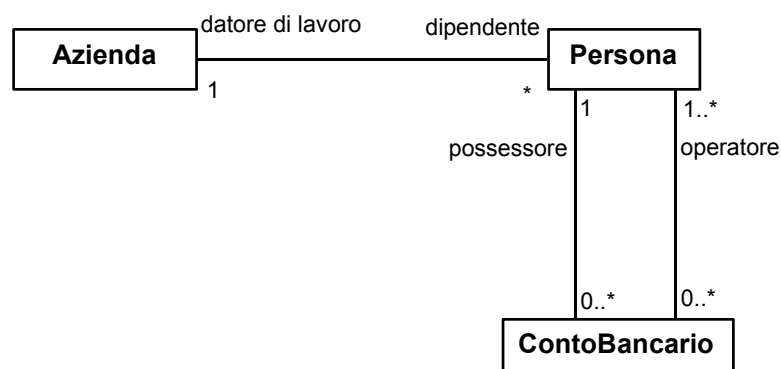
La molteplicità è specificata da una lista di intervalli, nella forma minimo..massimo, separati da virgole.

La tabella seguente mostra il significato dei simboli:

143

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

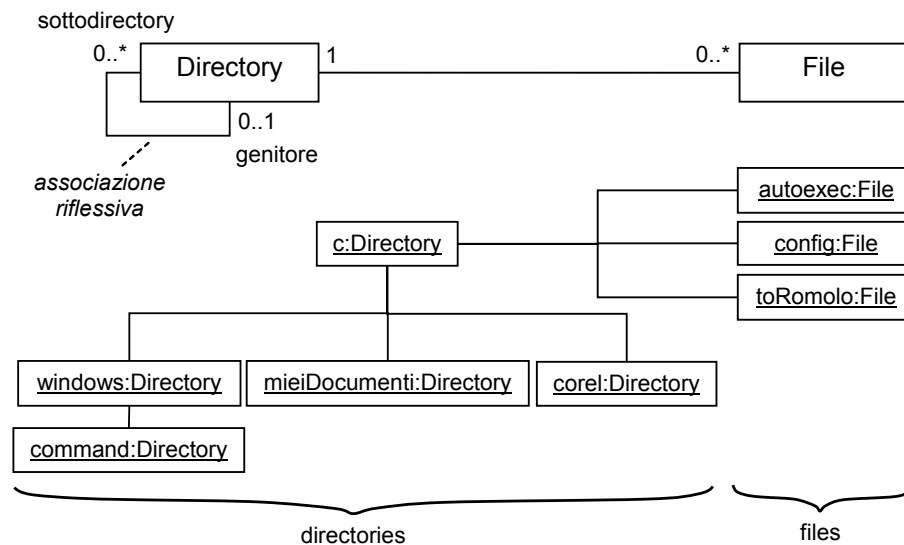
**Un esempio:**



144

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.

145

- **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.

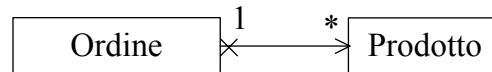
Sintassi UML 2	Idioma 1: Navigabilità UML 2.0 stretta	Idioma 2: Nessuna navigabilità	Idioma 3: standard nella pratica
	Da A a B è navigabile Da B a A è navigabile		
	Da A a B è navigabile Da B a A non è navigabile		
	Da A a B è navigabile Da B a A è indefinito		Da A a B è navigabile Da B a A non è navigabile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

146

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

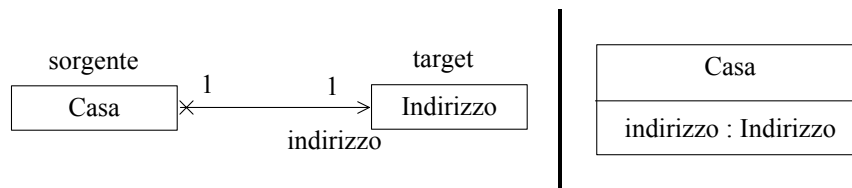


La associazione ci dice che un Ordine memorizza una lista di prodotti. Non si può navigare direttamente dal Prodotto all'Ordine. Comunque, si potrebbe ancora trovare l'oggetto Ordine associato al particolare Prodotto cercando tutti gli oggetti Ordine!!!!

147

## ***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.



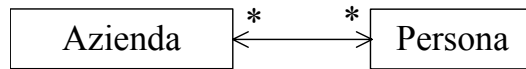
Se la molteplicità è multipla, le associazioni sono implementate o come array di attributi oppure come collezioni. Collezioni sono classi che hanno la capacità di memorizzare e recuperare riferimenti ad altri oggetti.

**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.

148

## 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:

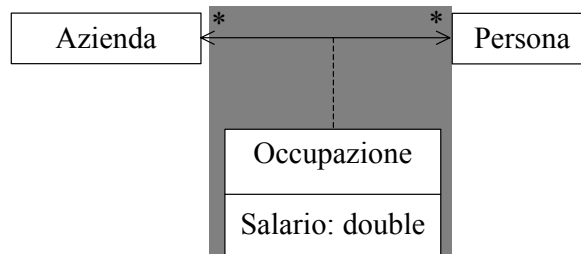


Cosa accade se si deve inserire la regola che ogni Persona percepisce un salario da ogni Azienda da cui è assunto? Dove dovremmo inserire l'attributo salario?

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

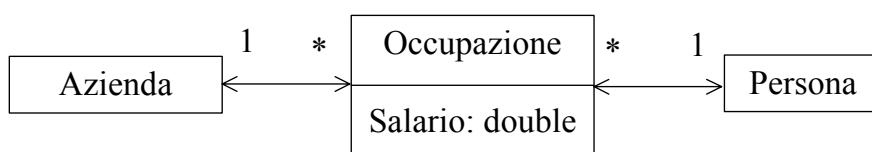
**CONCLUSIONE:** il salario è una proprietà dell'associazione. UML permette di modellare questo scenario con le classi di associazione. La **classe di associazione** è rappresentata da tutto ciò che è racchiuso nel rettangolo grigio (che fa parte del modello) nella figura seguente.

149



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

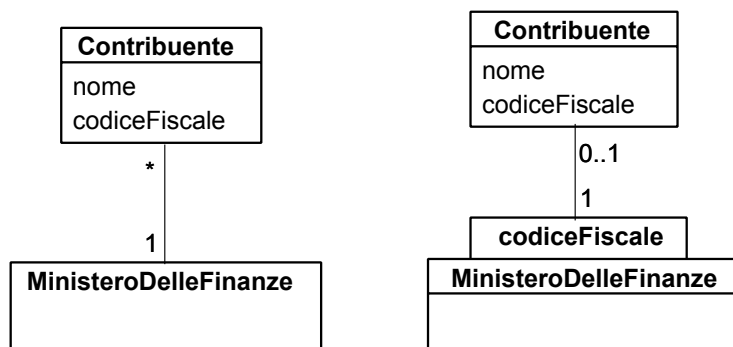


150

## 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.

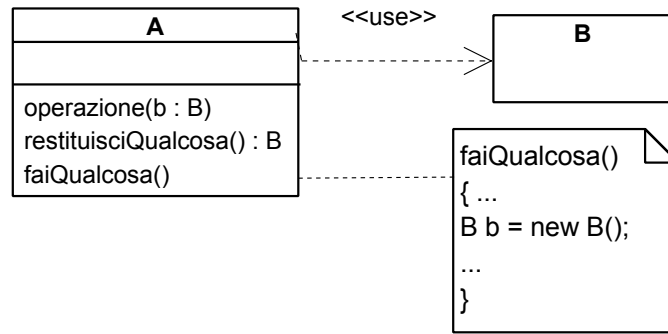


## Dipendenza

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 omissso, 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.

153

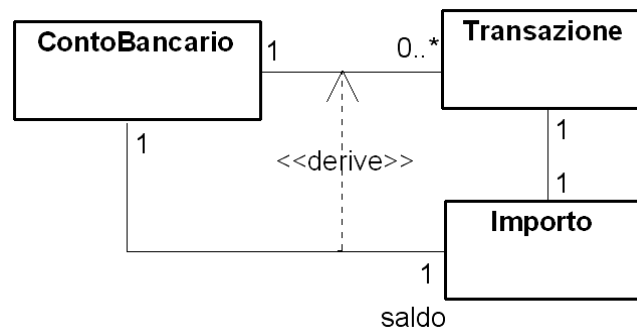
- **«parameter»** - il fornitore è un parametro di un'operazione del cliente;
- **«send»** - il cliente è un'operazione che invia il fornitore (che deve essere un segnale) a qualche specificato target.
- **«instantiate»** - il cliente è un'istanza del fornitore.

- **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.

154

- «**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.

155

- «**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++.

156

# 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.

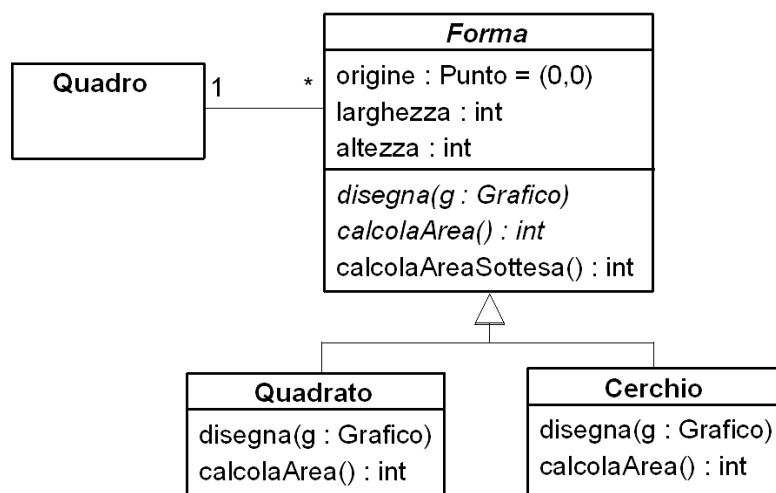
L'elemento più specifico può essere usato ovunque sia utilizzabile l'elemento più generale senza provocare problemi al sistema.

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.

157

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*.

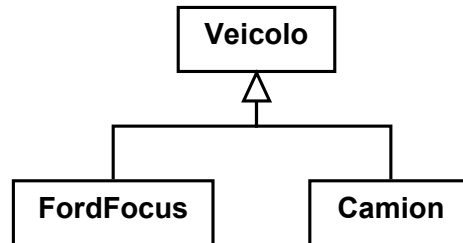
158



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.

159

## Polimorfismo

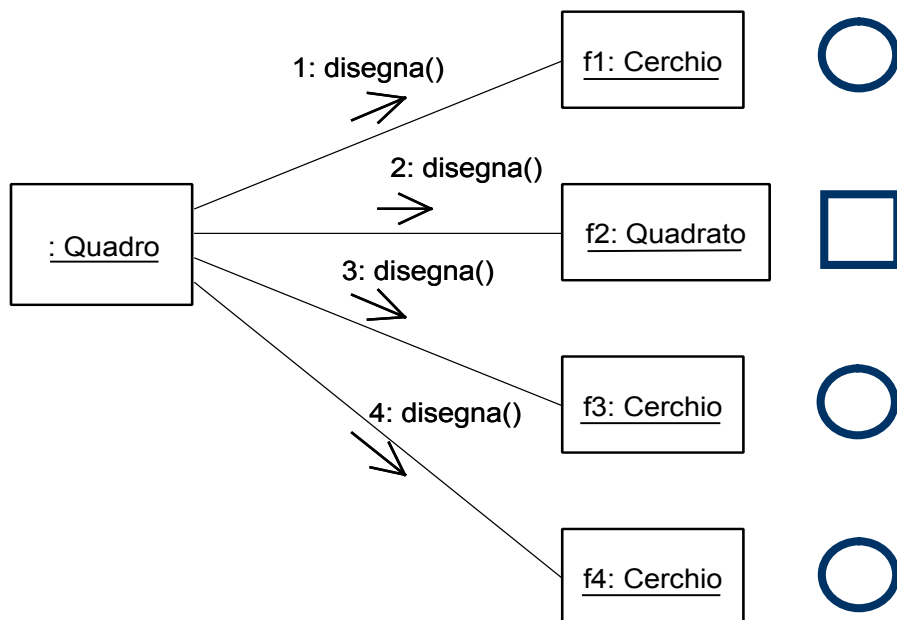
Polimorfismo significa “molte forme”. Un’operazione polimorfica è un’operazione che ha molte implementazioni. Le operazioni `disegna()` e `calcolaArea()` viste precedentemente sono esempi di operazioni polimorfiche.

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:

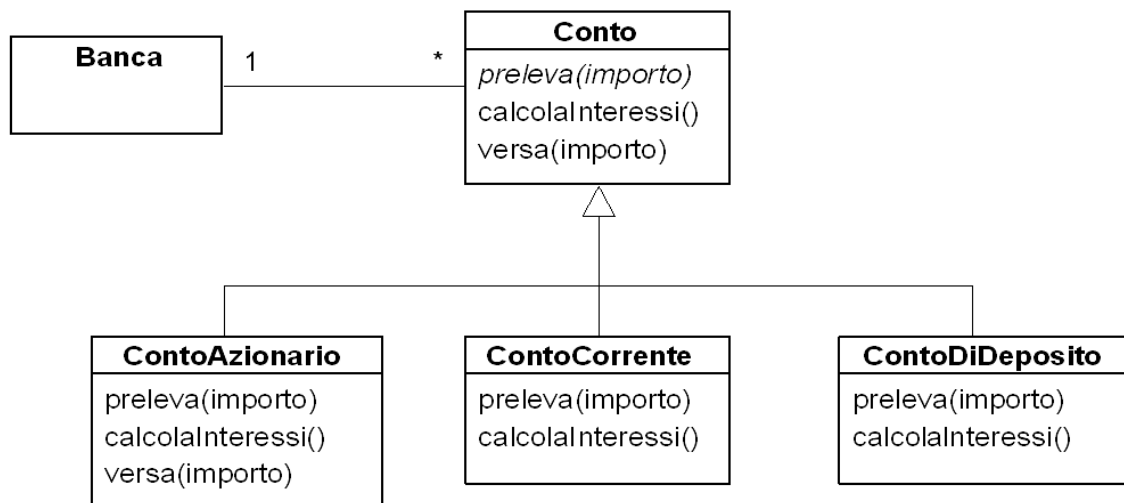
160



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.

161



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.

162

## Package di analisi

Un package è un'entità UML agglomerante, che contiene e gestisce elementi del modello. Il package è sostanzialmente un meccanismo per organizzare in gruppi gli elementi ed i diagrammi del modello. Il package può essere usato per:

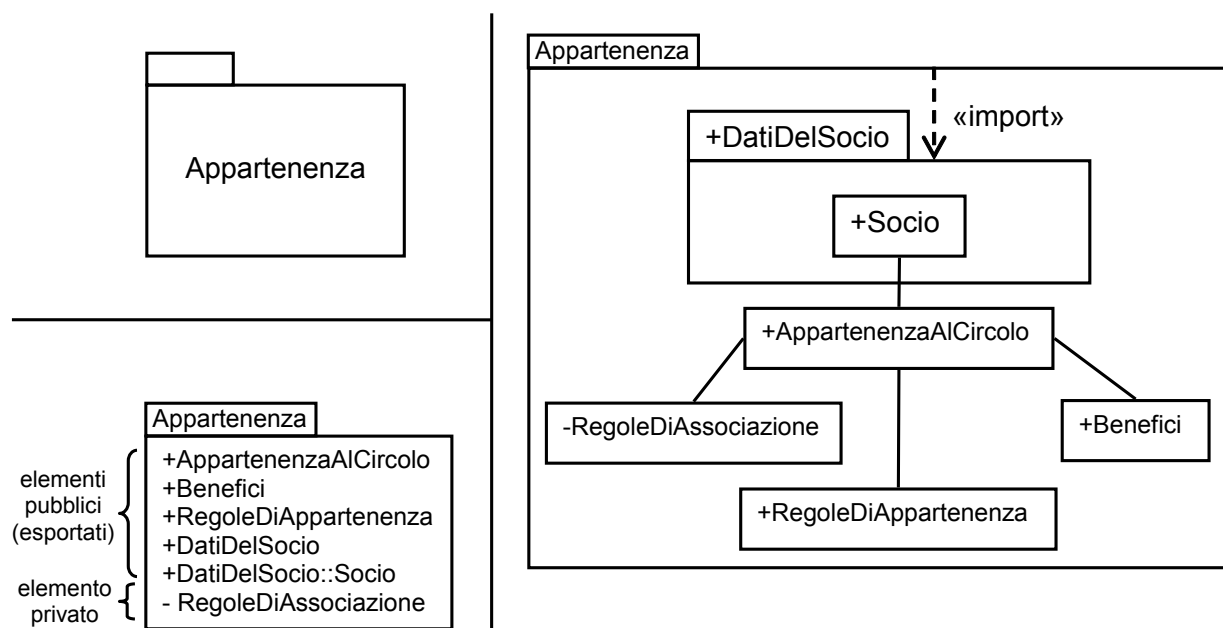
- **fornire uno spazio dei nomi (namespace) incapsulato** all'interno del quale tutti i nomi devono essere unici;
- **raggruppare elementi semanticamente correlati**;
- **definire confini semantici** nel modello;
- **definire unità per lavorare parallelamente e gestire la configurazione**.

NOTA BENE: il package è un meccanismo di raggruppamento logico che fornisce un namespace ai suoi componenti. Per raggruppare fisicamente elementi del modello si deve usare un componente, che vedremo in seguito.

NOTA BENE: ogni elemento è contenuto in un solo package e la gerarchia di contenimento forma un albero la cui radice è un package stereotipato «**topLevel**». Per default, se un elemento non viene inserito in un package, allora appartiene al package «**topLevel**».

163

La sintassi del package è rappresentata nella figura seguente:



I package di analisi dovrebbero contenere:

- casi d'uso;

164

- classi di analisi;
- realizzazioni di classi di analisi.

Gli elementi di un package possono avere visibilità pubblica (indicata con +) o privata (indicata con -). Un elemento pubblico agisce come da interfaccia al resto del package. Dato che l'interfaccia di un package dovrebbe essere il più possibile piccola e semplice, il numero di elementi pubblici del package dovrebbe essere minimizzato e il numero di elementi privati massimizzato. Questo obiettivo è più facilmente raggiungibile nel progetto, quando tutte le associazioni diventano unidirezionali, che nell'analisi.

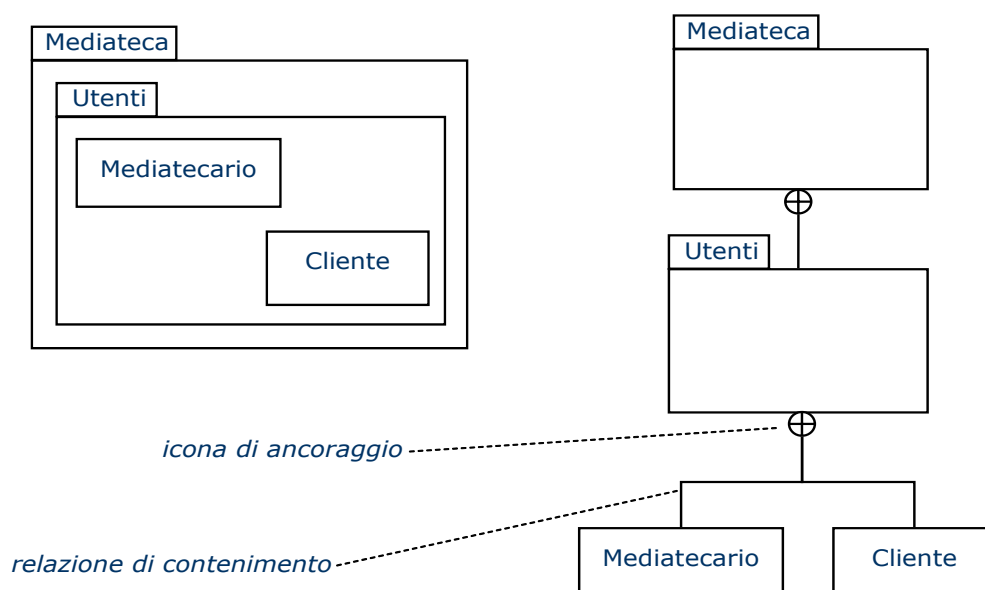
I nomi dei package vengono dati in UpperCamelCase.

### ***Package annidati***

I package possono essere annidati uno dentro l'altro. Il numero dei livelli di annidamento può essere qualsiasi, ma tipicamente non supera i tre. Nel modello seguente, il package Mediateca contiene il package Utenti che contiene le classi Cliente e Mediatecario. Per accedere alla classe Cliente dall'esterno del package bisogna fornire l'intero percorso (pathname o nome qualificato) *Mediateca::Utenti::Cliente*. Questo è dovuto alla presenza dei namespace che creano un confine all'interno del quale tutti i nomi devono essere unici.

165

ATTENZIONE: i package contenuti hanno accesso ai namespace dei package che li contengono. Il package Utenti può accedere agli elementi del package Mediateca usando nomi non qualificati. Al contrario, il package contenente deve usare il nome qualificato, cioè il pathname a partire dal proprio package escluso, per accedere agli elementi del package contenuto.

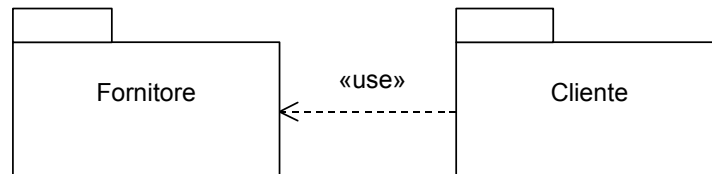


166

## Dipendenze tra i package

I package possono essere legati da dipendenze quali:

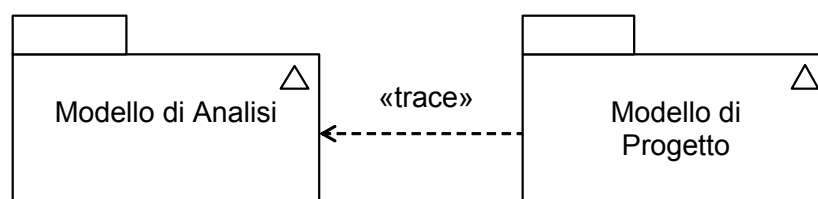
- **«use»** - un elemento del package cliente usa un elemento pubblico del package fornitore;



- **«import»** - gli elementi pubblici del namespace del fornitore sono aggiunti agli elementi pubblici del namespace del cliente, per cui gli elementi del cliente possono accedere a tutti gli elementi pubblici del fornitore;
- **«access»** - gli elementi pubblici del namespace del fornitore sono aggiunti come elementi privati al namespace del cliente, per cui gli elementi del cliente possono accedere a tutti gli elementi pubblici del fornitore. A differenza della dipendenza **«import»**, **«access»** esegue un'unione dei namespace privata, cioè gli elementi aggiunti al cliente sono privati nel cliente.

167

- **«trace»** - rappresenta l'evoluzione di un package durante lo sviluppo: da un package appena tratteggiato durante l'analisi ad uno sempre più specifico durante il progetto. Spesso rappresenta relazioni tra modelli differenti.



- **«merge»** - elementi pubblici del package fornitore sono fusi con gli elementi del package cliente. Crea elementi cliente nuovi ed espansi. Questa relazione non dovrebbe essere usata nell'analisi e nel progetto object-oriented standard.

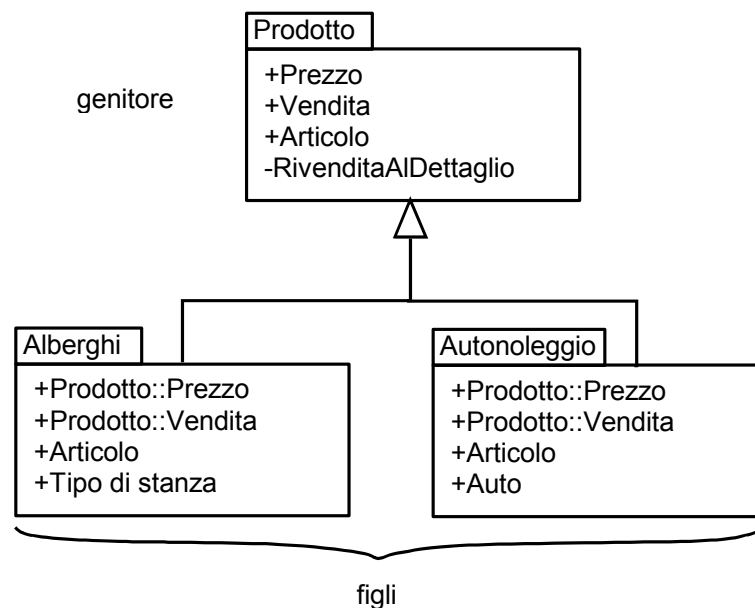
## Transitività

Se c'è una relazione tra un'entità A ed un'entità B ed una relazione tra B e C, allora c'è una relazione implicita anche tra A e C. La relazione di dipendenza **«import»** è transitiva (se A importa B, e B importa C, allora A implicitamente importa C); mentre la dipendenza **«access»** non lo è (se A accede a B, e B accede a C, allora A non può implicitamente accedere a C). La mancanza di transitività in questa relazione permette di gestire attivamente e controllare la coesione e l'accoppiamento nel modello: niente viene accaduto se non lo è esplicitamente.

168

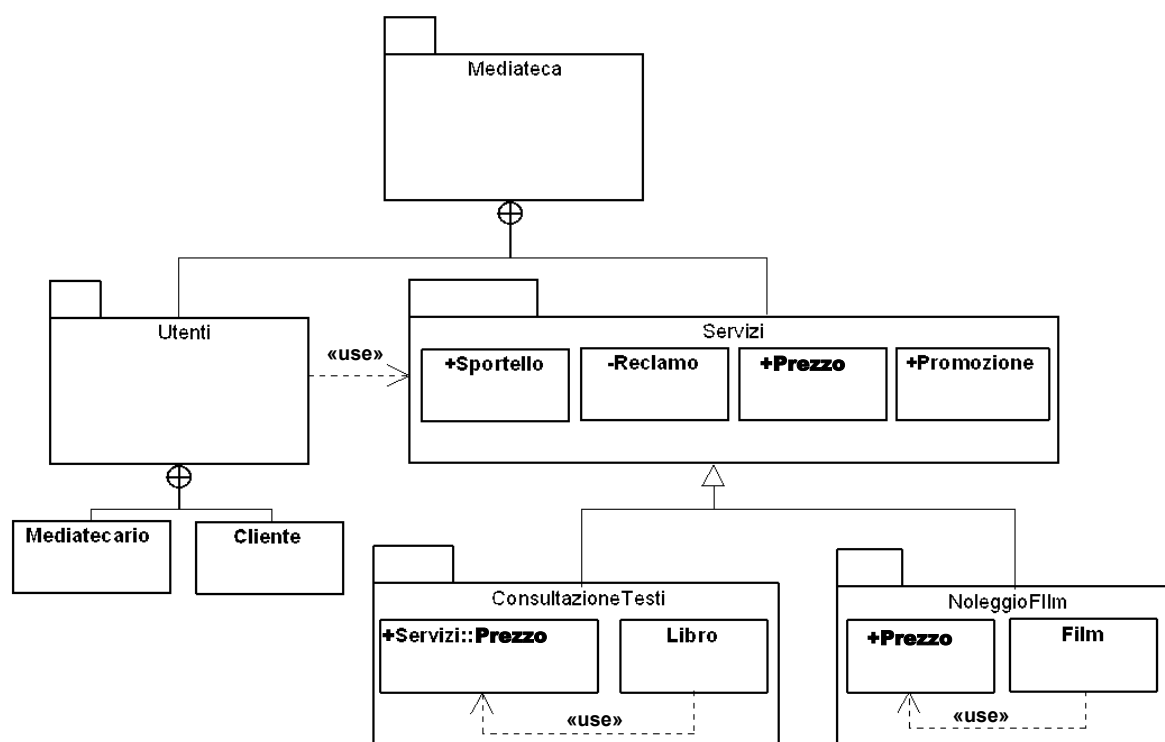
## Generalizzazione dei package

La generalizzazione tra package è simile a quella tra classi. I package figli ereditano gli elementi pubblici del package genitore, che possono essere sovrascritti. Vale il principio di sostituibilità (il package base può essere rimpiazzato dal package figlio).



169

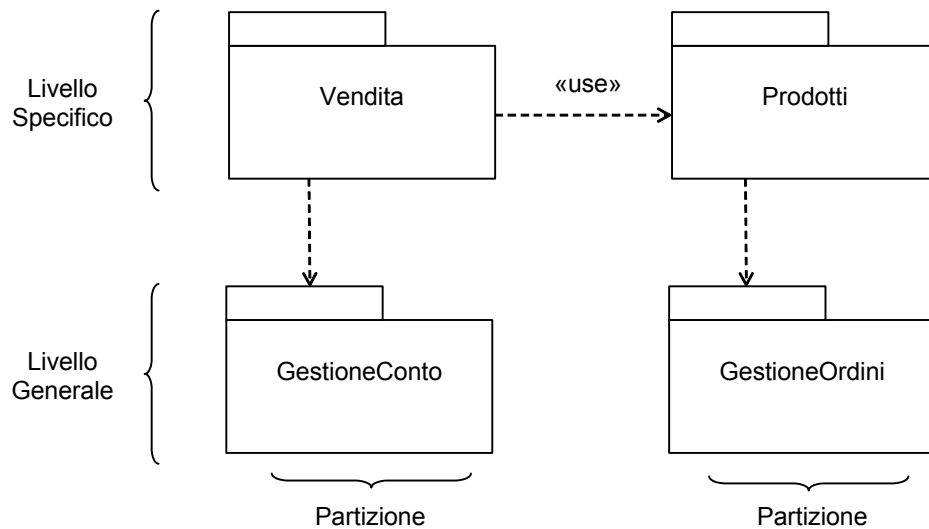
Nella figura seguente, il package *NoleggioFilm* ha sovrascritto la classe *Prezzo*, mentre il package *ConsultazioneTesti* ha adoperato il prezzo per i servizi generici che per comodità è stato riferito nel package esteso, con nome qualificato.



170

## Analisi architetturale

Nell'analisi architetturale, tutte le classi di analisi sono organizzate in un insieme di package di analisi coesivi e questi sono ulteriormente organizzati in partizioni e livelli. Ogni package di analisi all'interno di un livello è una partizione.



171

L'analisi architetturale dovrebbe minimizzare l'accoppiamento tra package, andando a minimizzare le dipendenze tra package ed il numero di elementi pubblici in ogni package, e massimizzando il numero di elementi privati.

In Analisi, i package generalmente vengono organizzati in due livelli: livello specifico e livello generale. Il livello specifico contiene funzionalità che sono specifiche della applicazione particolare. Il livello generale contiene funzionalità che sono più generalmente utili.

### Come si trovano i package di analisi?

- Identificando raggruppamenti reali di elementi del modello che hanno forti connessioni semantiche. Spesso, i package vengono scoperti man mano che il modello matura.
- Da dove si comincia ad individuare i package candidati? Il modello statico è la sorgente più utile. Cerca:
  - Gruppi coesivi di classi nel diagramma delle classi;
  - Gerarchie di ereditarietà;

172

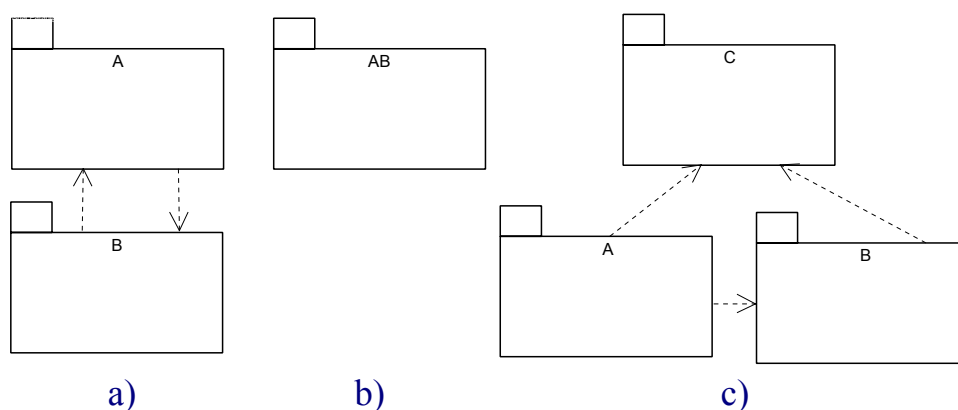
- Modello dei casi d'uso: uno o più casi d'uso che supportano un attore o un processo nel dominio o un insieme di casi d'uso correlati. È importante ottenere package che siano coesivi da un punto di vista dei processi di business o del dominio.

- Elabora i package candidati tentando di minimizzare i membri pubblici del package e le dipendenze tra package spostando classi tra package, aggiungendo package o cancellando package. Ricorda che i package dovrebbero contenere un gruppo di classi fortemente correlate. L'ereditarietà è la forma di correlazione più forte, poi seguono composizione, aggregazione e dipendenza.
- Mantieni il modello semplice evitando troppi annidamenti tra package (tipico della decomposizione funzionale), riducendo l'uso della generalizzazione e degli stereotipi di dipendenza.

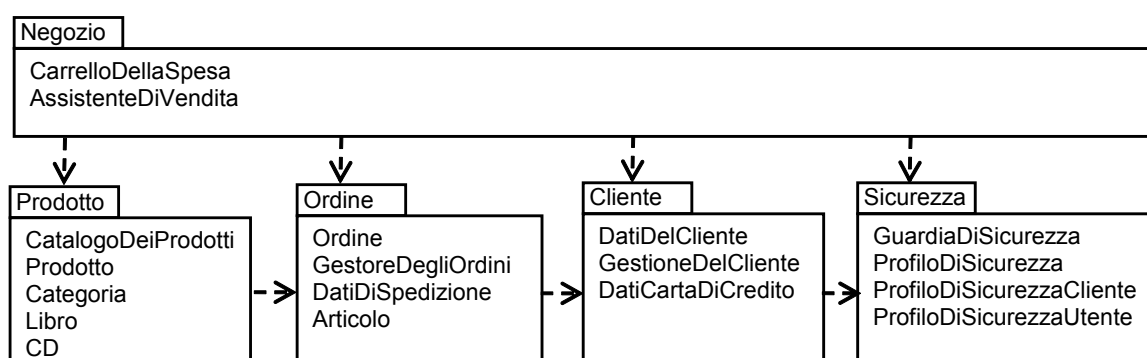
Come regola generale, cerca di inserire da 4 a 10 classi di analisi per package. A volte si rende necessario introdurre package con una o due classi per evitare dipendenze cicliche.

Consideriamo il caso dei due package A e B nella figura seguente. Visto che c'è una forte correlazione tra i due package, una soluzione potrebbe essere quella di unirli in un unico package. Un'altra soluzione, potrebbe essere quella di raggruppare gli elementi comuni ai due package generando un nuovo package (il package C in figura).

173



Un esempio di modello di package per un semplice sistema di e-commerce è mostrato nella figura seguente



174



## 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.

175

**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.

176

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.

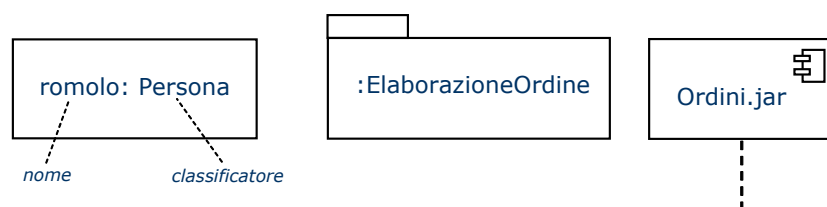
Nella realizzazione dei casi d'uso, il classificatore di contesto è il caso d'uso e le interazioni servono a dimostrare come il comportamento descritto dal caso d'uso possa essere realizzato attraverso le istanze dei classificatori (in questo caso, le classi di analisi) che si inviano messaggi l'un l'altro.

177

## 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.

178


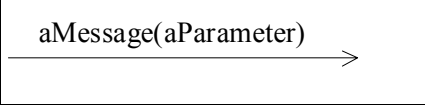
## Messaggi

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

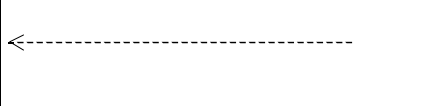
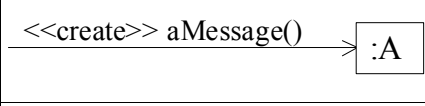
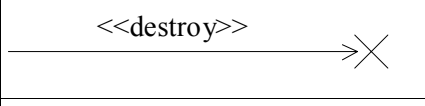
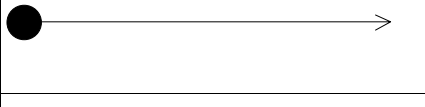
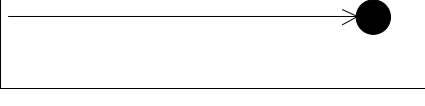
- Un'invocazione di un'operazione;
- La creazione o distruzione di un'istanza;
- L'invio di un segnale.

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

179

	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.

180

## Diagrammi di interazione

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

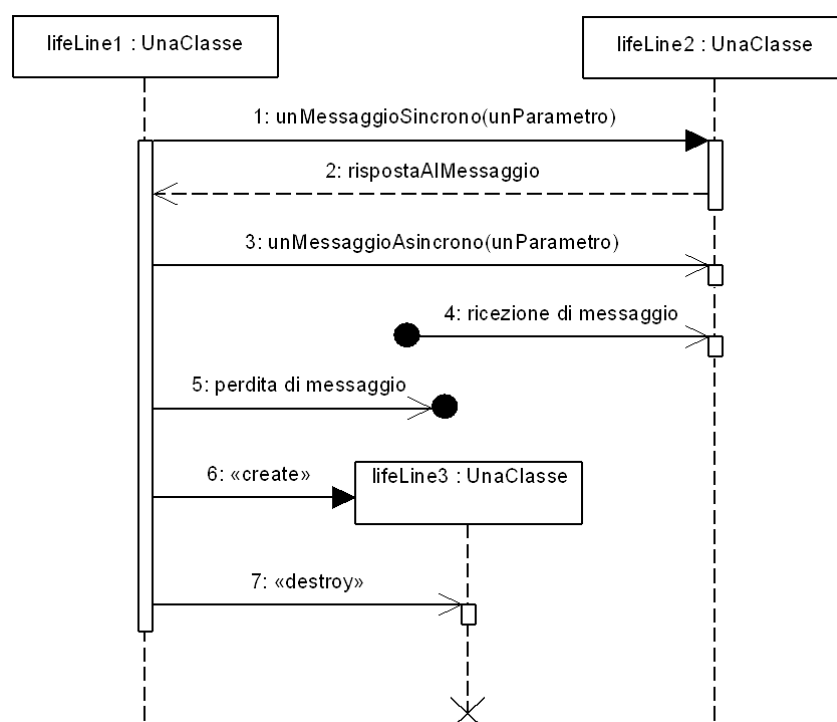
- **diagrammi di sequenza** - mostrano la sequenza, ordinata temporalmente, di messaggi scambiati tra linee di vita. In un diagramma di sequenza il tempo scorre dall'alto verso il basso, e le varie linee di vita sono affiancate orizzontalmente.
- **Diagrammi di comunicazione** – mostrano le relazioni 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

181

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.



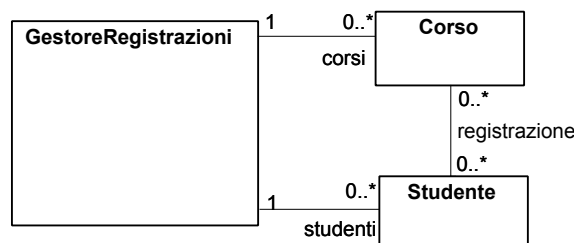
182

Consideriamo il caso d'uso seguente:

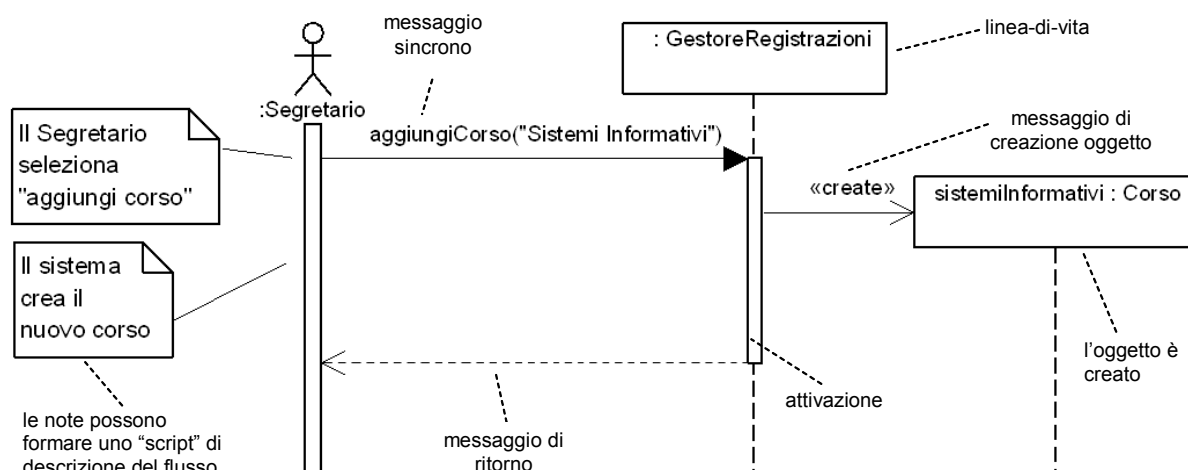
Caso d'uso: <i>AggiungiCorso</i>
ID: 8
Breve descrizione: <i>Aggiunge al sistema i dettagli di un nuovo corso</i>
Attori primari: <i>Segretario</i>
Attori secondari: <i>Nessuno</i>
Precondizioni: <i>1. Il Segretario ha effettuato il login nel sistema</i>
Flusso principale: <i>1. Il Segretario seleziona "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.

183



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



184

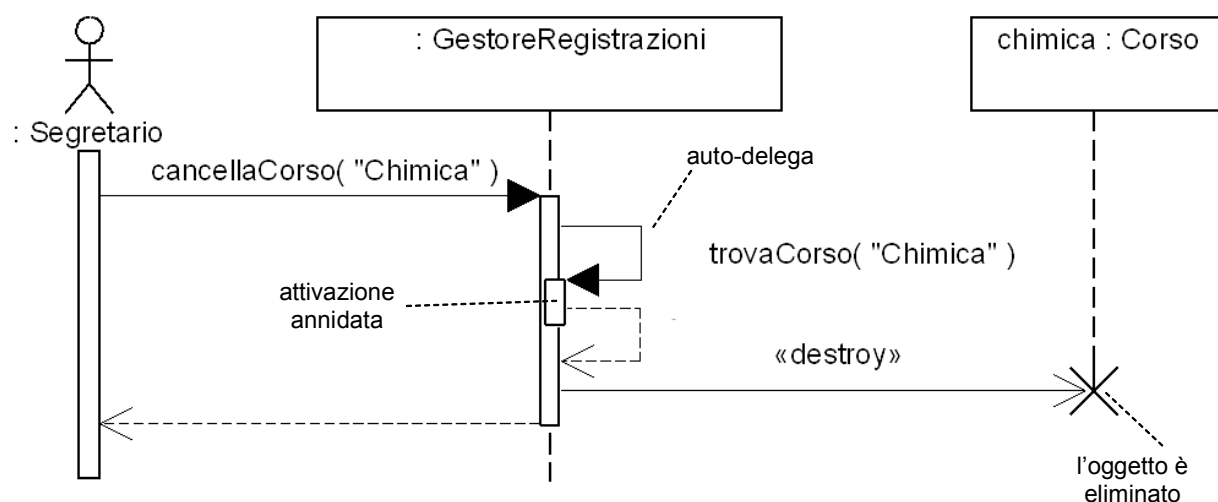
NOTA BENE: il diagramma non mostra una rappresentazione esatta di ogni passo del caso d'uso. I primi due passi prevedono una interazione con una interfaccia utente che verrà mostrata in fase di progetto.

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>Corsolnesistente</i>

185

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.



186

## *Invarianti di stato e vincoli*

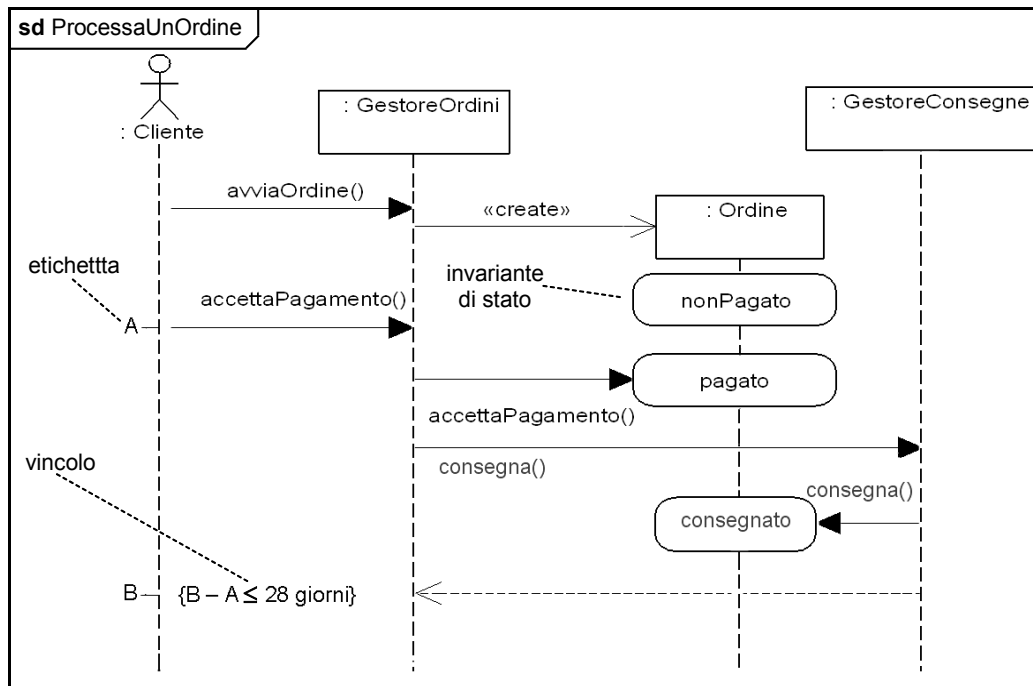
Stato: una condizione o situazione nella vita di un oggetto durante la quale l'oggetto soddisfa qualche condizione, realizza qualche attività o aspetta qualche evento.

Quando un'istanza riceve un messaggio, il suo stato può cambiare, e ciò può essere evidenziato mediante un rettangolo dai bordi smussati posto sulla linea-di-vita. Questi rettangoli rappresentano gli invarianti di stato. Gli invarianti di stato permettono di catturare gli stati chiave nell'esistenza di una linea di vita. Consideriamo il caso d'uso seguente:

187

Caso d'uso: <i>ProcessaUnOrdine</i>
ID: 5
Breve descrizione: <i>Il Cliente avvia un ordine che poi viene pagato e consegnato</i>
Attori primari: <i>Cliente</i>
Attori secondari: <i>Nessuno</i>
Precondizioni: <i>Nessuna</i>
Flusso principale: <i>1. Il caso d'uso inizia quando l'attore Cliente crea un nuovo ordine.</i> <i>2. Il Cliente paga interamente l'ordine.</i> <i>3. I beni sono consegnati al Cliente entro 28 giorni dalla data di pagamento</i>
Postcondizioni: <i>1. L'ordine è stato pagato</i> <i>2. I beni sono stati consegnati in 28 giorni dal pagamento finale</i>
Flussi alternativi: <i>PagamentoInEccesso</i> <i>OrdineCancellato</i> <i>BeniNonConsegnati</i> <i>BeniConsegnatiInRitardo</i> <i>PagamentoParziale</i>

188



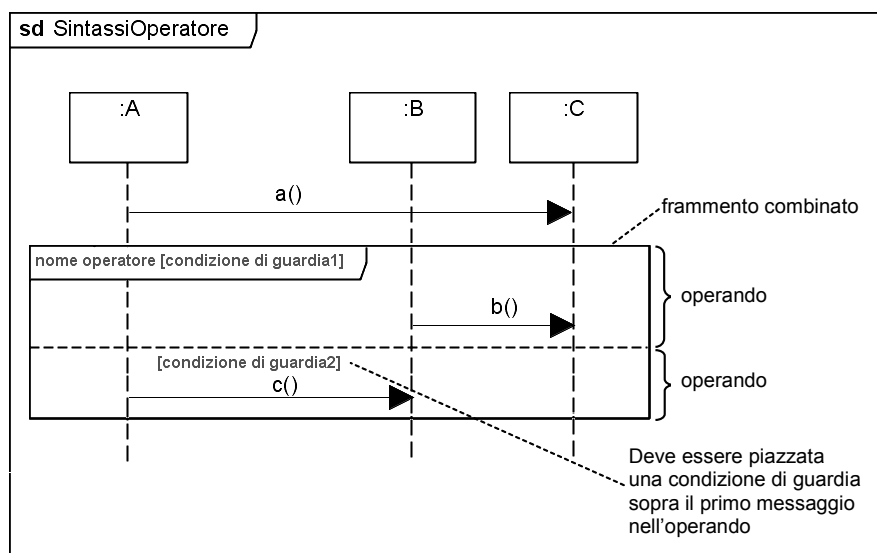
NOTA BENE: nell'esempio, l'ordine viene creato con uno stato iniziale "non pagato" e, alla ricezione del messaggio di accettazione di pagamento, il suo stato cambia in "pagato". Nel medesimo diagramma viene espresso anche il **vincolo** di consegna entro 28 giorni, mediante due etichette ("A" e "B") ed una espressione informale che deve essere vera.

189

Se nella classe Ordine è definita una macchina a stati (vedremo in seguito), gli stati nella macchina devono corrispondere agli invarianti di stato nel diagramma di sequenza.

### 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.



190



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 vita di operandi differenti avvengano nella stessa sequenza degli operandi
strict	sequenza stretta	Gli operandi vengono eseguiti in stretta sequenza

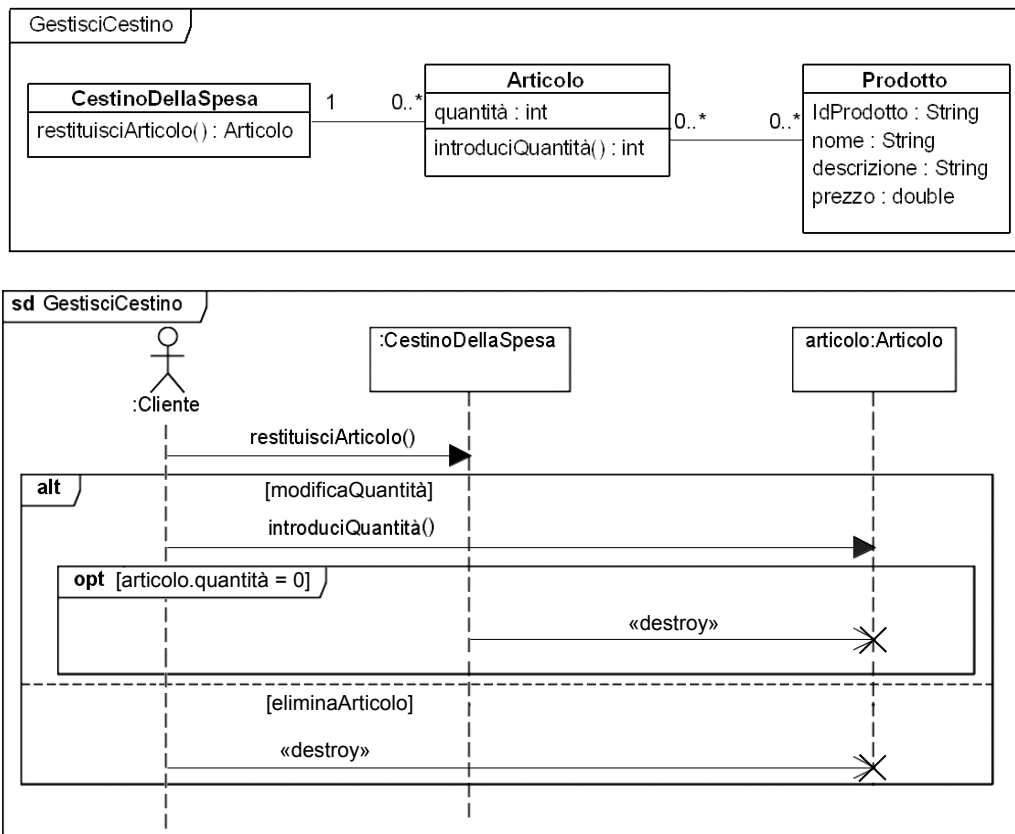
191

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. <b>If</b> il Cliente seleziona "elimina articolo"</i> <i>2.1 Il sistema rimuove l'articolo dal cestino</i> <i>3. <b>If</b> il Cliente introduce una nuova quantità</i> <i>3.1 Il sistema aggiorna la quantità di articoli nel cestino</i>
Postcondizioni: <i>Nessuna</i>
Flussi alternativi: <i>Nessuno</i>

192

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



193

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 *	=>	<code>while(true) {body}</code>
loop n,m	=>	<code>for i=n to m {body}</code>
loop [booleanExpression]	=>	<code>while(booleanExpression) {body}</code>
loop 1,* [booleanExpression]	=>	<code>repeat {body} while(booleanExpression)</code>
loop [for each object in collectionOfObjects]	=>	<code>forEach object in collection {body}</code>
loop [for each object in ClassName]	=>	<code>forEach object of class {body}</code>

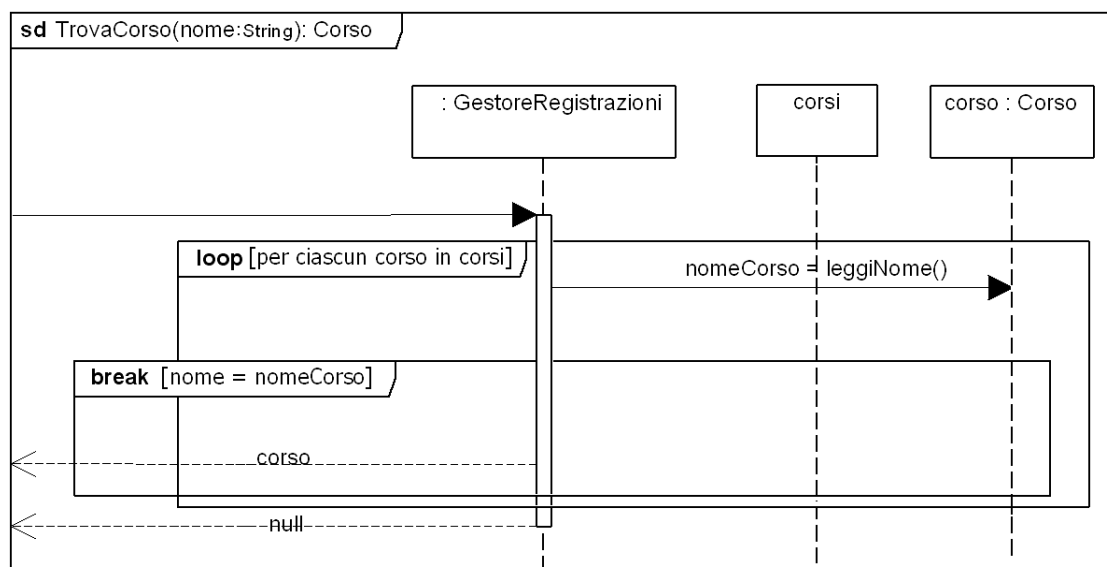
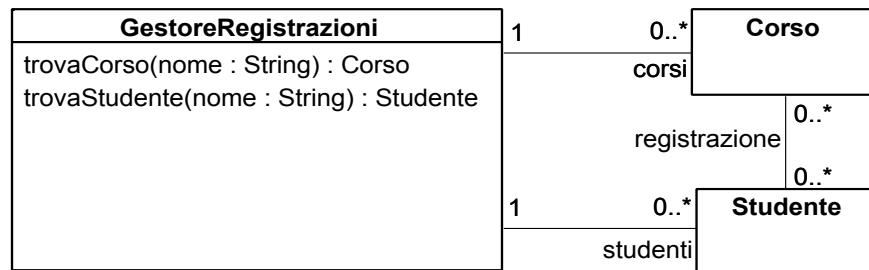
Consideriamo il caso d'uso seguente.

194

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: 1. <i>Il caso d'uso inizia quando il Segretario seleziona "trova corso".</i> 2. <i>Il Segretario inserisce il nome del corso da trovare.</i> 3. <b>IF</b> <i>il Sistema trova un corso con il nome inserito</i> 3.1 <i>Il Sistema restituisce il corso trovato</i> 4. <b>ELSE</b> 4.1 <i>Il sistema comunica che non esiste alcun corso con tale nome</i>
Postcondizioni: <i>Nessuna</i>
Flussi alternativi: <i>Nessuno</i>

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

195



196

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

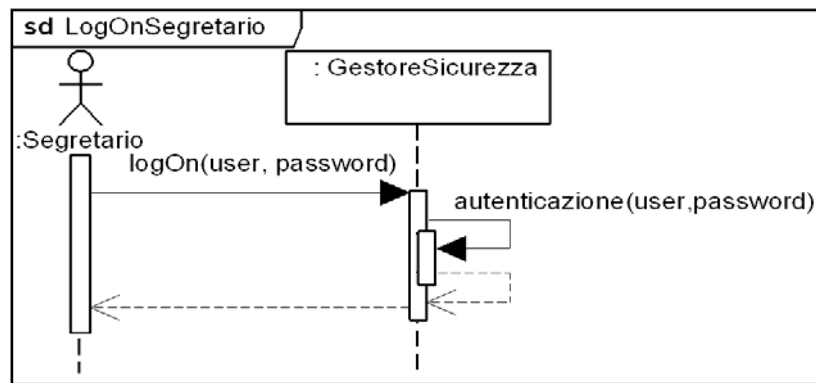
Per evitare di riusare più volte frammenti di interazioni che si ripetono in differenti diagrammi di sequenza, UML 2 mette a disposizione il riferimento ad interazione, detto **occorrenza di interazione**.

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

197

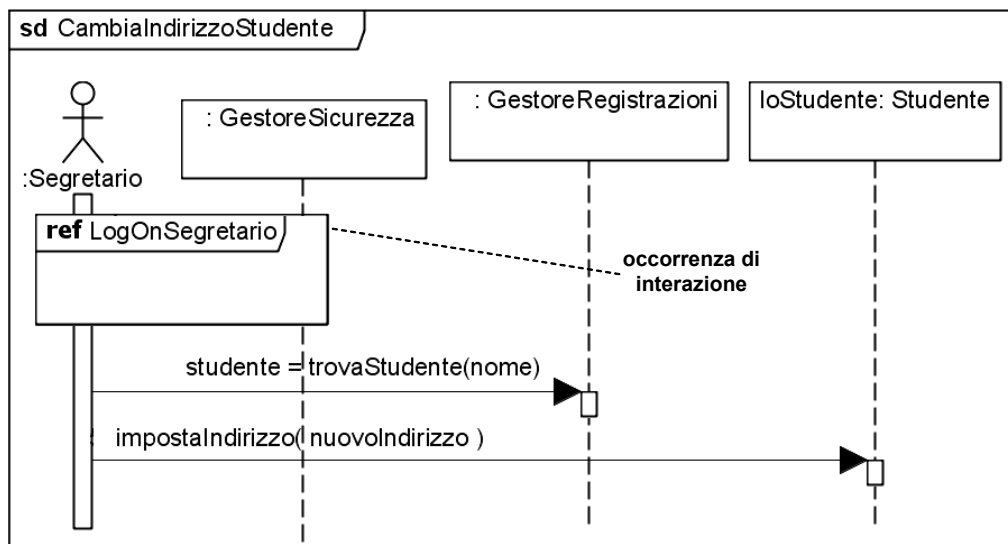
Caso d’uso: <i>LogOnSegretario</i>
ID: 4
Breve descrizione: <i>Il Segretario effettua il logon sul sistema</i>
Attori primari: <i>Segretario</i>
Attori secondari: <i>Nessuno</i>
Precondizioni: <i>Il Segretario non ha effettuato il logon sul sistema</i>
Flusso principale: <i>1. Il caso d’uso inizia quando 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>

198



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.

199



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

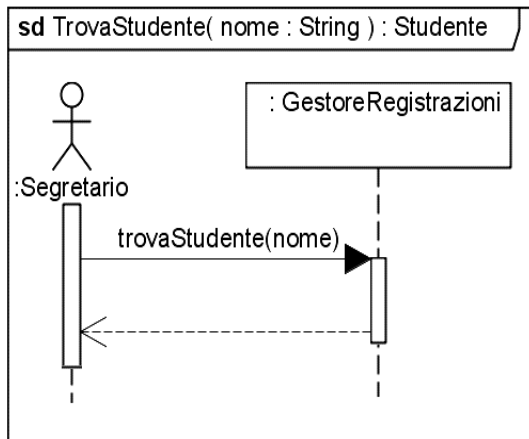
ATTENZIONE: per indicare la visibilità dell'occorrenza di interazione, tracciala attraverso le linee di vita che usa.

200

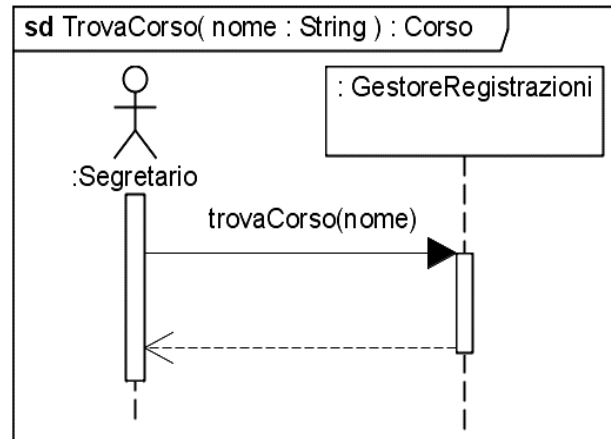
## Parametri

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)



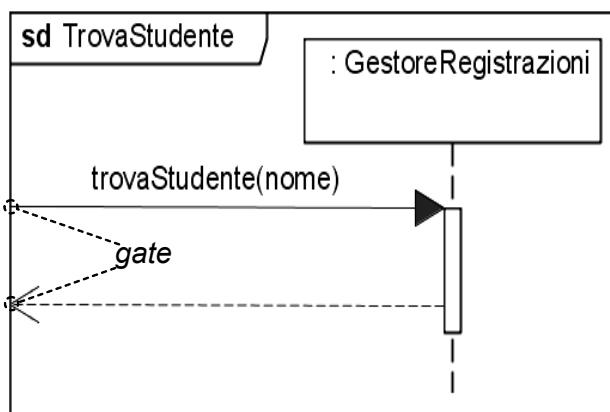
b)

201

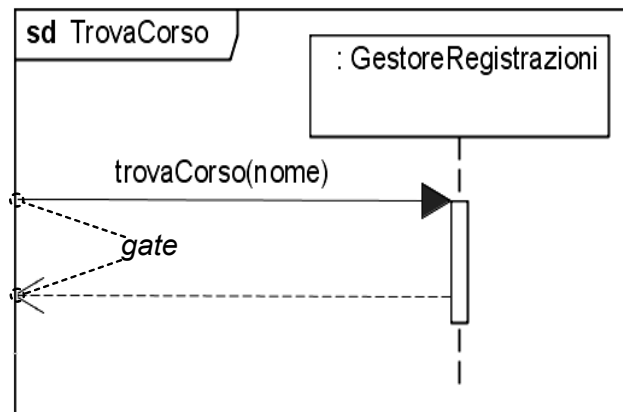
## Gate

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.



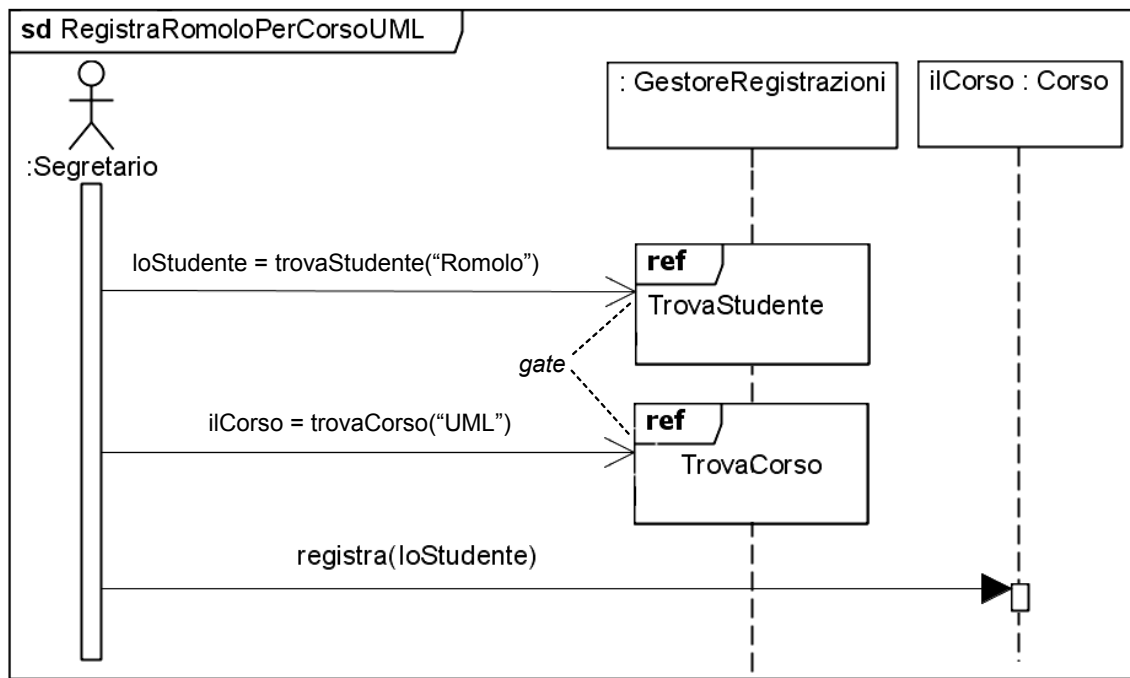
a)



b)

202

La figura seguente mostra l'utilizzo dei gate.



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

203

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.

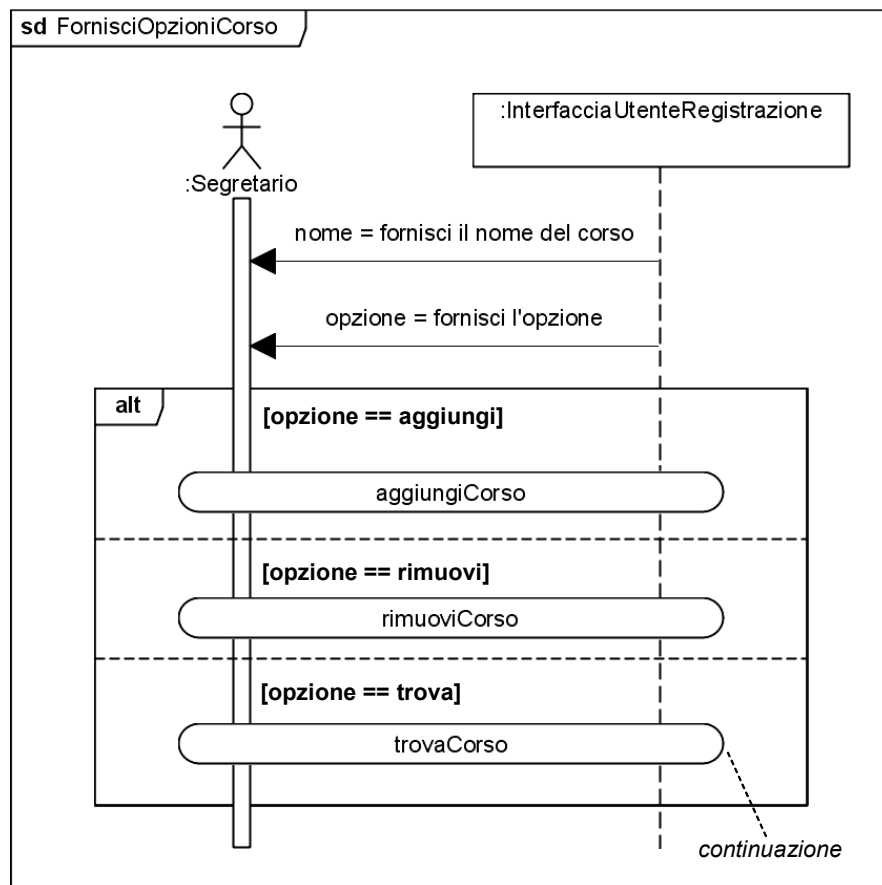
## Continuazioni

Una continuazione permette ad un frammento di interazione di indicare che il suo flusso può essere continuato da un altro frammento di interazione. La continuazione è indicata da un'etichetta racchiusa da un rettangolo arrotondato.

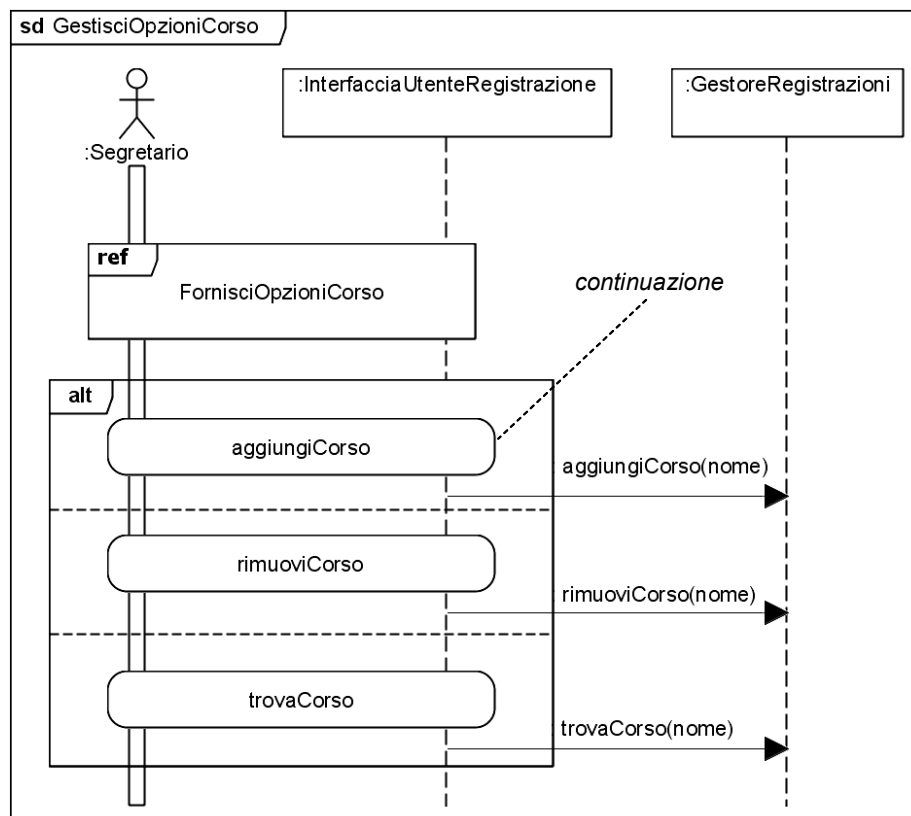
Quando è l'ultimo elemento di un frammento di interazione, una continuazione indica il punto dove finisce il frammento e che il frammento può essere continuato da un altro frammento.

Quando è il primo elemento di un frammento di interazione, una continuazione indica che questo frammento è la continuazione di un altro frammento.

204



205



206



La seconda interazione include la prima e prosegue da una delle sue continuazioni. Le continuazioni hanno permesso di disaccoppiare le due interazioni e quindi potenzialmente riusarle con altre interazioni.

ATTENZIONE: le continuazioni iniziano e finiscono le interazioni.

ATTENZIONE: le continuazioni hanno senso solo quando c'è almeno una sequenzialità debole in un'interazione.

ATTENZIONE: le continuazioni devono coprire tutte le linee di vita dei frammenti che le includono.

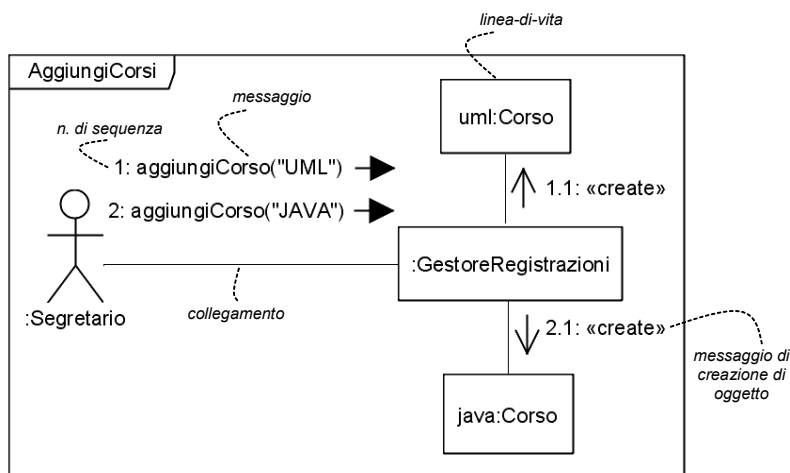
207

## Diagrammi di comunicazione

I **diagrammi di comunicazione** enfatizzano gli aspetti strutturali di un'interazione, cioè come le linee di vita si connettono.

La sintassi dei diagrammi di comunicazione è simile a quella dei diagrammi di sequenza eccetto che le linee di vita non hanno le code tratteggiate, ma sono connesse tramite collegamenti che forniscono canali di comunicazione per i messaggi.

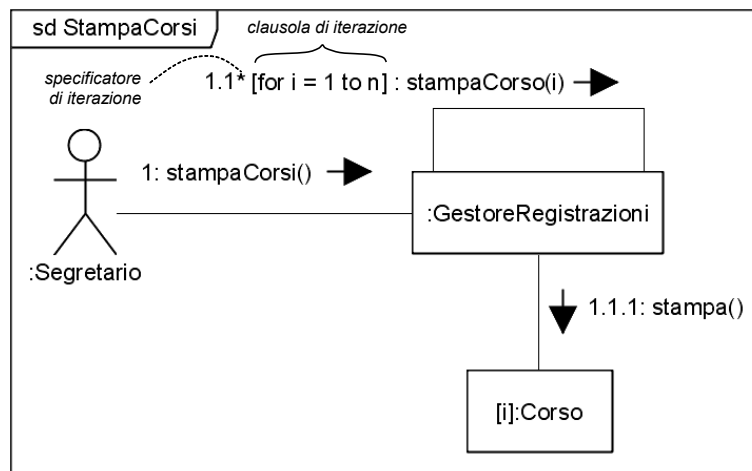
La sequenza è indicata numerando ogni messaggio gerarchicamente.



208

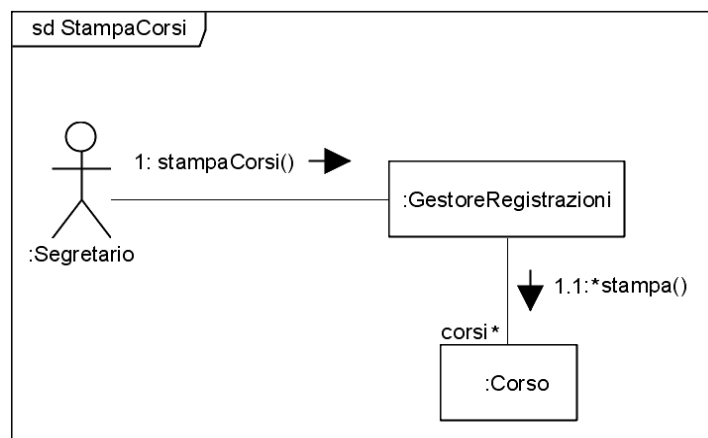
Nel diagramma seguente, viene mostrato come si possa realizzare un'iterazione nei diagrammi di comunicazione. L'espressione di iterazione comprende lo specificatore di iterazione (\*) e una clausola di iterazione.

NOTA BENE: UML 2 non prescrive nessuna sintassi particolare per le clausole di iterazione (possono quindi essere usati per chiarezza del codice o dello pseudo-codice).



Nel diagramma seguente, la collaborazione avviata dal messaggio stampaCorsi() è un esempio di iterazione. Nota che non viene usata esplicitamente una collezione di corsi, ma viene presentato attraverso la molteplicità che il GestoreRegistrazioni è connesso con una collezione di oggetti attraverso il ruolo *corsi*.

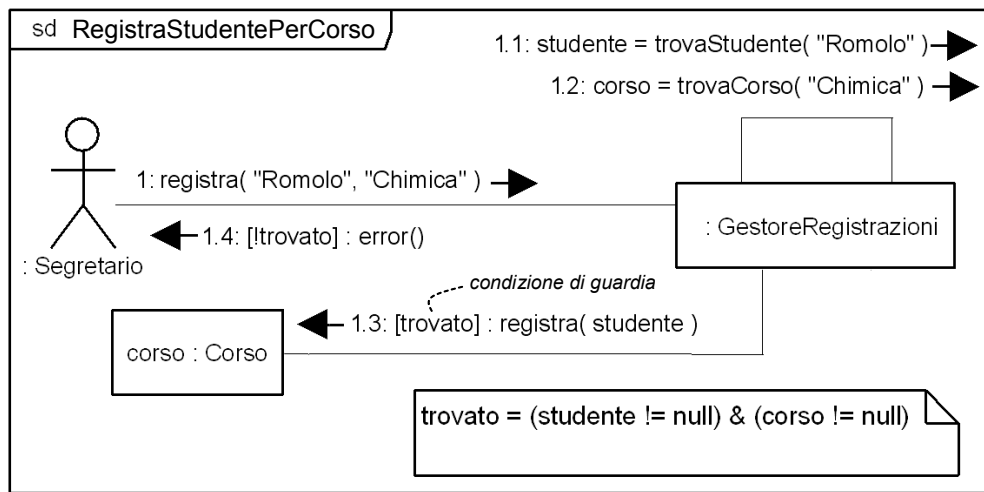
209



Nel diagramma seguente viene invece utilizzata una diramazione con condizioni di guardia inserite nei messaggi.

CONSIGLIO: presenta diramazioni molto semplici sui diagrammi di comunicazione: è più facile mostrare diramazioni complesse nei diagrammi di sequenza.

210



211

## Diagrammi di attività

I diagrammi di attività permettono di modellare un processo come un'attività che consiste in una collezione di nodi connessi da archi.

In UML 2.0 la notazione usata per i diagrammi di stato si avvicina al formalismo delle reti di Petri, con il vantaggio di una maggiore flessibilità ed una chiara distinzione tra diagrammi di attività e di macchine a stati.

Un'attività può essere associata ad un qualsiasi elemento di modellazione (casi d'uso, classi, interfacce, componenti, collaborazioni ed operazioni) allo scopo di descrivere il suo comportamento dinamico in modo facilmente comprensibile.

Un buon diagramma di attività deve focalizzarsi sul comunicare uno specifico aspetto del comportamento dinamico del sistema.

NOTA BENE: la caratteristica unica dei diagrammi di attività è che ti permettono di modellare un processo senza dover specificare la struttura statica delle classi e degli oggetti che realizzano il processo.

Quando usare i diagrammi di attività?

- Nel workflow Analisi

212

- Per modellare il flusso in un caso d'uso in un modo grafico che è facile da capire per le persone coinvolte nel progetto;
- Per modellare il flusso tra i casi d'uso. In questo caso, si usa una forma speciale di diagramma di attività chiamato un diagramma di visione generale dell'interazione (interaction overview diagram).
- Nel workflow Progetto
  - Per modellare i dettagli di un'operazione;
  - Per modellare i dettagli di un algoritmo;
- Nella modellazione del business
  - Per modellare un processo di business.

**Le attività sono reti di nodi connessi da archi.**

Ci sono tre categorie di nodi:

1. **Nodi azione** – rappresentano unità discrete di lavoro che sono atomiche nell'attività;

213

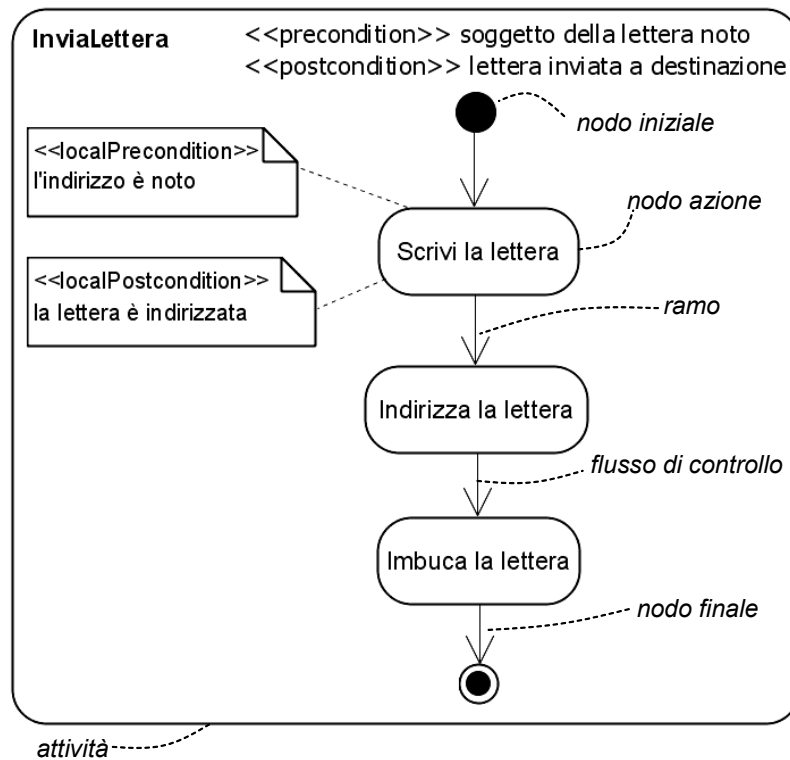
2. **nodi controllo** – controllano il flusso dell'attività;

3. **nodi oggetto** – rappresentano oggetti usati nell'attività.

Gli archi rappresentano **un flusso attraverso l'attività**. Ci sono due categorie di archi:

1. **flussi di controllo** – rappresentano il flusso di controllo nell'attività;
2. **flussi di oggetti** – rappresentano il flusso di oggetti attraverso l'attività.

214



In modo simile ai casi d'uso, le attività possono avere pre-condizioni (condizioni che devono essere vere prima che l'attività venga avviata) e post-condizioni (condizioni che saranno vere

215

quando l'attività sarà terminata). Anche le azioni (i nodi di azione) all'interno delle attività possono avere pre-condizioni e post-condizioni locali.

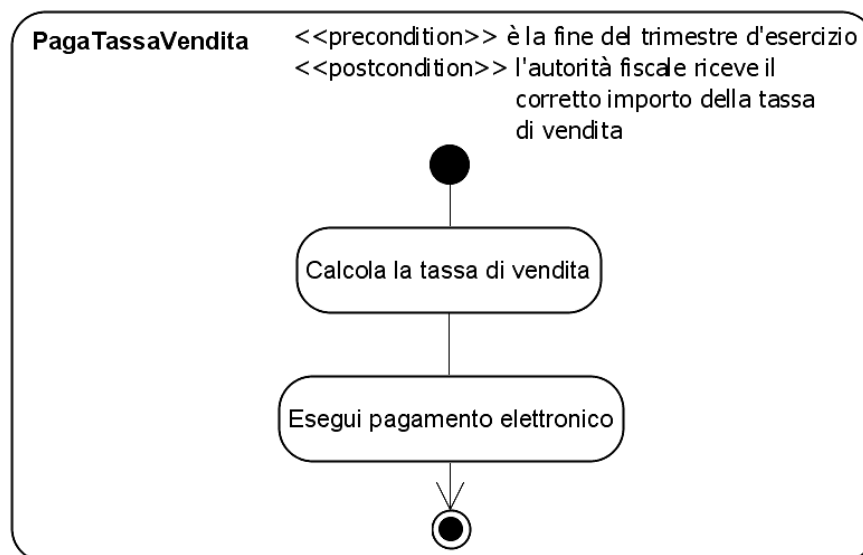
I casi d'uso esprimono il comportamento del sistema come un'interazione tra un attore ed il sistema, mentre i diagrammi di attività lo esprimono come una serie di azioni. I casi d'uso e i diagrammi di attività sono viste complementari dello stesso comportamento.

Consideriamo il caso d'uso seguente:

216

<b>Use case:</b> <i>PagaTassaVendita</i>
<b>ID:</b> 1
<b>Brief description:</b> <i>Paga la tassa di vendita all'autorità fiscale alla fine del trimestre d'esercizio</i>
<b>Primary actors:</b> <i>tempo</i>
<b>Secondary actors:</b> <i>autorità fiscale</i>
<b>Preconditions:</b> <i>1. È la fine del trimestre d'esercizio</i>
<b>Main flow:</b> <i>1. Il caso d'uso inizia quando il trimestre d'esercizio termina 2. Il sistema determina l'ammontare della tassa di vendita dovuta all'autorità fiscale 3. Il sistema esegue un pagamento elettronico all'autorità fiscale</i>
<b>Postconditions:</b> <i>1. L'autorità fiscale riceve il corretto importo della tassa di vendita</i>
<b>Alternative flows:</b> <i>nessuno</i>

217



I diagrammi di attività sono basati sulle reti di Petri. Il comportamento viene modellato attraverso il flusso di gettoni (token) lungo la rete di nodi e archi secondo regole specifiche. I token in UML rappresentano:

- Il flusso di controllo;
- un oggetto;

218

- alcuni dati.

Lo stato del sistema è determinato dalla disposizione dei token.

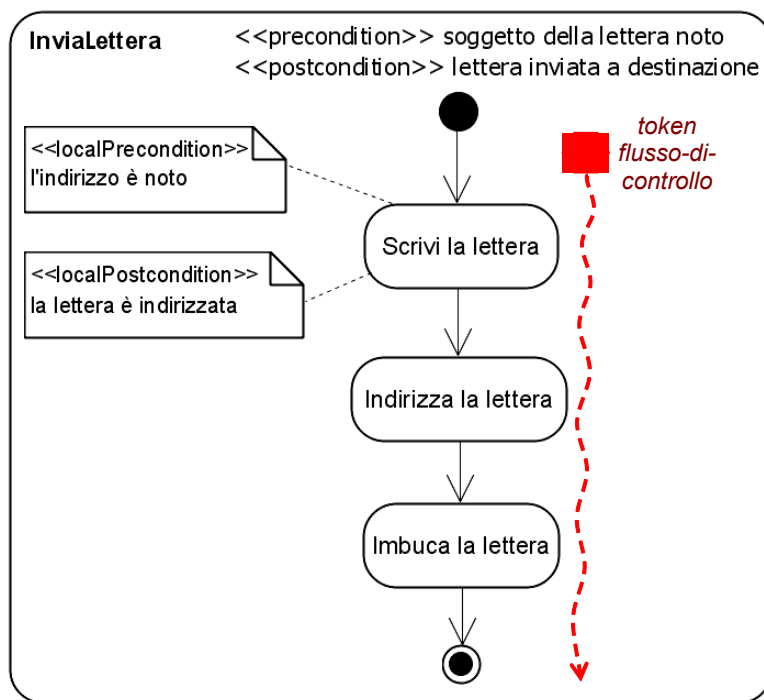
I token si muovono da un nodo sorgente ad uno destinatario lungo un arco. Il movimento dei token è soggetto a condizioni e può verificarsi quando tutte le condizioni sono soddisfatte.

Per i **nodi azione**, tali condizioni sono le post-condizioni del nodo sorgente (ossia l'attività del nodo sorgente si è conclusa), le condizioni di guardia dell'arco e le pre-condizioni dei nodi di destinazione.

Per i **nodi controllo**, vi sono speciali semantiche per decidere su quali archi di uscita proseguono i token. Ad esempio, il nodo iniziale fa "apparire" un nuovo token, il nodo finale lo fa "scompare", un nodo di unione (barra di sincronizzazione) presenta un token su ogni singola uscita se e solo se vi sono token su tutti i suoi archi di ingresso.

I **nodi oggetto** rappresentano oggetti che scorrono lungo il sistema.

219



Nella figura il flusso di controllo (visivamente il token) passa attraverso un nodo azione alla volta e ne causa l'esecuzione.

220

La disposizione dei token può rappresentare lo stato del sistema. Per esempio, quando il token è nell'attività "Scrivi lettera", il sistema si trova nello stato di "Scrittura lettera". Comunque, non ogni azione di esecuzione o flusso di token costituisce un rilevante cambio di stato nel sistema.

ATTENZIONE: verificate che i diagrammi di attività e delle macchine a stati siano sempre consistenti tra di loro.

Il "percorso dei token" rappresenta uno schema descrittivo e non implementativo. È difficile che il concetto di token venga implementato in un sistema software. Una attività è solo una specifica alla quale possono corrispondere molte implementazioni.

## ***Partizioni di Attività***

È utile suddividere le attività in **partizioni**, ossia delle "corsie" (swimlane) orizzontali o verticali che raggruppano azioni correlate. In UML 2.0, è colui che costruisce il modello a definire la semantica delle partizioni. Le attività possono essere raggruppate, per esempio, per responsabilità o per sequenza di esecuzione. Le partizioni sono usate per rappresentare:

- casi d'uso,
- classi,
- componenti,

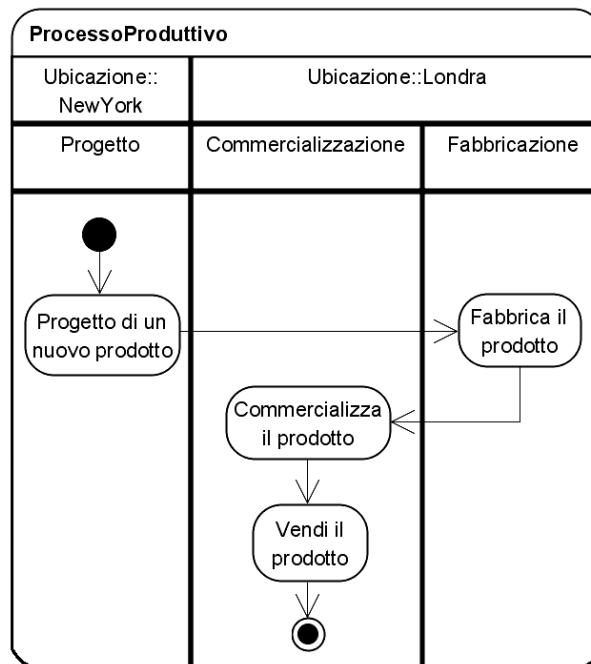
221

- unità organizzative (in un processo aziendale),
- ruoli (nella modellazione di workflow),
- unità di elaborazione (nella modellazione del progetto),
- ecc..

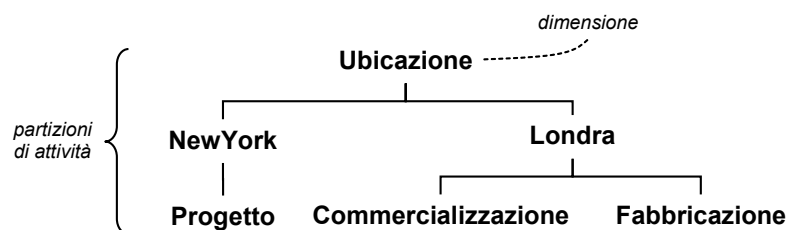
Ogni insieme di partizioni dovrebbe avere una dimensione singola che descrive la semantica di base dell'insieme. All'interno di questa dimensione, le partizioni possono essere organizzate in modo gerarchico. Nella figura seguente, la dimensione Ubicazione stabilisce la semantica delle partizioni *New York* e *Londra*. All'interno abbiamo un'altra partizione basata sui diversi *reparti*: *Progetto*, *Commercializzazione* e *Fabbricazione*.

222





223



**ATTENZIONE:** per non complicare eccessivamente la notazione, cerca di non usare più di tre livelli in una gerarchia e più di due dimensioni per diagramma.

## Nodi Azione

I nodi azione vengono eseguiti quando:

- c'è un token su ognuno dei loro archi di ingresso (AND logico degli ingressi);
- i token soddisfano tutte le pre-condizioni locali del nodo azione.

Quando il nodo azione finisce l'esecuzione, se le post-condizioni sono verificate, il nodo invia token su tutti i suoi archi d'uscita (*fork* implicito).

**NOTA BENE:** i diagrammi di attività sono concorrenti.

224

Il **nome** di un nodo azione è un verbo o una frase con verbo. Adottiamo lo stile di nominare il nodo azione con una frase la cui prima parola inizia con una lettera maiuscola, mentre le altre iniziano con lettere minuscole. Se, comunque, una parola si riferisce ad un modello, usa il nome del modello come è. Usa gli spazi tra parole quando necessario.

Crea ordine


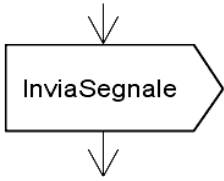
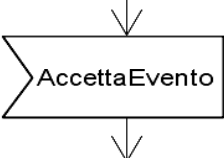
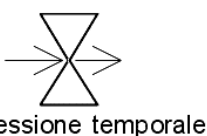
“ordine” si riferisce ad un ordine di qualche tipo

Crea Ordine

“Ordine” si riferisce ad un elemento di modello chiamato Ordine

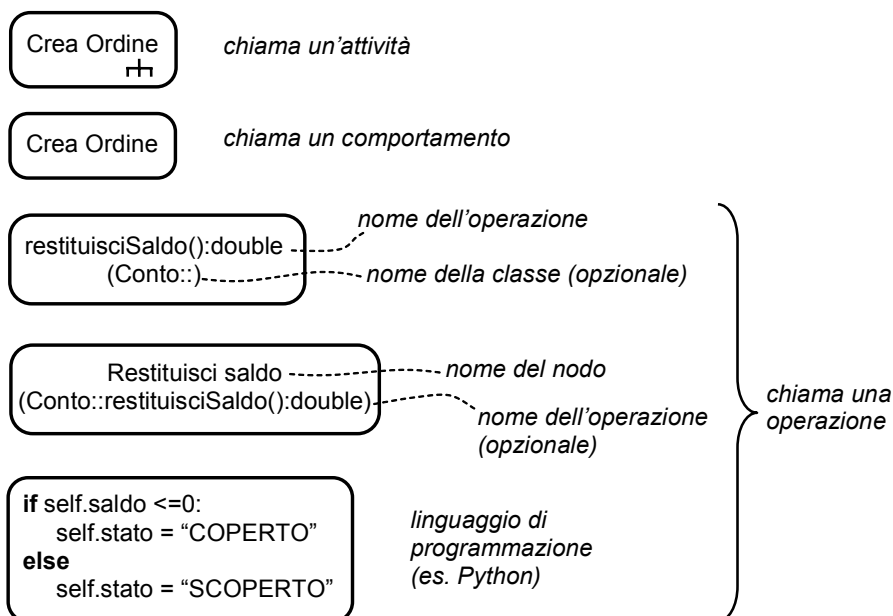
Dettagli dell’azione sono dati nella specifica del nodo azione. Mentre in analisi questa specifica è una descrizione testuale semplice, nel progetto può essere un testo strutturato o uno pseudo-codice.

La tabella seguente riporta i tipi di nodi azione.

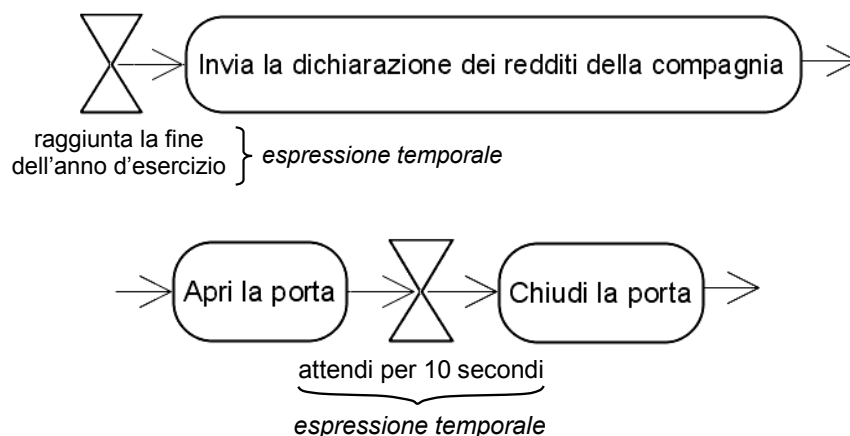
Sintassi	Nome	Semantica
	Invoca (Call Action node)	Invoca un’attività, un comportamento, oppure un’operazione
	Invia segnale (Send signal Action node)	Invia un segnale in modo asincrono
	Accettazione di Evento (Accept event action node)	Aspetta gli eventi monitorati dal suo oggetto proprietario e offre l’evento al suo arco di uscita.
 espressione temporale	Accettazione di Evento temporale (Accept time event action node)	Accetta un evento temporale

Si può indicare che l'azione invoca un'altra attività usando il simbolo del rastrello nella parte più in basso a destra dell'icona.

Di seguito vengono dati alcuni esempi di sintassi di nodo azione.



227






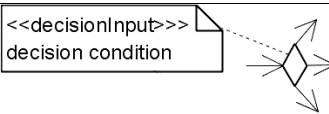
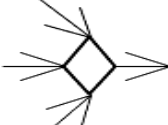
L'espressione temporale può riferirsi a:

- un evento (ad esempio, fine dell'anno);
- un punto nel tempo (ad esempio, 11/03/1960)
- una durata (ad esempio, aspetta 10 secondi).

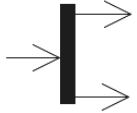
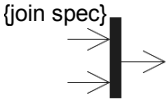
228

## Nodi Controllo

Gestiscono il flusso di controllo all'interno di un'attività.

Sintassi	Nome	Semantica
	Nodo iniziale (Initial node)	Indica dove inizia il flusso quando viene invocata un'attività
	Nodo finale dell'attività (Activity final node)	Termina un'attività
	Nodo finale del flusso (Flow final node)	Termina un flusso specifico all'interno dell'attività
	Nodo di decisione (Decision node)	Viene attraversato l'arco di uscita la cui condizione di guardia è vera
	Nodo di combinazione (Merge node)	Copia i token d'ingresso al singolo arco di uscita

229

	Nodo di diramazione (Fork node)	separa il flusso in flussi multipli concorrenti
	Nodo di sincronizzazione (Join node)	Sincronizza flussi multipli concorrenti

NOTA BENE: un'attività può avere più nodi iniziali. In questo caso, tutti i flussi iniziano simultaneamente e procedono concorrentemente.

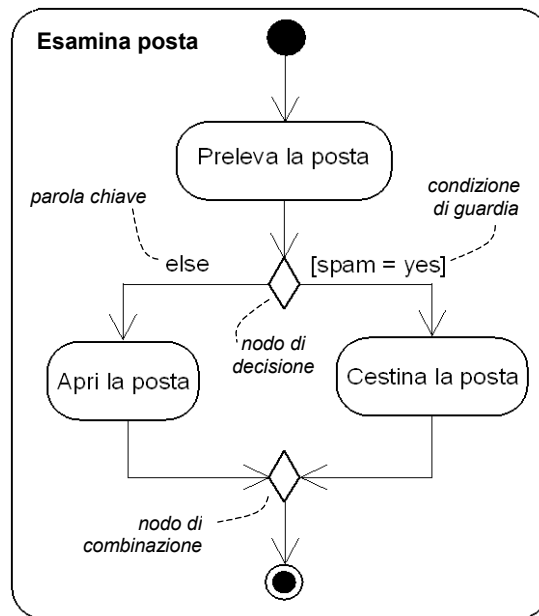
NOTA BENE: un'attività può avere più nodi finali. In questo caso, il primo nodo finale che termina fa terminare l'attività.

Il nodo finale di flusso termina il flusso a cui corrisponde e non l'attività.

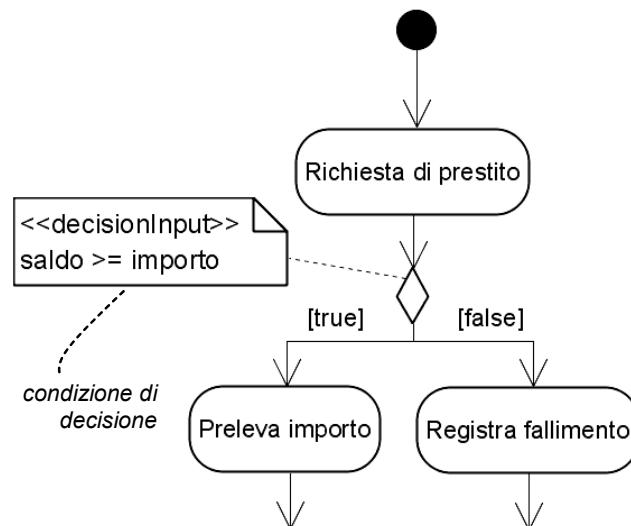
NOTA BENE: nei nodi di decisione, le guardie devono essere mutuamente esclusive; altrimenti il comportamento del nodo di decisione è formalmente indefinito in accordo alla specifica UML 2.0.

Esempio di nodi di decisione e di combinazione

230

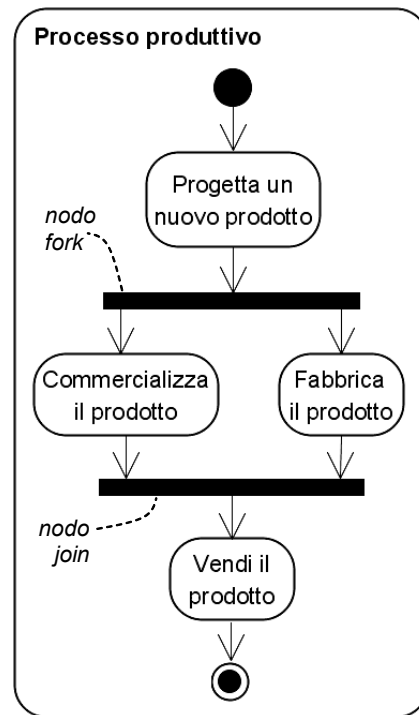


231



Esempio di nodi di fork e join.

232



NOTA BENE: quando modelli i nodi di sincronizzazione è importante assicurare che tutti gli archi riceveranno un token. Fai attenzione a non avere archi con guardie mutuamente esclusive.

233

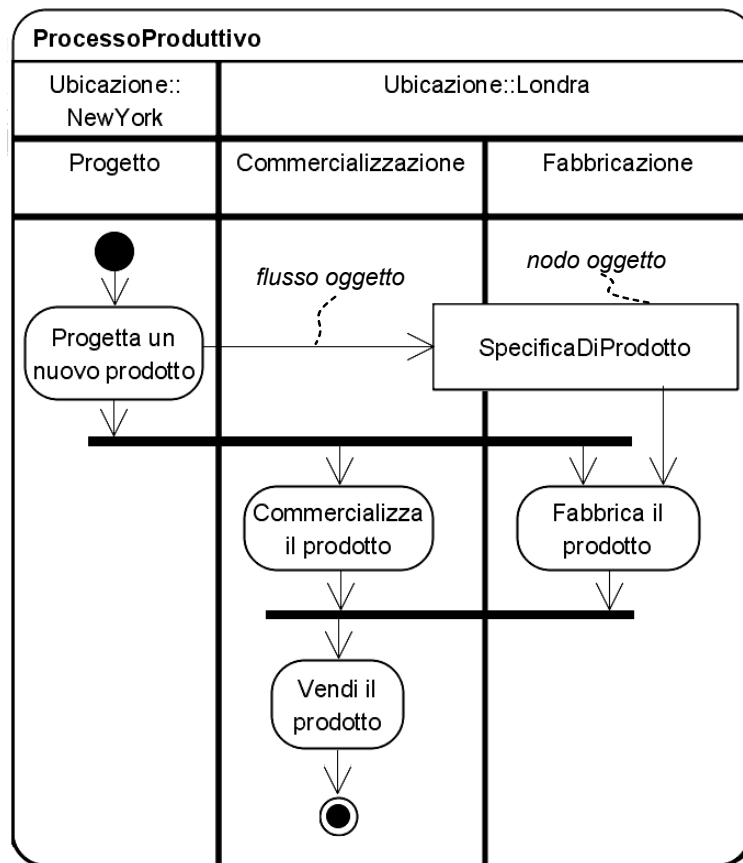
## ***Nodi oggetto***

I nodi oggetto sono nodi speciali che indicano che istanze di un particolare classificatore sono disponibili ad un punto specifico nell'attività.

I nodi oggetto hanno lo stesso nome del classificatore e rappresentano istanze del classificatore o delle sue sottoclassi.

Gli archi di ingresso e di uscita di nodi oggetto rappresentano flussi di oggetti. Gli oggetti sono creati e consumati da nodi azione.

234



235

I nodi oggetto agiscono come buffer dove gli oggetti possono essere memorizzati mentre sono in attesa di essere accettati da altri nodi.

Per default, ogni nodo oggetto può contenere un numero infinito di token.

Si può comunque specificare per il buffer:

- un limite superiore;
- un ordinamento – specifica come il buffer gestisce gli oggetti;
- un comportamento di selezione – permette al buffer di selezionare oggetti dagli archi di ingresso in accordo a qualche criterio definito da chi realizza il modello.

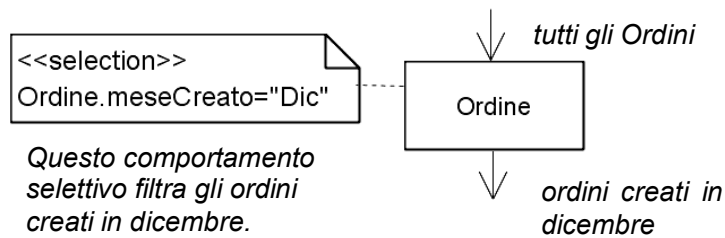
Questo nodo oggetto può gestire un massimo di 12 token oggetto

**Ordine**

L'ultimo oggetto ad entrare è il primo ad uscire.

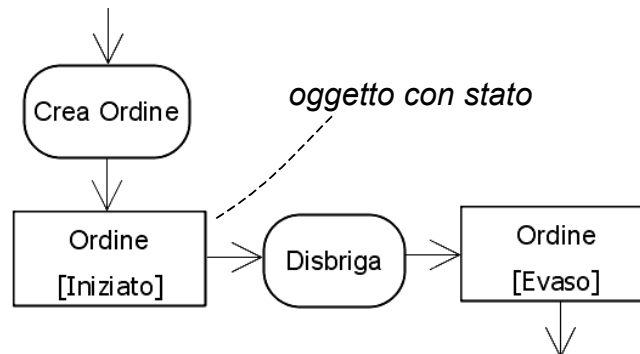
{ upperBound = 12}  
{ ordering = LIFO }

236



NOTA BENE: il nodo oggetto può anche rappresentare un insieme (ricorda un insieme è un gruppo di oggetti in cui non ci sono duplicati).

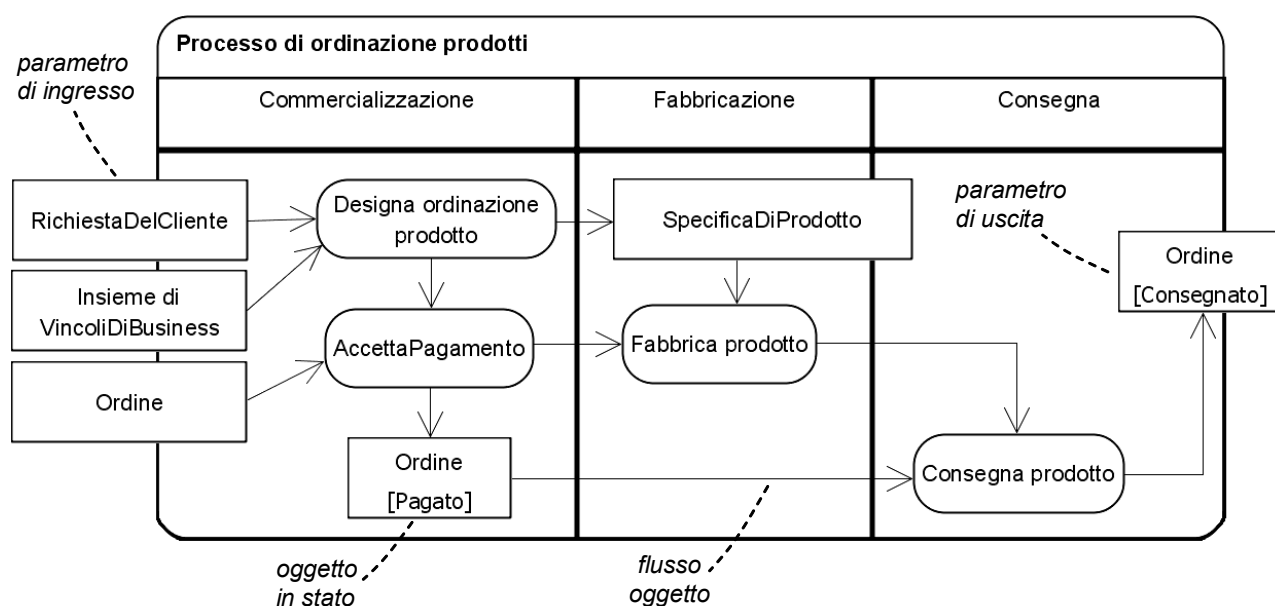
NOTA BENE: i nodi oggetto possono anche rappresentare oggetti in uno stato particolare.



237

## Parametri dell'Attività

I nodi oggetto possono essere anche usati per fornire ingressi ed uscite alle attività. I nodi oggetto di ingresso e di uscita dovrebbero essere disegnati sovrapposti alla cornice dell'attività.



238

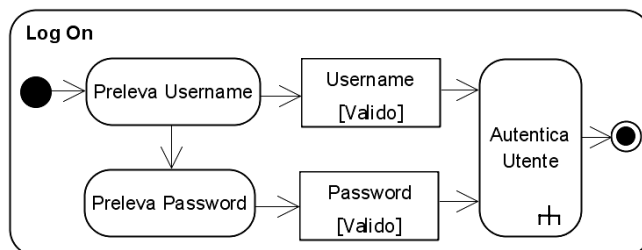


OSSERVA come il diagramma di attività riesca a realizzare i vincoli di business imposti dall'applicazione.

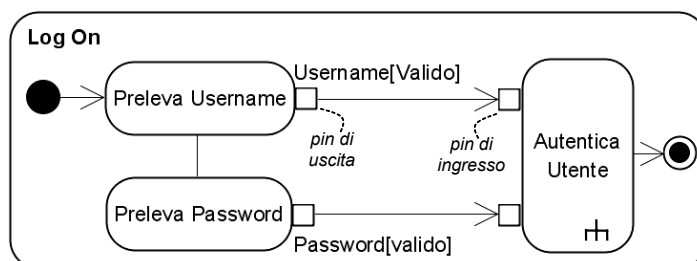
## ***PIN***

Un pin è un nodo oggetto che rappresenta un ingresso o un'uscita in un'azione.

Un pin di ingresso ha esattamente un arco di ingresso ed un pin di uscita ha esattamente un arco di uscita.



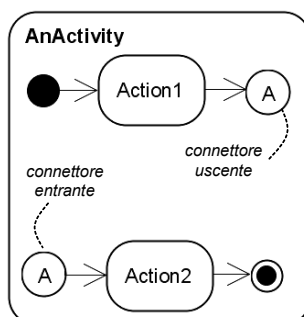
239



## ***Diagrammi avanzati di attività***

### ***Connettori***

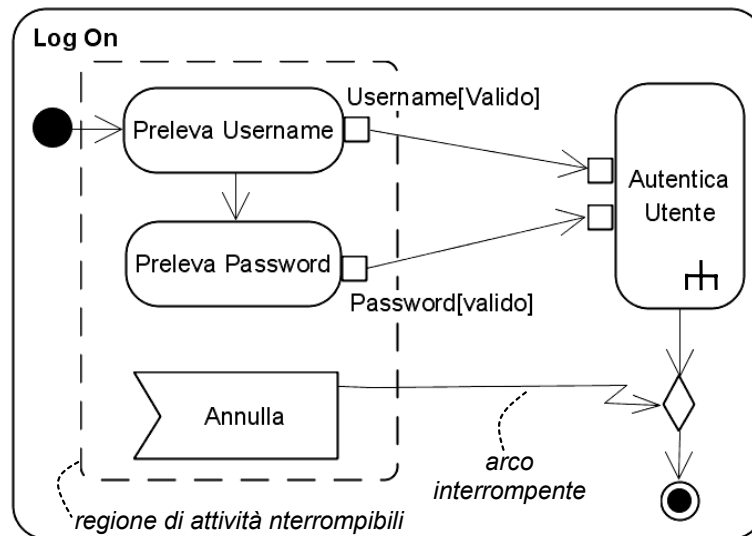
I connettori vengono utilizzati per rompere archi molto lunghi che sono difficili da seguire o per eliminare intersezioni tra gli archi.



240

## Regioni di attività interrompibili

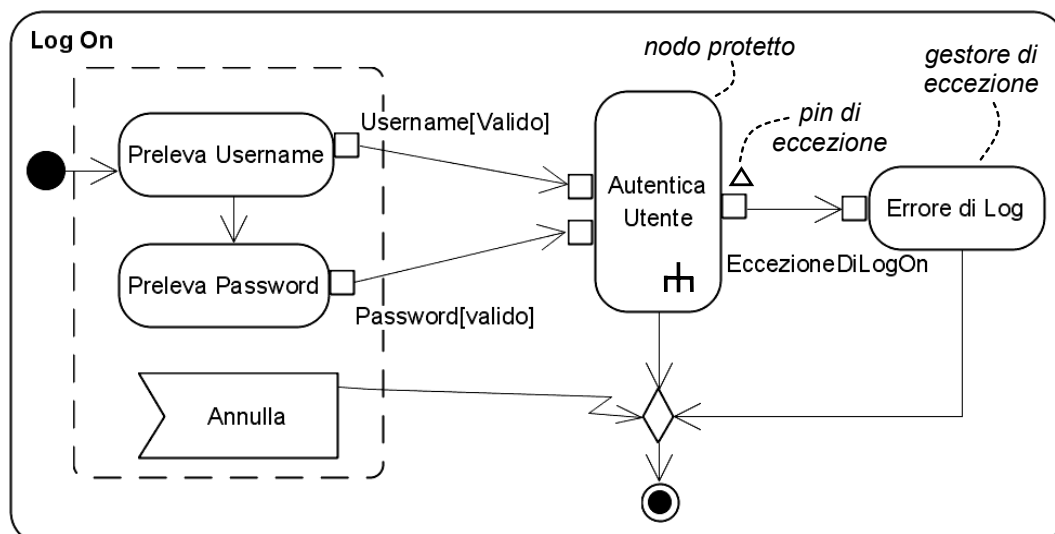
Sono regioni di un'attività che sono interrotte quando un token attraversa un arco interrompente (*interrupting edge*). Quando la regione è interrotta, tutti i flussi all'interno della regione sono immediatamente abortiti. Regioni di attività interrompibili forniscono un modo utile per modellare interruzioni ed eventi asincroni.



241

## Gestione delle eccezioni

Quando un errore si manifesta in un pezzo di codice protetto, viene creato un oggetto eccezione ed il flusso di controllo salta ad un handler di eccezione che processa l'oggetto eccezione in qualche modo. L'oggetto contiene informazione sull'errore e viene usato dallo handler.



Un nodo è conosciuto come nodo protetto quando ha associato un handler di eccezione.

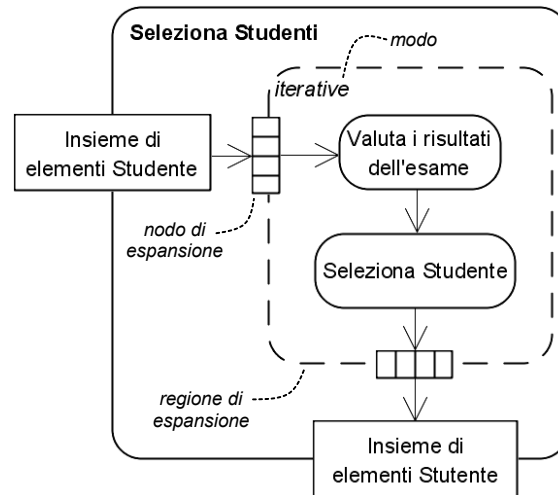
242

NOTA BENE: la gestione dell'eccezione è spesso un problema di progetto piuttosto che di analisi.

## Nodi di espansione

Un nodo di espansione permette di presentare come una collezione di oggetti è processata da una parte del diagramma di attività chiamata *regione di espansione*.

La regione di espansione è eseguita per ogni elemento.



243

Ci sono due vincoli sui nodi di espansione:

- Il tipo della collezione di uscita deve coincidere con il tipo della collezione di ingresso
- Il tipo degli oggetti nelle collezioni di ingresso e di uscita deve essere lo stesso.

Il numero di collezioni di uscita può essere differente dal numero di collezioni di ingresso, così le regioni di espansione possono essere usate per combinare o separare collezioni.

Ogni regione di espansione ha un modo che determina l'ordine in cui esso processa gli elementi della sua collezione di ingresso.

I modi possono essere:

- Iterative – ogni elemento della collezione è processato sequenzialmente;
- Parallel – ogni elemento della collezione è processato parallelamente;
- Stream – processa ogni elemento della collezione di ingresso come arriva al nodo.

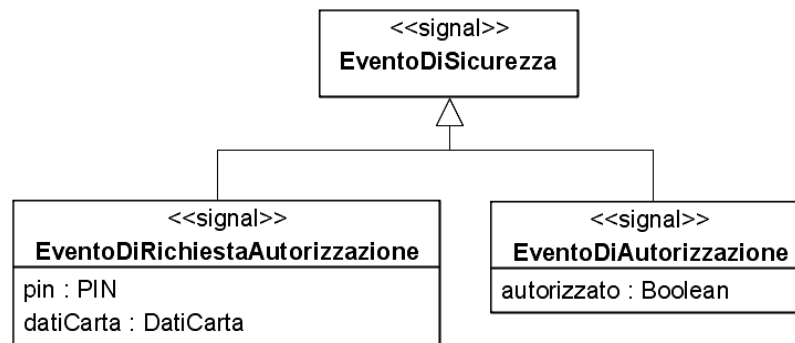
L'esempio precedente mostra un modo iterativo. Se il modo fosse stato stream, allora ogni studente sarebbe stato inviato in uscita non appena processato.

244

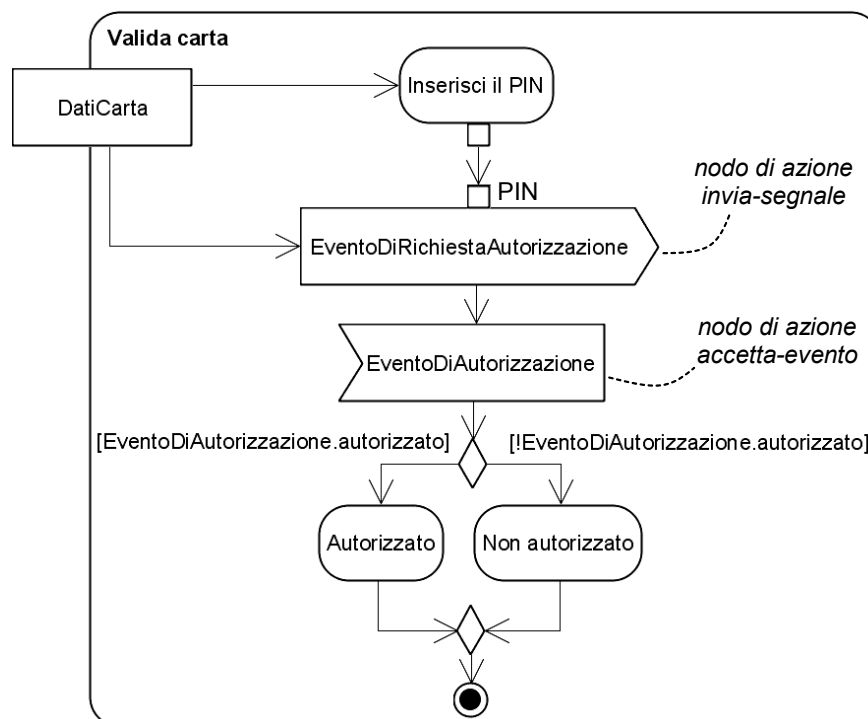
## Inviare segnali ed accettare eventi

Un segnale rappresenta informazione che è passata in modo asincrono tra oggetti. L'informazione è mantenuta negli attributi del segnale.

In analisi, i segnali vengono utilizzati per presentare invio o ricezione di eventi di business asincroni. Nel progetto, i segnali vengono utilizzati per mostrare una comunicazione asincrona tra sistemi differenti, sottosistemi o pezzi di hardware.



245



La semantica del nodo di azione “invia segnale” è la seguente:

- L'azione di invio del segnale è attivata quando su tutti gli archi di ingresso è presente un token;

246

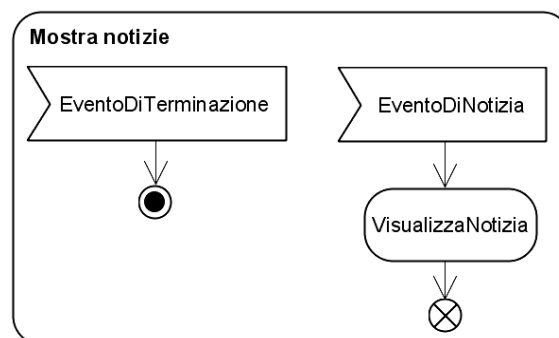
- Quando l'azione viene eseguita, viene costruito ed inviato un segnale.
- L'azione di invio non aspetta per la conferma (è asincrona);
- Quando l'azione finisce, i token sono inviati negli archi di uscita.

Il nodo azione “accetta evento” ha al più un arco di ingresso. La semantica del nodo azione “accetta evento” è la seguente:

- L'azione di invio del segnale è attivata da un arco di controllo entrante, o se non ha un arco entrante, quando l'attività inizia;
- L'azione aspetta di ricevere un evento del tipo specificato. L'evento è conosciuto come il trigger.
- Quando l'azione riceve il trigger, produce in uscita un token che descrive l'evento. Se l'evento era un segnale, il token è un segnale.
- L'azione continua ad accettare eventi mentre l'attività prosegue.

Nella figura seguente viene dato un altro esempio di azione “accetta evento”.

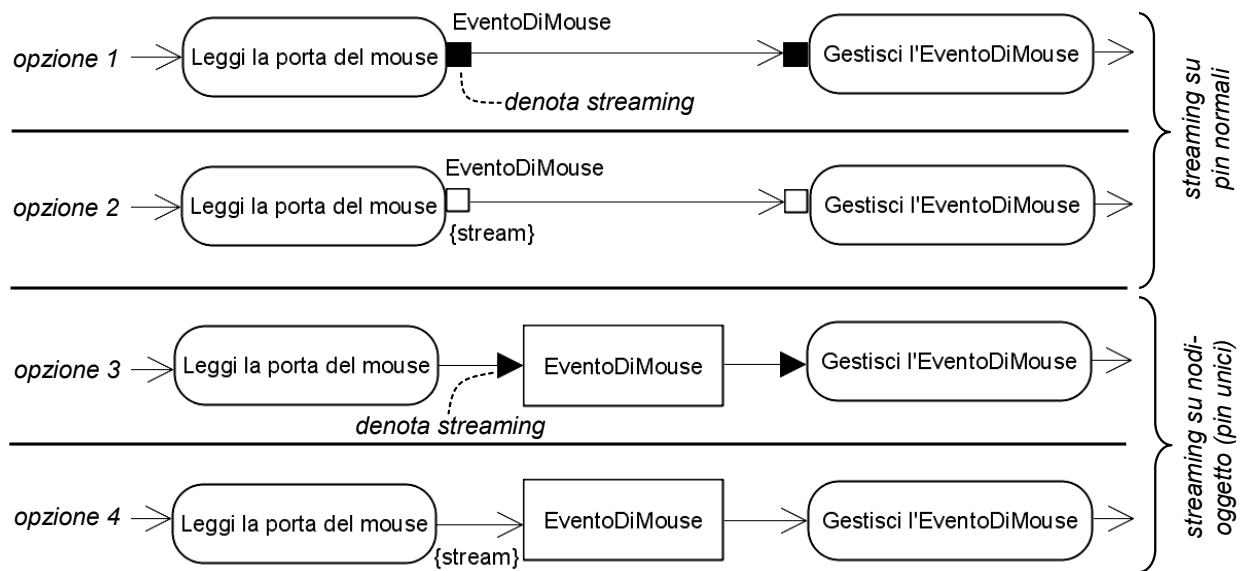
247



## Streaming

A volte, serve che un'azione venga eseguita in continuazione mentre accetta e offre token periodicamente. Questo comportamento è chiamato streaming.

248



## Effetti di ingresso e di uscita

Gli effetti di ingresso e di uscita presentano l'effetto che un'azione ha sugli oggetti di ingresso o di uscita. L'effetto è descritto ponendo una breve descrizione dell'effetto tra parentesi graffe vicino ai pin di ingresso o di uscita.

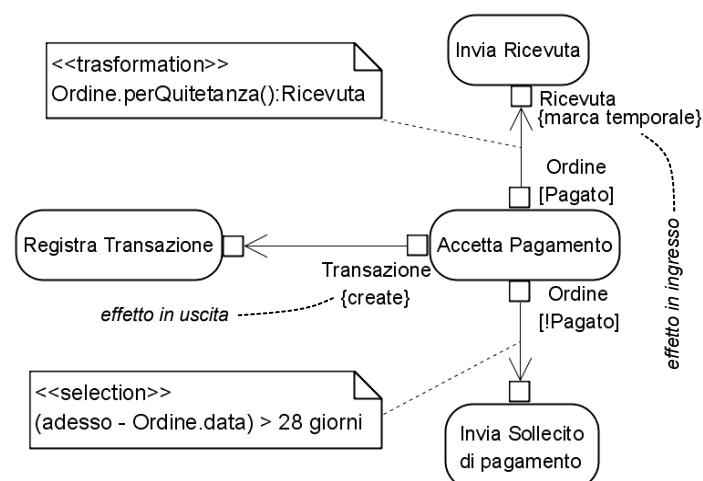
249

## Selezione

Una selezione è una condizione associata al flusso di un oggetto che consente di accettare solo quegli oggetti che soddisfano la condizione stessa. La selezione è presentata in una nota stereotipata «**selection**».

## Trasformazione

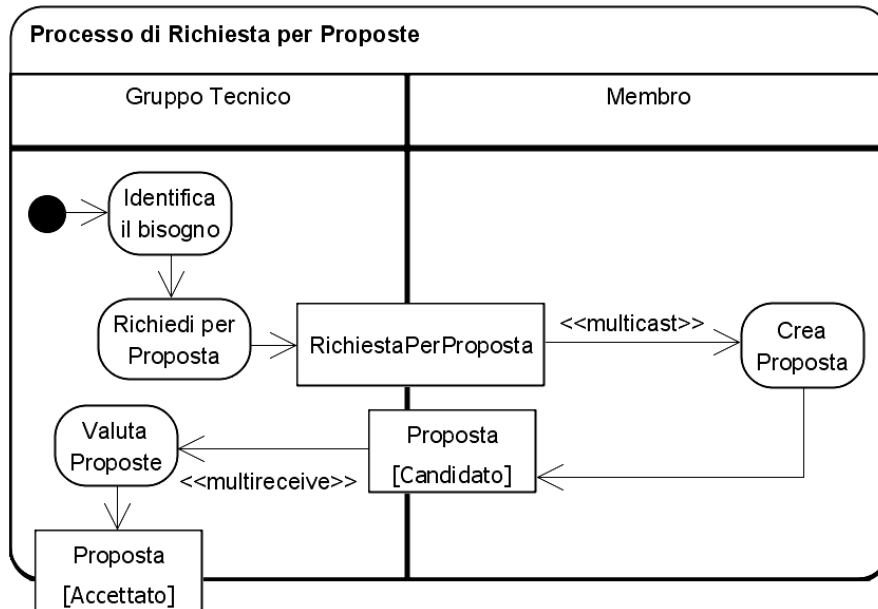
Una trasformazione trasforma oggetti da un tipo ad un altro. La trasformazione è presentata in una nota stereotipata «**transformation**».



250

## Multicast e multireceive

A volte un oggetto può essere inviato a ricevitori multipli (*multicast*) oppure oggetti sono ricevuti da mittenti multipli (*multireceive*). La figura seguente presenta un processo di business simile a quello che lo OMG usa per le sottomissioni di proposte per i propri standard.

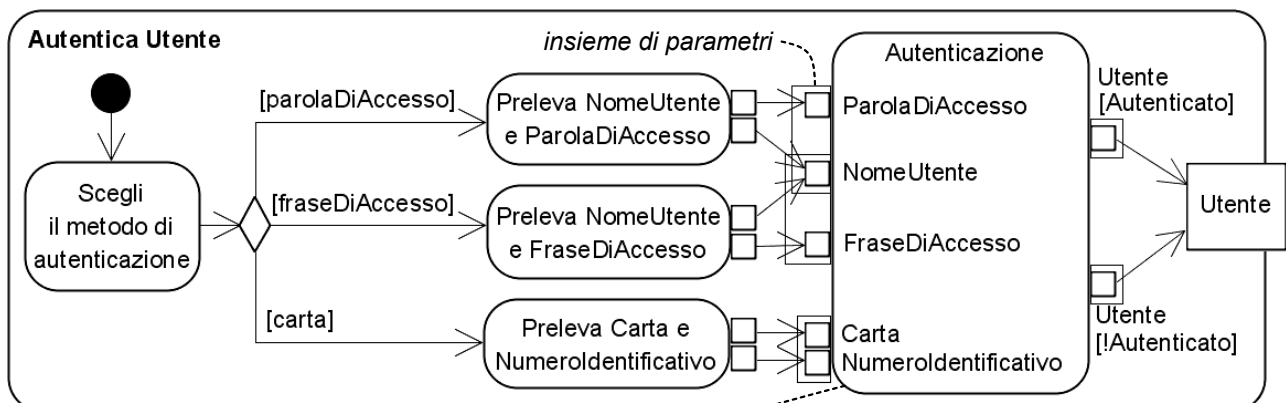


251

## Insiemi di parametri

Gli insiemi di parametri permettono ad un'azione di avere insiemi alternativi di pin di ingresso ed uscita. ATTENZIONE: i pin di ingresso non possono essere mischiati con quelli di uscita.

Analizziamo l'esempio seguente:



condizione di ingresso: (NomeUtente AND ParolaDiAccesso) XOR (NomeUtente AND FraseDiAccesso) XOR (Carta AND Numeroidentificativo)  
uscita: ( Utente[Autenticato] ) XOR ( Utente[!Autenticato] )

252

Ognuna delle tre azioni (“Preleva NomeUtente e ParolaDiAccesso”, “Preleva NomeUtente e FraseDiAccesso”, “Preleva Carta e NumeroIdentificativo”) producono un differente insieme di oggetti.

Una soluzione a questo problema sarebbe di avere tre distinte azioni di autenticazione per i tre insiemi di oggetti. Questa soluzione presenta due problemi:

- Rende il diagramma di attività piuttosto pesante e ripetitivo;
- L'autenticazione sarebbe distribuita su tre azioni e non più concentrata in un unico posto.

Gli insiemi di parametri consentono di usare un'unica azione che può correttamente gestire i tre differenti ingressi.

ATTENZIONE: tra gli insiemi di parametri c'è una relazione XOR. Questo significa che solo uno degli insiemi di parametri può essere usato per l'esecuzione dell'azione.

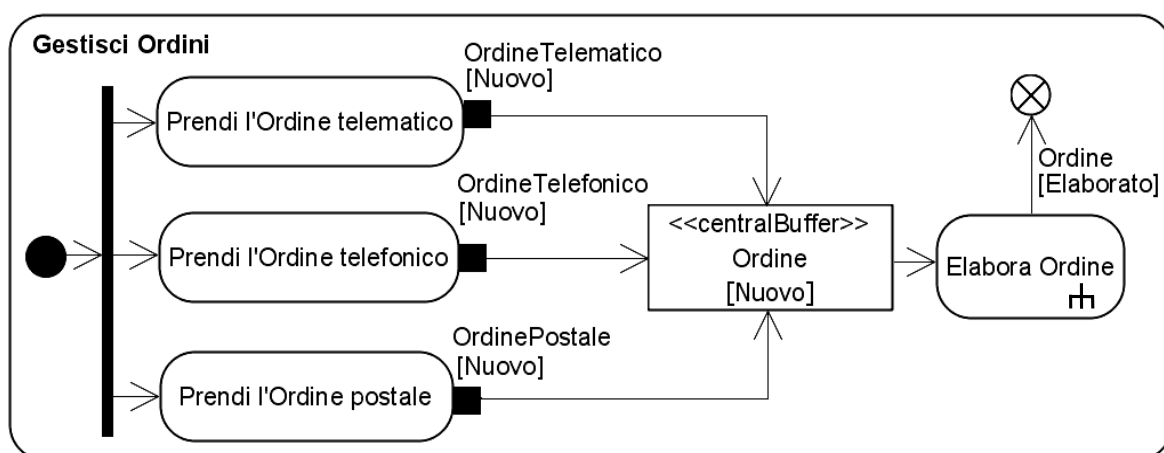
NOTA BENE: insiemi differenti possono avere pin in comune. Se il diagramma diventa troppo complesso, i pin possono essere replicati.

NOTA BENE: a destra dell'azione di autenticazione ci sono due insiemi di parametri di uscita con un unico pin.

253

## Nodo «centralBuffer»

I nodi «centralBuffer» sono nodi oggetto che sono usati specificatamente come buffer tra il flusso di ingresso ed il flusso di uscita. Questi nodi permettono di combinare flussi di oggetti multipli in ingresso e di distribuire gli oggetti tra flussi multipli di uscita.



254



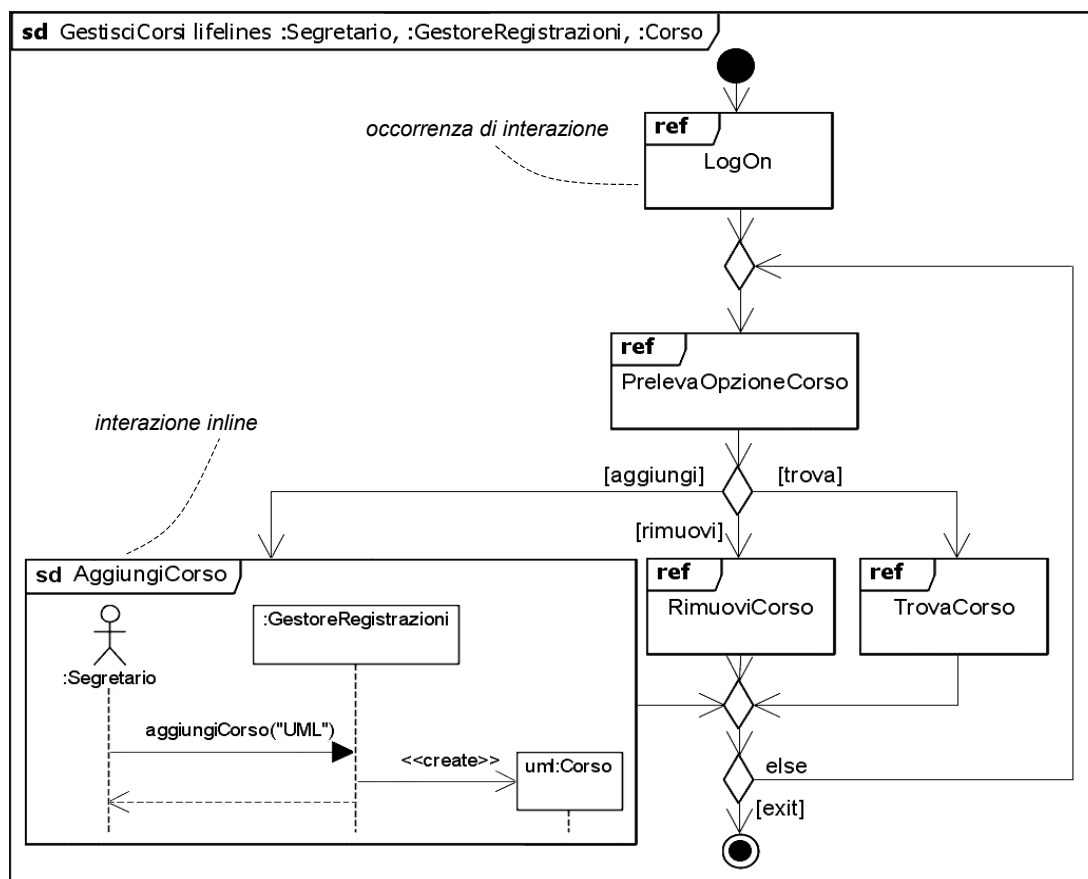
## Diagrammi di visione di insieme dell'interazione (Interaction Overview diagrams)

I diagrammi di visione di insieme dell'interazione presentano interazioni e occorrenze di interazione. Sono usati per modellare il flusso di controllo ad alto livello tra interazioni.

Questi diagrammi vengono spesso usati per illustrare il flusso di controllo tra casi d'uso. Se ogni caso d'uso è rappresentato come un'interazione, si può usare la sintassi dei diagrammi di attività per presentare come il flusso di controllo procede tra di loro.

NOTA BENE: i diagrammi di visione di insieme dell'interazione hanno la stessa sintassi dei diagrammi di attività eccetto che presentano interazioni ed occorrenze di interazioni piuttosto che nodi di attività e nodi oggetto.

255



256

NOTA BENE: le linee di vita che partecipano nell'interazione possono essere elencate dopo le parola chiave lifelines nell'intestazione del diagramma. Questo risulta utile per la documentazione visto che le lifelines sono spesso nascoste dentro le occorrenze di interazione.

La tabella seguente presenta cosa può essere usato nei diagrammi di visione d'insieme.

Azione	Diagrammi di sequenza	Diagrammi di visione d'insieme
Diramazione	alt e opt	Nodi di decisione e nodi di combinazione
Concorrenza	par	nodo fork e join
Iterazione	loop	cicli nel diagramma

I diagrammi di sequenza possono fare le stesse cose dei diagrammi di visione d'insieme dell'interazione. Questi ultimi però utilizzano una notazione visuale e si concentrano sul flusso di controllo.

NOTA BENE: i diagrammi di visione d'insieme dell'interazione possono essere utilizzati per illustrare processi di business.

# Sistemi Informativi

F. Marcelloni – M. G. Cimino



## Workflow Progetto

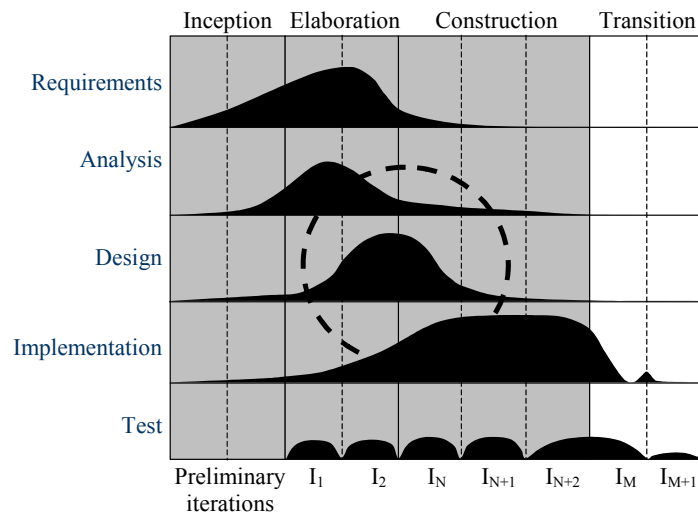
### Introduzione

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.

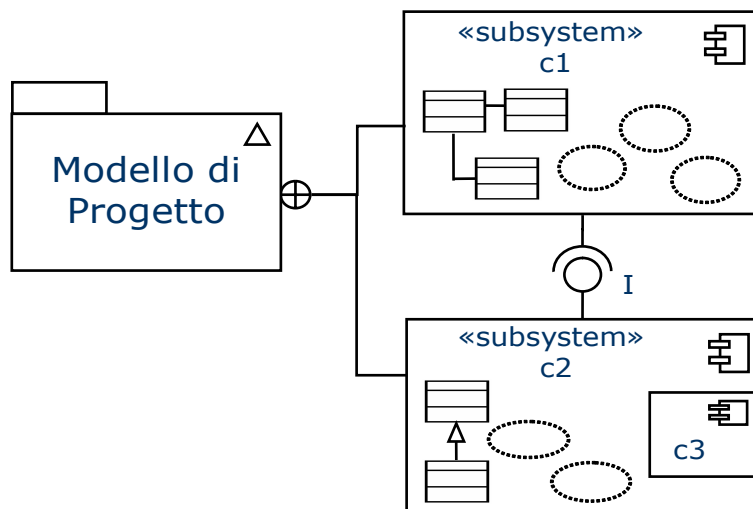
Come abbiamo visto precedentemente, i requisiti del sistema sono tipicamente estratti dal dominio del problema e l'analisi può essere considerata come l'esplorazione del dominio dal punto di vista delle persone coinvolte nel sistema. Il progetto consiste nel costruire un modello implementabile del sistema 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.

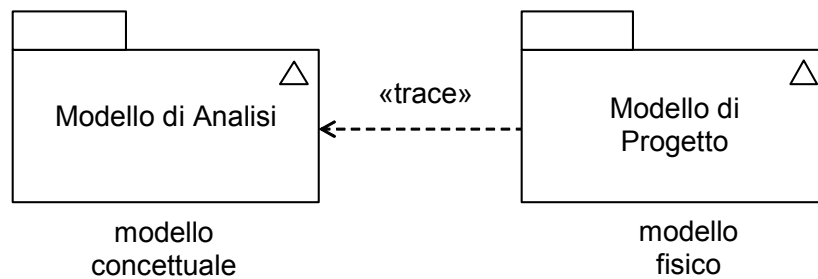
Il workflow Progetto si svolge in stretta concomitanza con il workflow Requisiti e Analisi (vedi figura seguente). Ricorda che UP raccomanda che un team sia responsabile della vita di un artefatto (per esempio, un caso d'uso) lungo i workflow Requisiti, Analisi, Progetto ed Implementazione. Infatti,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



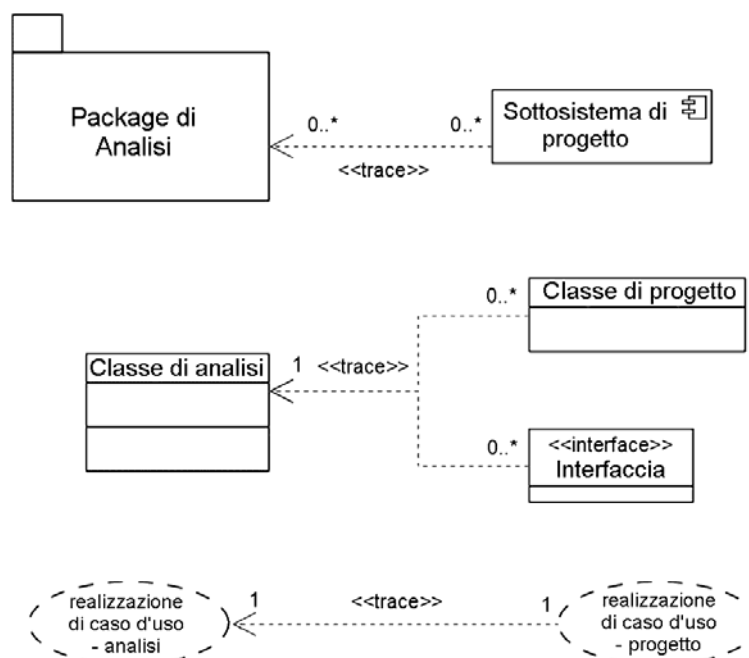
Il modello di progetto può essere visto come un'elaborazione del modello di analisi con l'aggiunta di dettagli tecnici e soluzioni specifiche: tutti gli artefatti devono essere completamente definiti e devono includere dettagli di implementazione.



I modelli di progetto sono composti da:

- Sottosistemi di progetto;
- Classi di progetto;
- Interfacce;
- Realizzazioni di casi d'uso a livello di progetto;
- Un diagramma di dislocazione.

La figura seguente mostra le relazioni tra gli artefatti principali di analisi e progetto.



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.

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.

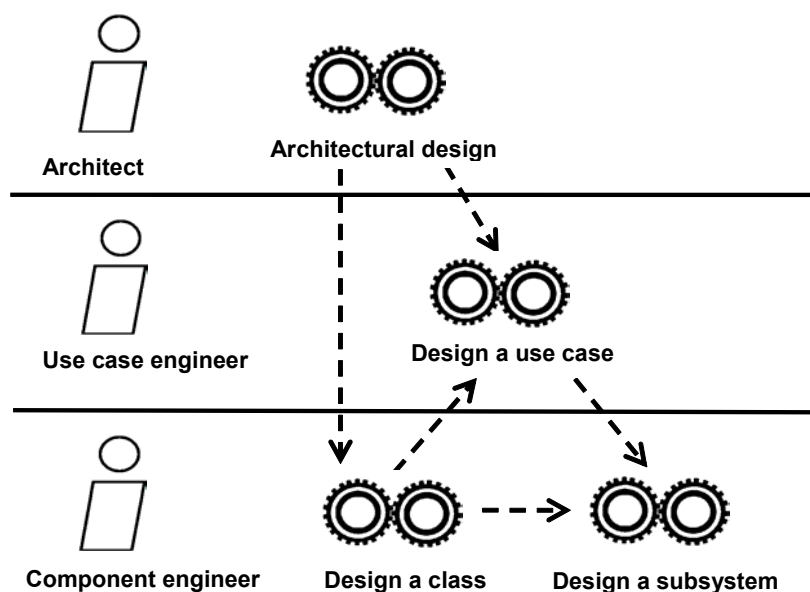
Esaminando la figura precedente, viene spontaneo domandarsi: visto che il modello di progetto rifinisce il modello di analisi, si devono mantenere entrambi i modelli?

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:

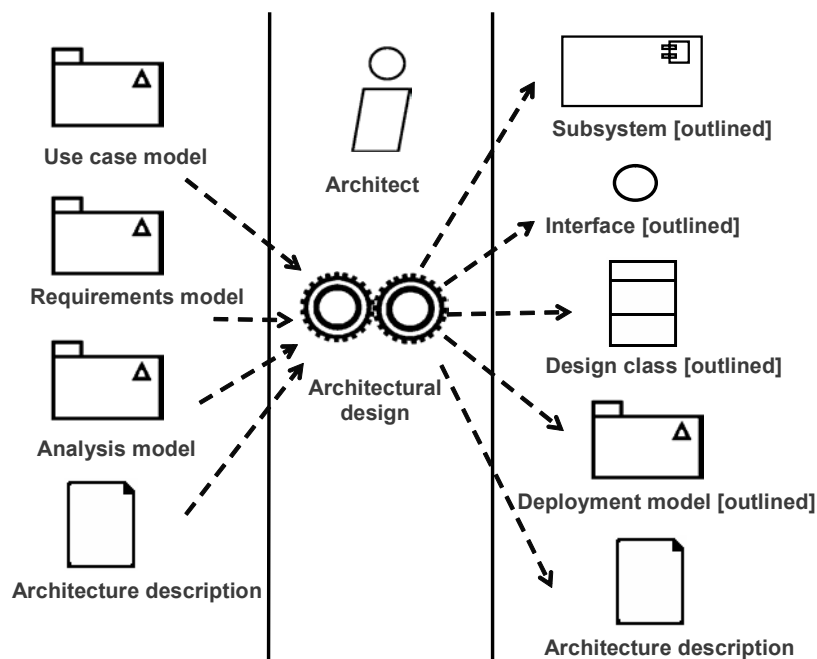
- introdurre nuove persone nel progetto;
- capire il sistema mesi o anni dopo la consegna;
- capire come il sistema soddisfa i requisiti;
- gestire la tracciabilità dei requisiti;
- pianificare la manutenzione e i miglioramenti;
- capire l'architettura logica del sistema;
- commissionare l'implementazione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

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

La figura seguente mostra le attività principali che compongono il workflow Progetto.

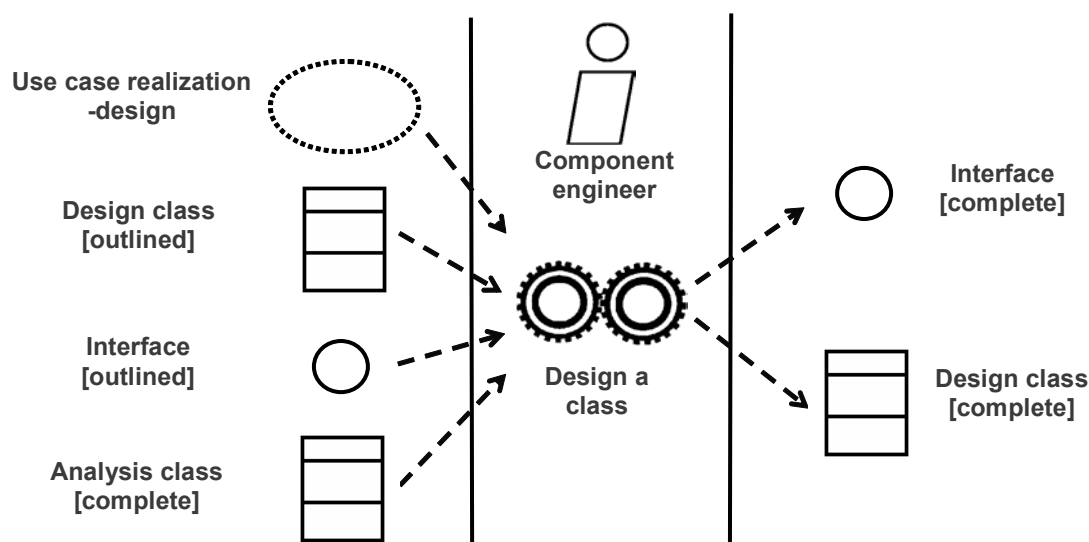


Come si può vedere dal grafico, l'attività che dà il via all'intero processo di progettazione è il Progetto Architeturale. Questa attività serve a delineare gli artefatti architetturalmente significativi per dare un quadro d'insieme dell'architettura del sistema. Questi artefatti vengono poi rifiniti nelle altre attività del workflow Progetto.



## Le classi di progetto

La figura seguente descrive l'attività "Progetta una classe" del workflow Progetto. Questa attività ha in ingresso le classi di analisi, le realizzazioni dei casi d'uso, le classi di progetto e le interfacce appena abbozzate nel progetto architeturale.



**ATTENZIONE:** nel grafico la classe di progetto appena abbozzata e quella completa sono in realtà la stessa classe a due stadi diversi di sviluppo. La classe di progetto completa è una

classe sufficientemente dettagliata da servire come una buona base per creare codice sorgente.

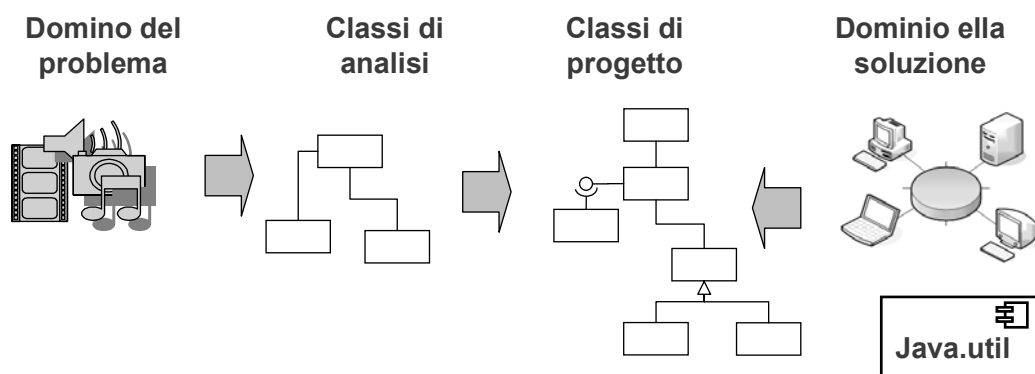
Se lo strumento utilizzato per sviluppare il sistema permette di generare il codice direttamente dal modello, le classi di progetto dovranno essere sviluppate in grande dettaglio; altrimenti potranno essere sviluppate in un dettaglio tale da renderle comprensibili ai programmatori.

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.

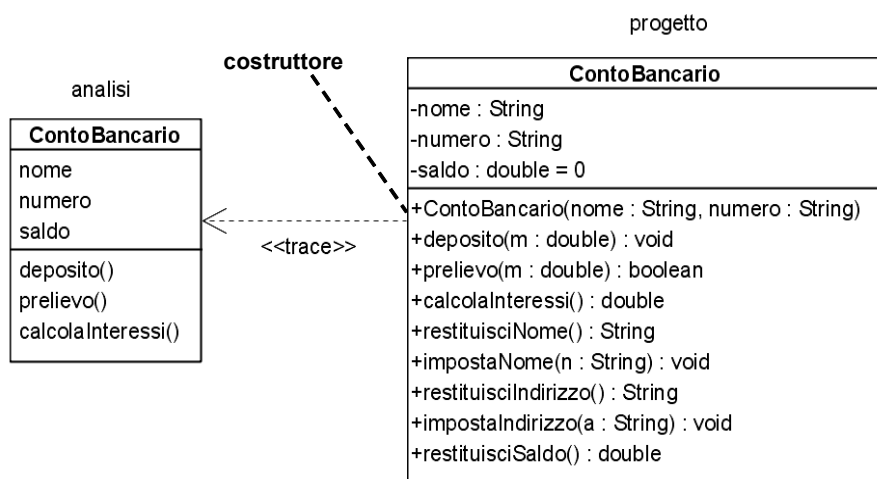




Una classe di analisi può diventare più classi di progetto perché andando a definire in dettaglio le operazioni e gli attributi la classe potrebbe crescere troppo, rendendo quindi necessario suddividerla in classi più piccole. Evita lo “Swiss Army Knife”.

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()`.

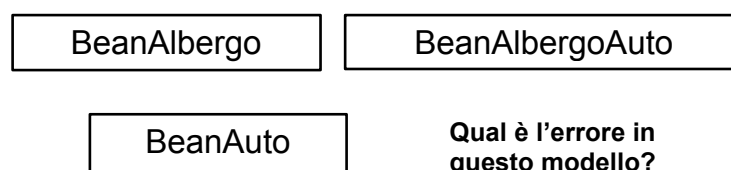
Le classi dovrebbero implementare l'insieme più semplice e più piccolo possibile di operazioni. A volte, per motivi di efficienza e prestazioni questa caratteristica

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***

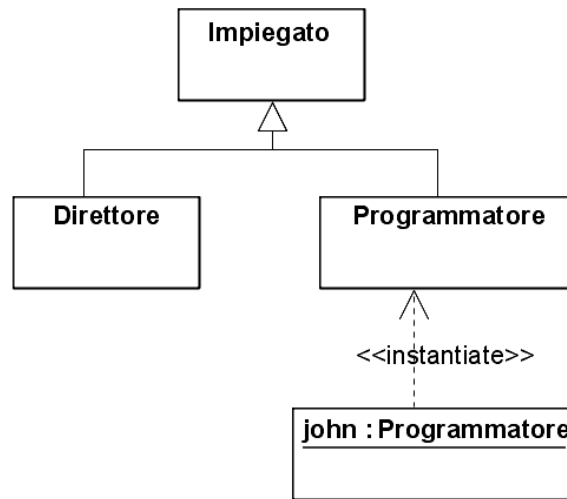
Nel workflow Analisi viene introdotta l'ereditarietà solo se c'è una relazione “is a” chiara e non ambigua. Nel workflow Progetto l'ereditarietà può anche essere usata per riusare codice. Nel seguito, l'ereditarietà verrà esaminata in dettaglio.

### ***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.

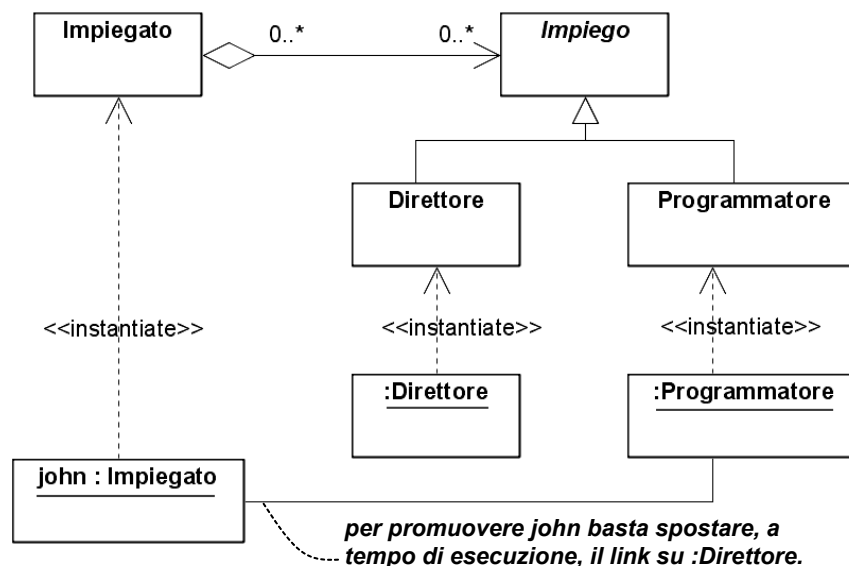
Consideriamo l'esempio seguente. Che problema ha?



Cosa succede se John da programmatore viene promosso a Direttore?

Nel modello c'è un errore semantico fondamentale. Un impiegato è il suo ruolo o piuttosto un impiegato *ha* un ruolo?

La soluzione semanticamente corretta è presentata nel modello seguente.



**ATTENZIONE:** le sottoclassi dovrebbero essere legate alla classe base da una relazione “è una specie di” piuttosto che “è un ruolo interpretato da”. È evidente che un lavoro è un ruolo interpretato da un impiegato e non una specie di impiegato. Ci sono, comunque, molte specie di lavori in un'azienda.

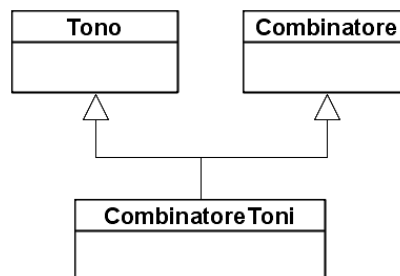
## *Ereditarietà multipla*

Molti linguaggi object-oriented non supportano l'ereditarietà multipla, che può comunque essere sempre realizzata usando l'ereditarietà singola e la delegazione.

Quando si usa l'ereditarietà multipla si deve fare attenzione che:

- Tutte le classi base siano semanticamente disgiunte, altrimenti potrebbero manifestarsi interazioni impreviste. Tutte le classi base devono quindi essere ortogonali.
- La relazione “è una specie di” ed il principio di sostituibilità devono essere validi tra la sottoclasse e tutte le classi base.
- Le classi base non dovrebbero avere una classe base in comune. Se questo accade, si ha un ciclo nella gerarchia e le stesse caratteristiche possono essere ereditate attraverso percorsi multipli.

Un idiomma comune per usare l'ereditarietà multipla è la classe “mixin”, cioè una classe che è progettata per essere una combinazione di altre classi.



## *Ereditarietà e realizzazione di un'interfaccia*

L'ereditarietà fornisce:

- un'interfaccia attraverso le operazioni pubbliche delle classi base;
- un'implementazione attraverso gli attributi, le relazioni e le operazioni protette e pubbliche.

La realizzazione di un'interfaccia consiste nel fornire solamente un insieme di operazioni, attributi e relazioni pubbliche senza specificare alcuna implementazione.

L'ereditarietà è utile se oltre all'interfaccia si vuole riusare anche l'implementazione. Per anni, l'ereditarietà è stata considerata il meccanismo principale di riuso. Oggi si ritiene che ponga vincoli troppo stringenti.

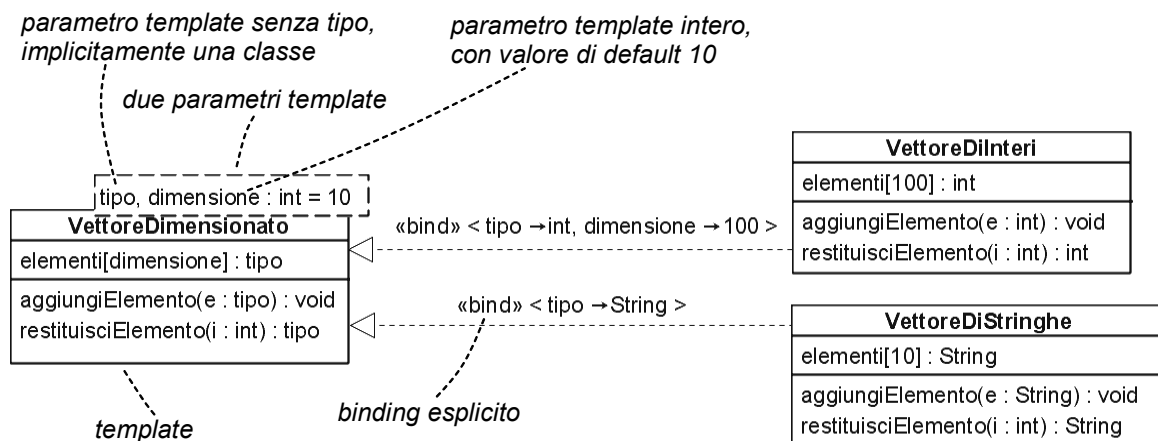
La realizzazione dell'interfaccia è utile quando si vuole definire un contratto senza ereditare i dettagli dell'implementazione. In questo modo, non si ha riuso di codice, ma si fornisce un meccanismo per definire contratti ed assicurare che le classi che si vanno ad implementare siano conformi a quei contratti.

## Template

I template permettono di parametrizzare un tipo. Consideriamo l'esempio seguente.

VettoreDimensionatoDiInteri	VettoreDimensionatoDiReali	VettoreDimensionatoDiStringhe
dimensione : int elementi : int[]	dimensione : int elementi : double[]	dimensione : int elementi : String[]
aggiungiElemento(e : int) : void restituisceElemento(i : int) : int	aggiungiElemento(e : double) : void restituisceElemento(i : int) : double	aggiungiElemento(e : String) restituisceElemento(i : int) : String

Invece di definire tre classi separate che differiscono solo per il tipo degli elementi dell'array, si utilizza un template.



Istanziando il template, cioè sostituendo valori specifici ai parametri formali, si possono creare nuove classi. L'istanziamento avviene utilizzando la relazione stereotipata **«bind»**. Il simbolo  $\rightarrow$  si può leggere come “rimpiazzato da”.

**ATTENZIONE:** i nomi dei parametri di un template sono locali al particolare template.

## 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

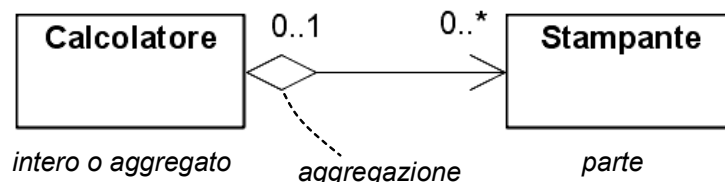
**Aggregazione:** è un tipo di relazione lieve tra oggetti – per esempio, un computer e le sue periferiche. Una periferica può essere condivisa tra computer e non è di proprietà di nessun particolare computer.

**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*

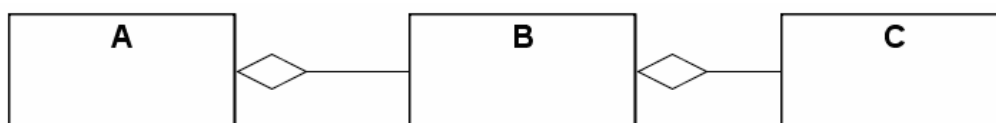
L'aggregazione è un tipo di relazione intero-parte in cui l'intero è fatto di molte parti. In questo tipo di relazioni un oggetto (l'intero) usa i servizi di un altro oggetto (la parte).

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



- 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.

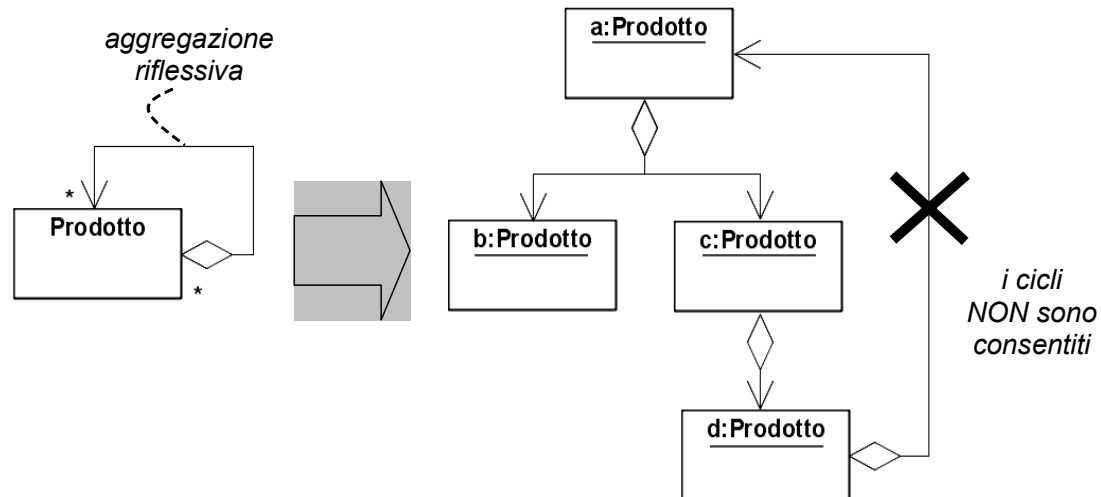
**L'aggregazione è transitiva.**



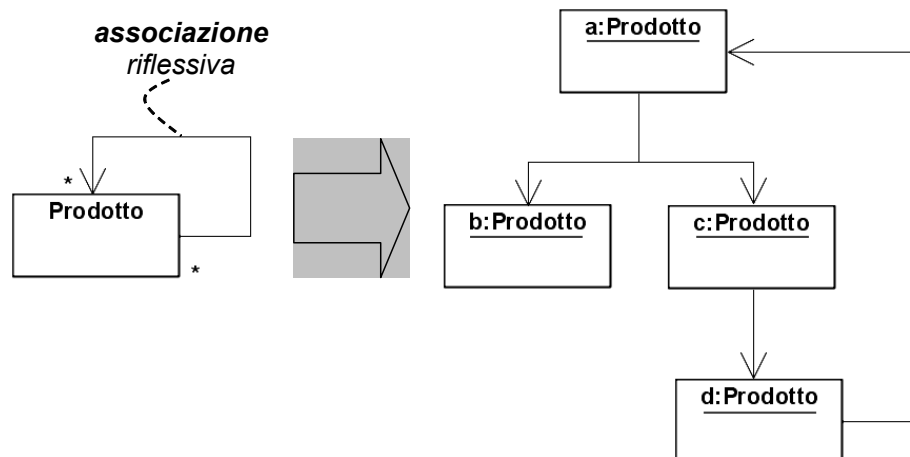
*transitività dell'aggregazione:  
se C è parte di B e B è parte di A, allora C è parte di A*

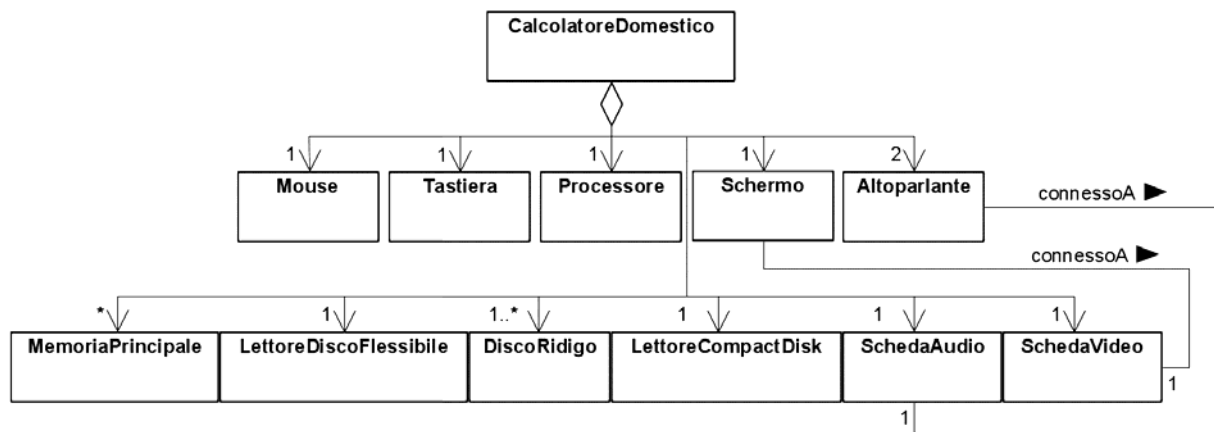


**L'aggregazione è asimmetrica: un oggetto non può mai o indirettamente o direttamente essere parte di sé stesso.**



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

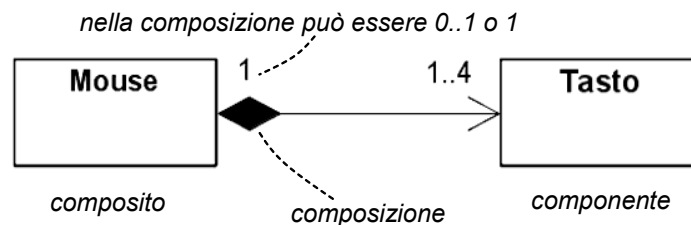




## Semantica della composizione

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.

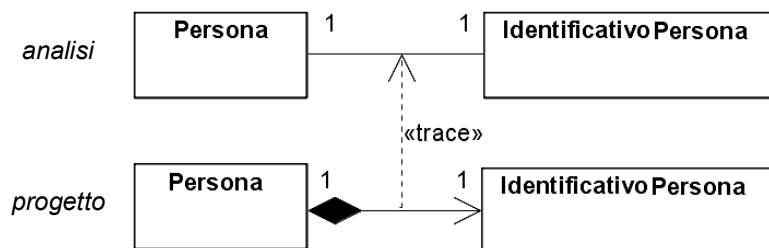
ATTENZIONE: nel decidere se utilizzare un attributo o la composizione tieni sempre in mente la chiarezza, l'utilità e la leggibilità del modello.

### *Come rifinire le relazioni di analisi.*

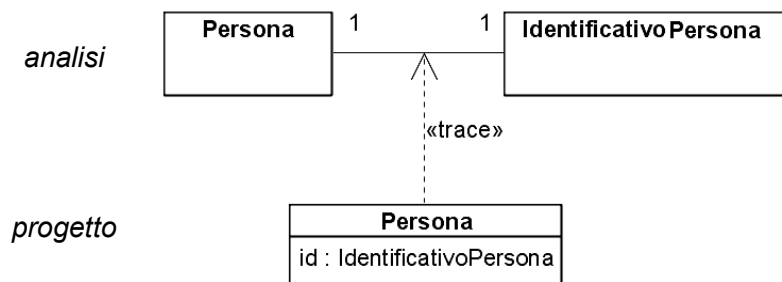
In analisi, le relazioni sono semplicemente relazioni di associazione. Tranne nel caso in cui si viene a creare un ciclo nel grafo di aggregazione, le relazioni di associazione sono trasformate in relazioni di aggregazione o composizione. Dopo aver deciso in quale relazione trasformare una relazione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

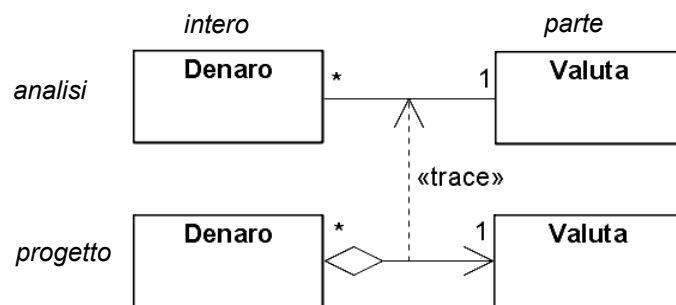
### *Associazioni uno-a-uno*



Se **IdentificativoPersona** non è una classe importante per l'applicazione, allora trasformala in un attributo.



## Associazioni multi-a-uno



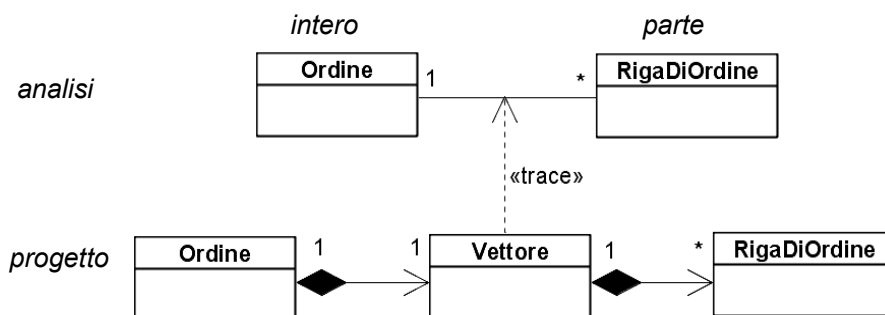
ATTENZIONE: controlla che non ci siano cicli nel grafo di aggregazione.

## 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.



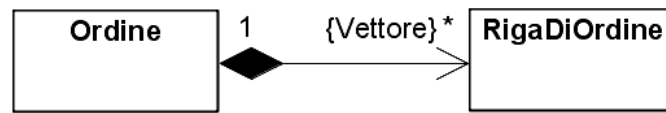
Ci sono quattro strategie fondamentali per modellare le collezioni:

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

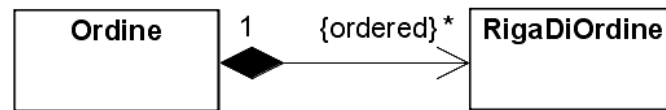
Vantaggio: molto esplicito;

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.



3. Specificare la semantica della collezione aggiungendo una proprietà alla relazione, ma non specificare nessuna classe di implementazione.



La tabella seguente mostra proprietà standard che possono essere applicate alle molteplicità per indicare la semantica richiesta.

Proprietà	Semantica
{ordered}	Elementi ordinati
{unordered}	Elementi non ordinati
{unique}	Gli elementi sono unici
{nonunique}	Gli elementi nella collezione possono essere duplicati

Le varie combinazioni delle proprietà sull'ordinamento e sull'unicità danno vita all'insieme di collezioni seguente

Proprietà	Collezioni in OCL
{unordered,nonunique}	Bag
{unordered, unique}	Set
{ordered,unique}	OrderedSet
{ordered,nonunique}	Sequence

4. Non preoccuparsi di rifinire le associazioni uno-a-molti in classi collezione, ma delegare tutto al programmatore.

## Mappa

La mappa è un tipo di classe collezione molto utile: è molto simile ad una tabella di una base di dati con due colonne (chiave e valore).

Le mappe mantengono un insieme di nodi, dove ogni nodo punta a due oggetti: l'oggetto valore e l'oggetto chiave. Le mappe sono ottimizzate per trovare velocemente un oggetto valore quando viene fornito uno specifico oggetto chiave.

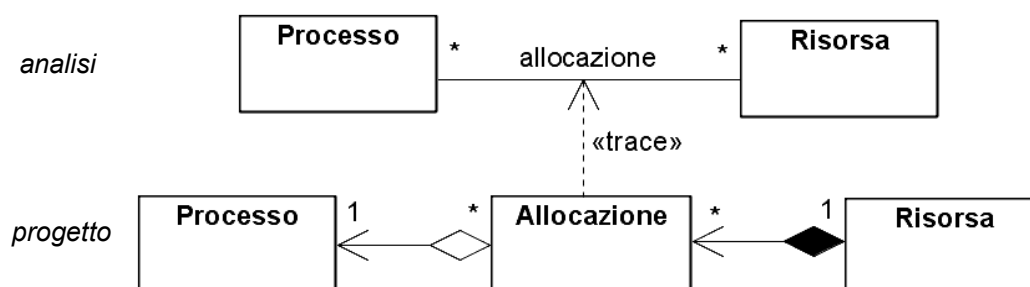
UML non ha proprietà standard per indicare una mappa. Per indicare una mappa sul modello di progetto, si può usare il tipo di collezione {HashMap} o il valore etichettato {map keyName}.

## Rendere concrete le relazioni (reification)

Alcune relazioni individuate nel workflow Analisi non sono supportate da nessun linguaggio object-oriented. Queste relazioni (associazioni multi-a-molti, associazione bidirezionali, classi di associazione) devono quindi essere rese concrete (*reification*).

### Associazioni multi-a-molti

Le associazioni multi-a-molti devono essere rese concrete trasformandole in classi, in aggregazioni, in composizioni o in dipendenze.

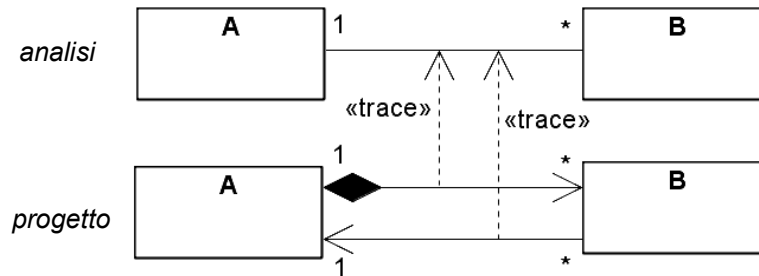


NOTA BENE: Nel diagramma precedente, è stato scelto di riservare il ruolo di intero alla classe **Risorsa** perché il sistema è centrato sulle risorse. Se avessimo avuto un sistema centrato sui processi, avremmo modellato la classe **Processo** come intero.

Se volessimo rappresentare entrambi i punti di vista, allora dovremmo introdurre un nuovo oggetto (ManagerAllocazione) che mantiene una lista di oggetti Allocazione dove ogni oggetto punta a sia una Risorsa che un oggetto Processo.

### ***Associazioni bidirezionali***

Servono per modellare la situazione in cui un oggetto a di classe A usa i servizi di un oggetto b di classe B e l'oggetto b usa i servizi di a per completare i suoi. L'associazione bidirezionale deve essere trasformata in due associazioni o dipendenze unidirezionali.



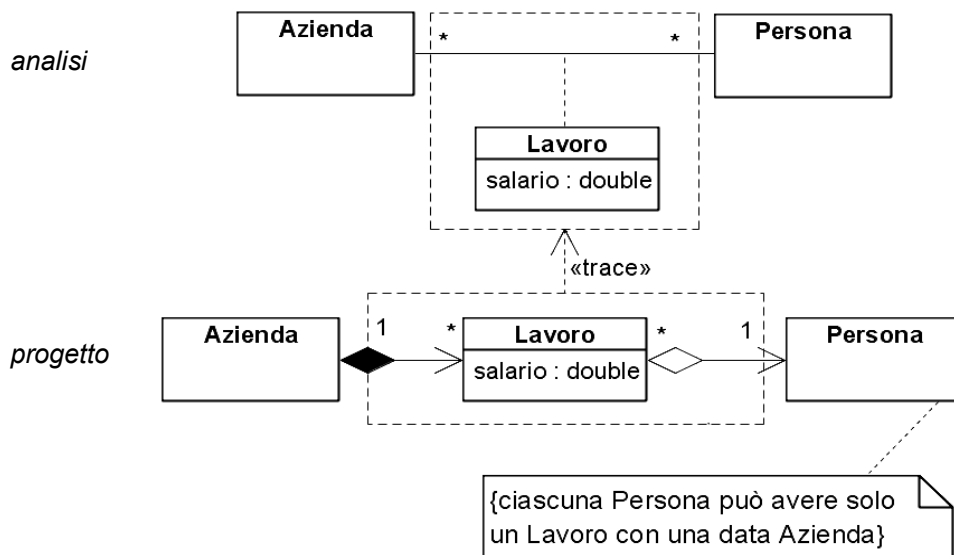
**ATTENZIONE:** ricorda sempre il vincolo di asimmetria per l'aggregazione e la composizione. Questo significa che se la classe A ha una relazione di aggregazione o composizione con la classe B, l'associazione da B ad A non deve essere rifinita.

**NOTA BENE:** le associazioni bidirezionali esistono anche quando un intero passa come parametro a qualche operazione di una sua parte un riferimento a se stesso oppure quando una parte istanzia l'intero in una delle sue operazioni. In questo caso, usa una relazione di DIPENDENZA piuttosto che un'associazione.

### ***Classi di associazione***

Le classi di associazione sono artefatti di analisi e non sono usate nei linguaggi orientati agli oggetti.

Trasforma una classe di associazione in una classe normale e usa una combinazione di associazione, aggregazione, composizione o dipendenza per catturare la semantica della classe di associazione. LA TRASFORMAZIONE PUÒ INSERIRE VINCOLI NEL MODELLO.



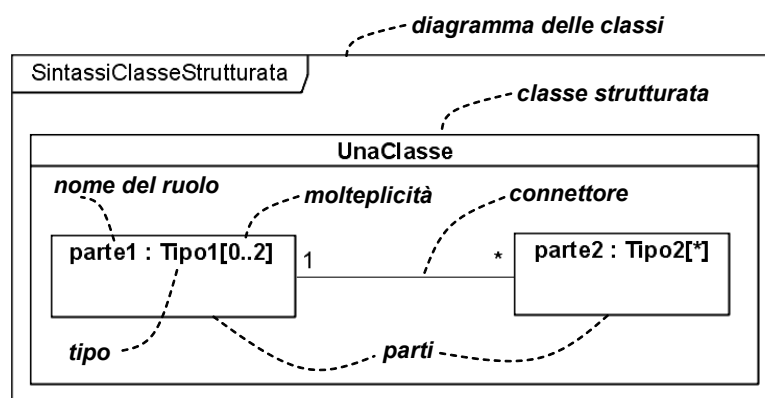
Nella figura la semantica della classe di associazione (gli oggetti agli estremi della classe di associazione devono formare un'unica coppia) viene recuperata aggiungendo la nota con l'appropriato vincolo.

## Esplorare la composizione con le classi strutturate

UML 2 permette di esplorare la relazione tra un classificatore composito e le sue parti.

Un **classificatore strutturato** è semplicemente un classificatore che ha una struttura interna. Questa struttura è modellata da connettori che collegano le parti del classificatore. Una parte è un ruolo che una o più istanze possono interpretare nel contesto del classificatore strutturato.

L'interazione tra il classificatore strutturato e l'ambiente esterno è modellato dalle sue interfacce e dalle sue porte.





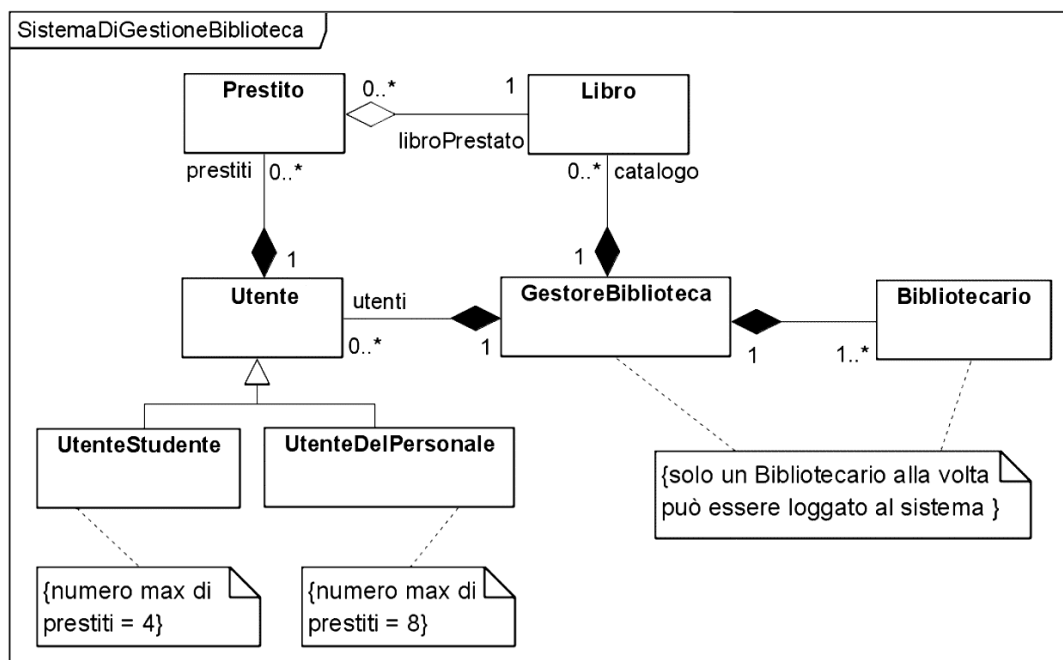
Un connettore è una relazione tra le parti nel contesto di un classificatore strutturato. Queste relazioni potrebbero individuare associazioni tra le classi delle parti oppure potrebbero semplicemente essere relazioni ad hoc in cui le parti sono collegate nel classificatore strutturato in una collaborazione temporanea per realizzare qualche attività.

Quindi:

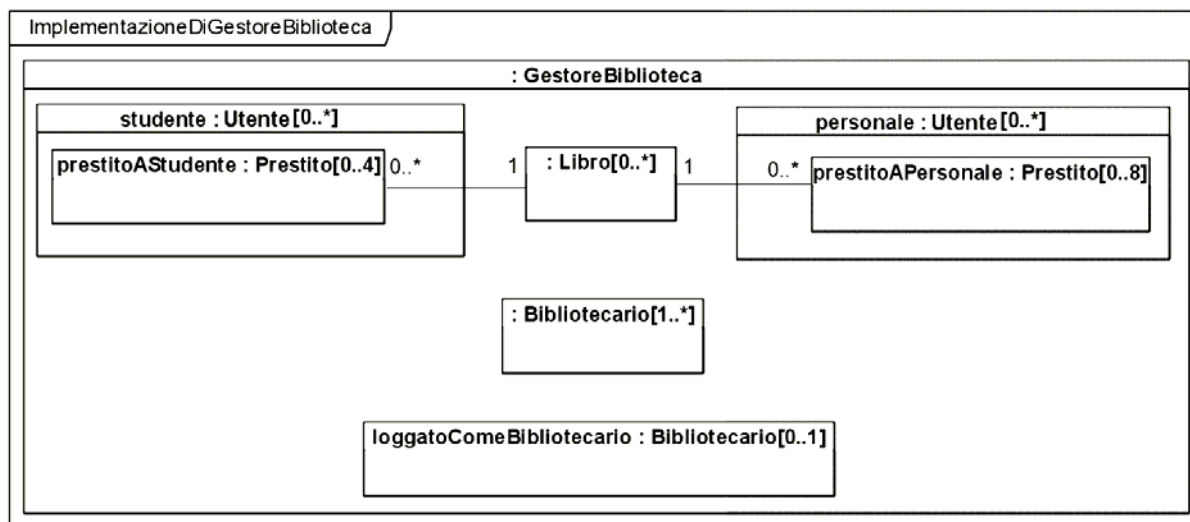
1. le parti collaborano nel contesto del classificatore strutturato;
2. le parti rappresentano ruoli che le istanze di un classificatore possono interpretare nel contesto del classificatore strutturato – le parti non rappresentano classi;
3. il connettore è una relazione tra due parti ed indica che le parti possono comunicare tra di loro.

Una **classe strutturata** ha l'ulteriore vincolo, rispetto al classificatore, che tutte le parti, i connettori e le porte le appartengono.

La figura seguente mostra il diagramma delle classi per un semplice sistema di gestione delle biblioteche.



Come si vede dal diagramma, la classe **GestoreBiblioteca** ha una struttura interna e quindi può essere modellata come una classe strutturata.

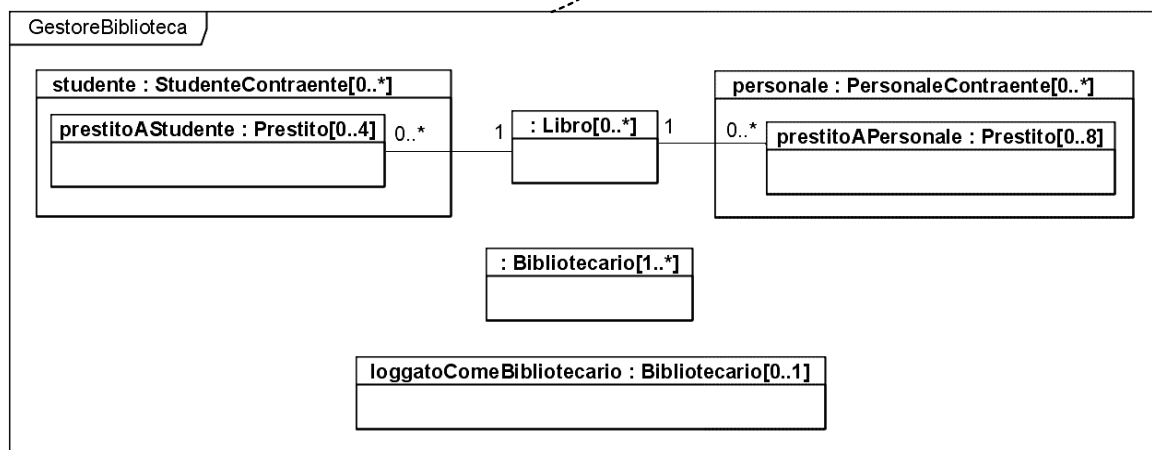


Il modello che utilizza la classe strutturata permette di andare più in dettaglio, andando a rappresentare i ruoli che le istanze della classe interpretano nell'implementazione della classe. Per esempio, i libri possono essere presi in prestito da due tipi diversi di utenti: gli studenti ed il personale. Studenti e personale possono prendere in prestito un diverso numero massimo di libri.

NOTA BENE: i ruoli interpretati dalle parti nella classe strutturata possono essere diversi da quelli interpretati dalle classi nelle associazioni. Per esempio, il ruolo generico di utenti è stato rimpiazzato da studente e personale.

Per rappresentare i classificatori strutturati esiste un diagramma apposito: il diagramma di struttura composita (composite structure diagram). Il nome del diagramma è il nome del classificatore strutturato ed i contenuti del diagramma sono proprio i contenuti del classificatore strutturato.

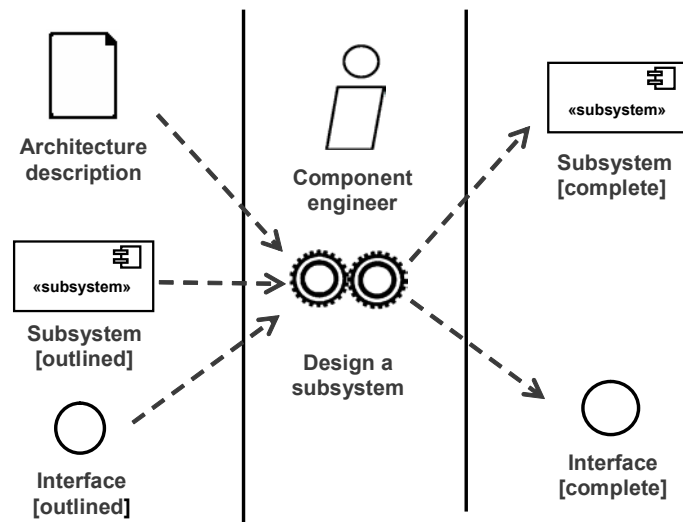
*diagramma di struttura composita*



## Interfacce e componenti

L'attività dello UP "Progetta un sottosistema" è presentata nella figura seguente. Questa attività scompone il sistema in parti che sono il più indipendenti possibile. Questo comporta un'attenta progettazione delle interfacce e l'assicurazione che il sottosistema realizzi il comportamento specificato dalle interfacce.

I sottosistemi sono tipi di componenti.



Un'**interfaccia** specifica un insieme di caratteristiche pubbliche. L'interfaccia serve a separare la specifica delle funzionalità dalla loro implementazione.

Un'interfaccia non può essere istanziata: semplicemente definisce un contratto che può essere realizzato da classi o altri classificatori.

Le interfacce possono specificare le caratteristiche elencate nella tabella seguente (la tabella presenta anche le responsabilità dei classificatori rispetto all'interfaccia).

Interfaccia specifica	Classificatore
Operazione	Deve avere un'operazione con la stessa intestazione e semantica
Attributo	Deve avere operazioni pubbliche per leggere o cambiare il valore dell'attributo – al classificatore non è richiesto di possedere l'attributo specificato dall'interfaccia, ma deve comportarsi come se lo avesse
Associazione	Deve avere un'associazione al classificatore target – se un'interfaccia specifica un'associazione ad un'altra interfaccia, i classificatori di queste interfacce devono avere un'associazione tra di loro
Vincolo	Deve supportare il vincolo

Stereotipo	Deve avere lo stereotipo
Valore etichettato	Deve avere il valore etichettato
Protocollo	Deve realizzare il protocollo

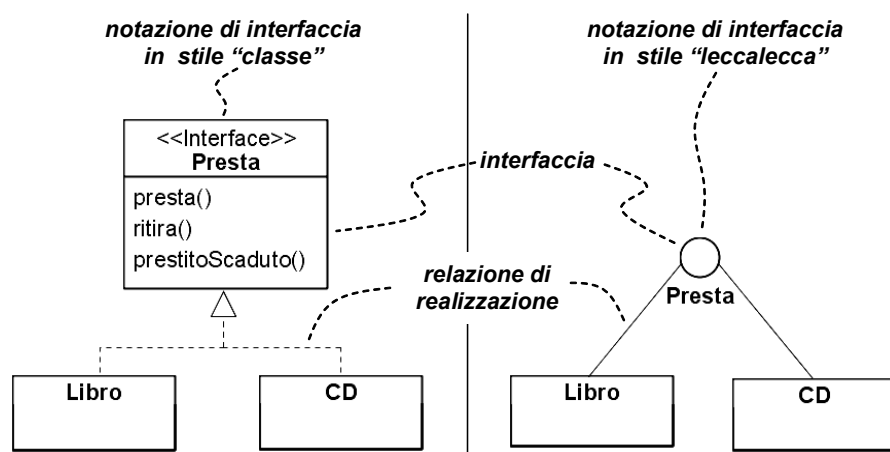
NOTA BENE: un'interfaccia definisce una specifica per le caratteristiche e non implica mai una particolare implementazione.

Progettare interfacce piuttosto che connessioni dirette tra classi consente di svincolarsi dalla particolare implementazione. La connessione è verso un'interfaccia e questa interfaccia può essere realizzata da qualsiasi classe o classificatore. **Architetture orientate ai servizi.**

Dal punto di vista dei linguaggi di programmazione, Java fornisce esplicitamente le interfacce. In C++ lo stesso risultato si può ottenere attraverso una classe astratta.

**Interfacce fornite** (provided interfaces): insieme di interfacce realizzate da un classificatore.

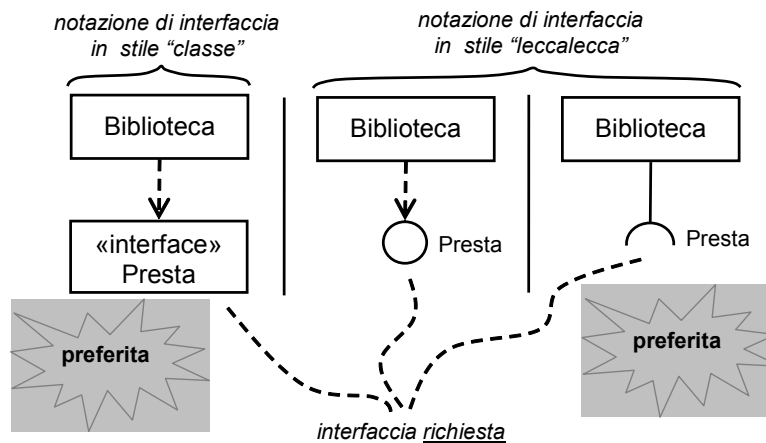
**Interfacce richieste** (required interfaces): insieme di interfacce richieste da un classificatore per realizzare le sue operazioni.



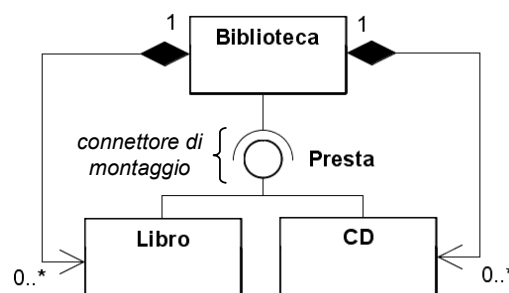
Nell'esempio, l'interfaccia Presta permette di trattare libri e CD allo stesso modo da un punto di vista del prestito.

Le interfacce sono nominate in UpperCamelCase. In C# e Visual Basic viene usata la convenzione di prefissare la lettera I al nome delle interfacce. La relazione di realizzazione è la relazione tra l'interfaccia e le classi UML che realizzano l'interfaccia.

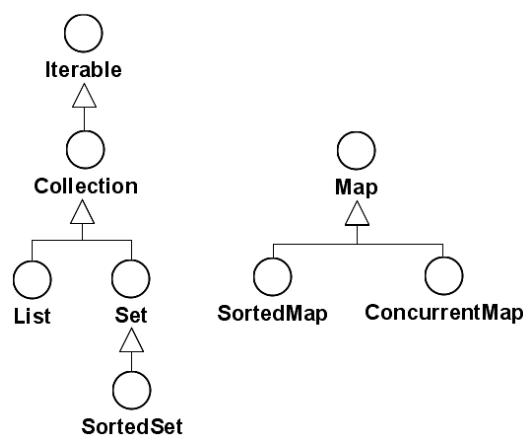
ATTENZIONE: le interfacce possono essere realizzate non solo dalle classi, ma anche da altri classificatori come package e componenti.



Nella figura, la classe **Biblioteca** conosce e richiede lo specifico protocollo definito dall'interfaccia **Prestito**.



La figura seguente mostra un esempio di interfacce con implementazioni multiple per le classi collezione della libreria standard di Java.

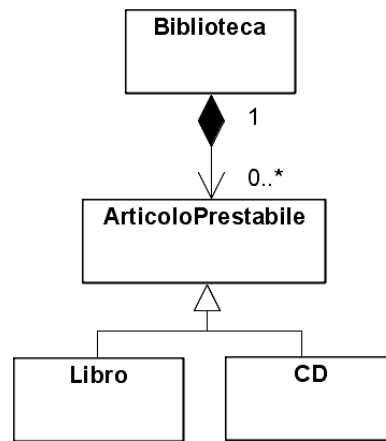


## Realizzazioni di Interfacce ed Ereditarietà

Semantica della realizzazione di un'interfaccia: "realizza il contratto specificato da".

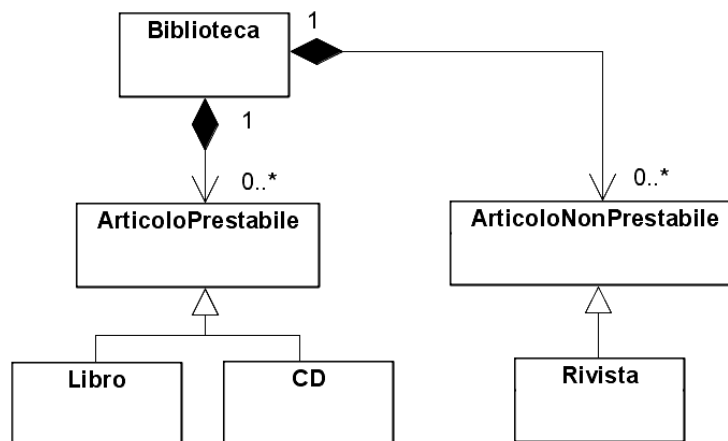
Semantica dell'ereditarietà: "è un".

Osserviamo l'esempio seguente. Cosa c'è di sbagliato?



Nella figura, i libri e i CD sono di tipo ArticoloPrestabile. Questa capacità di essere prestabili è effettivamente sufficiente per definire libri e CD? Ovviamente no: l'essere prestabili è un ruolo che i libri ed i CD interpretano nell'ambito del contesto biblioteca.

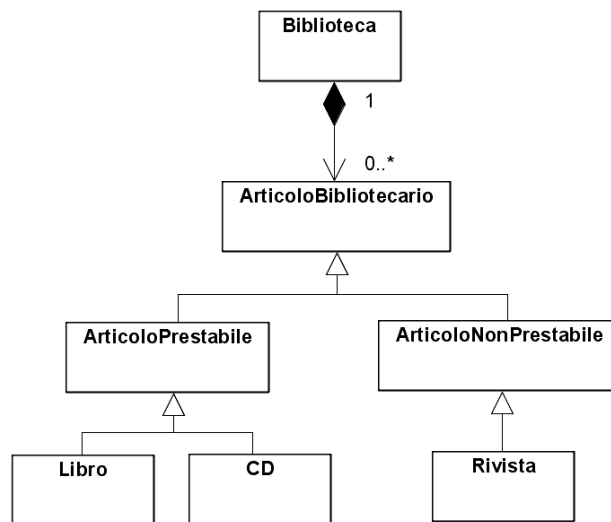
Immaginiamo di dover aggiungere le riviste al nostro sistema e supponiamo che le riviste non siano prestabili.



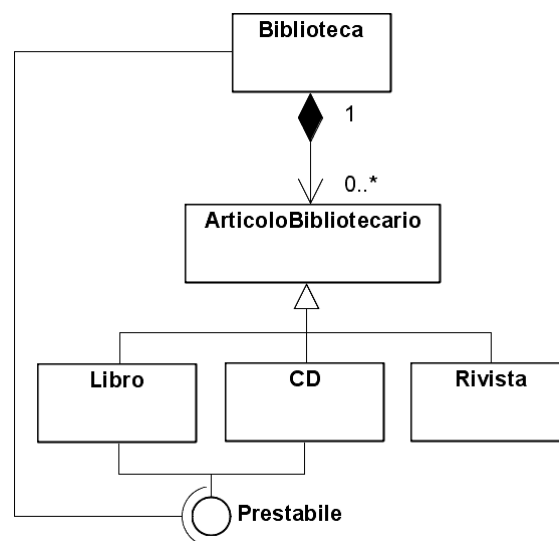
La biblioteca deve ora gestire due liste di articoli: quelli prestabili e quelli non prestabili.

ATTENZIONE: stiamo confondendo la memorizzazione con il prestito.

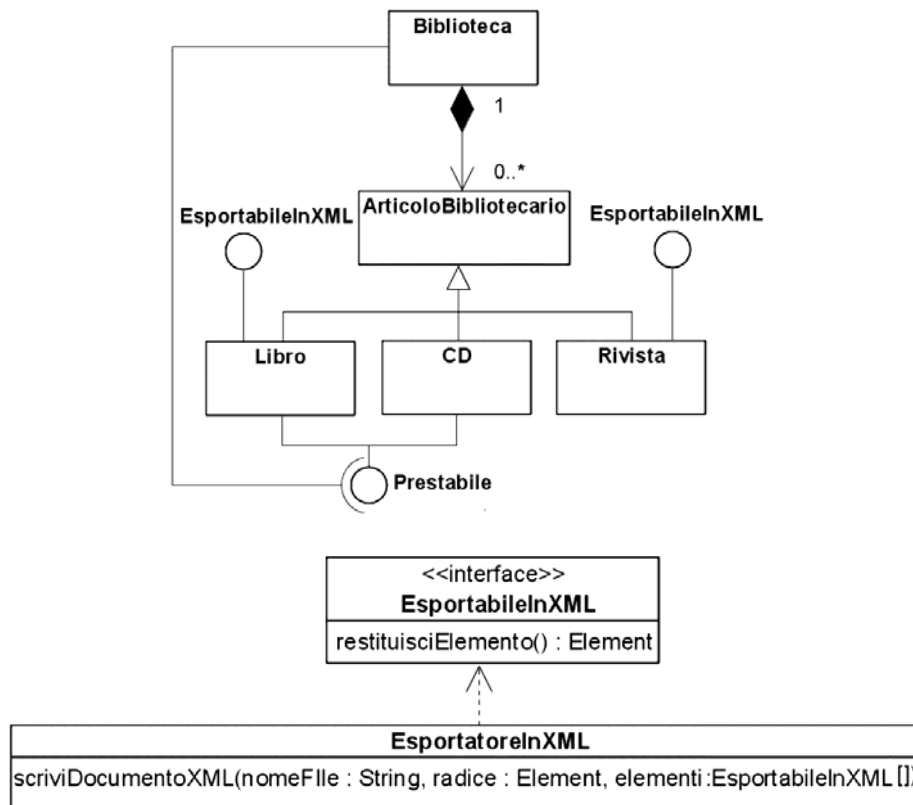
Una possibile soluzione è di inserire il protocollo “prestabile” in un livello più profondo della gerarchia:



Il modello seguente è sicuramente più elegante. Meno classi, meno relazioni di composizione e due livelli soli di ereditarietà.



Supponiamo di dover esportare alcuni dati relativi a libri e riviste, ma non a CD. Introduciamo una classe `EsportatoreInXML` ed un'interfaccia `EsportabileInXML` che definisce il protocollo che ogni oggetto esportabile deve avere per poter funzionare con l'`EsportatoreInXML`.



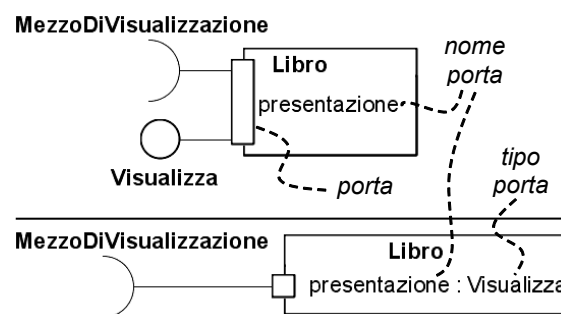
NOTA BENE: la soluzione presentata nella figura consente di separare la memorizzazione (relazioni di composizione) dal prestito e dall'esportabilità.

Sistemi Informativi

317

## Porte

Una porta raggruppa un insieme semanticamente coesivo di interfacce fornite e richieste. Una porta indica un punto specifico di interazione tra un classificatore ed il suo ambiente.



La notazione alternativa usata nella parte in basso della figura è applicabile solo se la porta ha un tipo singolo di interfaccia fornita.

Per connettere due porte, le interfacce fornite e richieste devono combaciare.

**VISIBILITÀ:** una porta che si sovrappone al bordo del classificatore è pubblica. Se la porta è dentro il confine del classificatore, la porta ha o visibilità protetta o privata (la visibilità effettiva è memorizzata nella specifica della porta).



**MOLTEPLICITÀ:** le porte possono avere una molteplicità il cui valore viene dato tra parentesi quadre dopo il nome della porta ed il nome del tipo (per esempio, presentazione: Visualizza[1]). Questa molteplicità indica il numero di istanze della porta che avrà ogni istanza del classificatore.

## Componenti

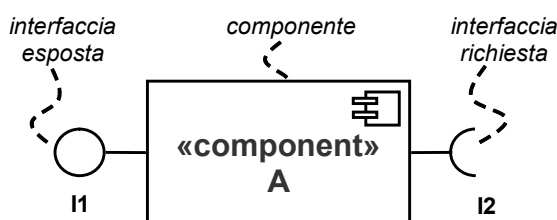
Specifica di UML 2.0: “Un componente rappresenta una parte modulare di un sistema che incapsula il suo contenuto e il cui operato è rimpiazzabile all’interno del suo ambiente”.

Componente come scatola nera il cui comportamento è completamente definito dalle sue interfacce sia fornite che richieste.

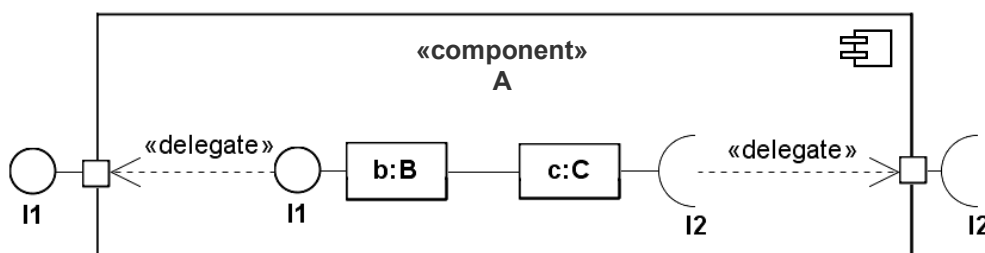
I componenti possono avere attributi ed operazioni e possono partecipare nelle relazioni di associazione e generalizzazione. Componenti sono classificatori strutturati e possono avere una struttura interna che comprende parti e connettori.

Un componente può rappresentare semplicemente un costrutto logico, per esempio un sottosistema, istanziato indirettamente quando sono istanziate le sue parti, oppure può essere un costrutto istanziato a tempo di esecuzione (Enterprise JavaBean).

Il *diagramma dei componenti* può rappresentare componenti, dipendenze tra componenti ed il modo in cui i classificatori sono assegnati ai componenti.

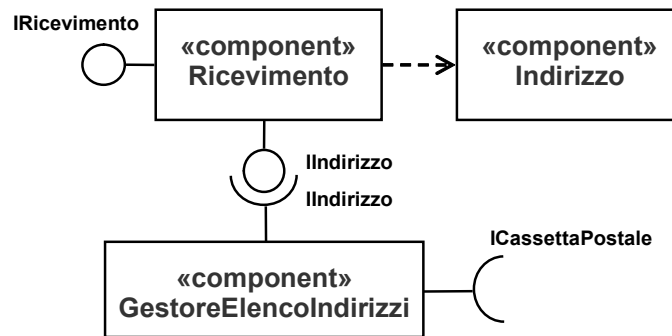


Un componente può avere una struttura interna. Le parti possono essere annidate dentro il componente o presentate esternamente attraverso relazioni di dipendenza.



Quando un componente ha struttura interna delegherà le responsabilità ad una o più delle sue parti interne.

I componenti possono dipendere da altri componenti. Per disaccoppiare i componenti, inserisci sempre un'interfaccia come intermediario per la dipendenza.



Il componente Ricevimento serve a disaccoppiare il componente GestoreElencoIndirizzi da Indirizzo (agisce come un Facade).



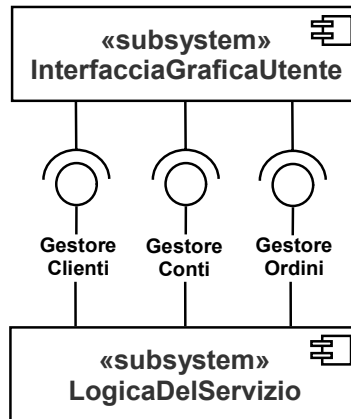
UML 2 fornisce un piccolo insieme di stereotipi standard per i componenti.

Stereotipo	Semantica
« <b>buildComponent</b> »	Un componente che definisce un insieme di entità per scopi organizzativi o di sviluppo del sistema
« <b>entity</b> »	Un componente di informazione persistente che rappresenta un concetto del business
« <b>specification</b> »	Un classificatore che specifica un dominio di oggetti senza definire l'implementazione fisica degli oggetti – per esempio, un componente che ha solo interfacce fornite e richieste, ma non classificatori che le realizzino
« <b>implementation</b> »	Un componente che non ha specifica – è un'implementazione per un componente stereotipato « <b>specification</b> » da cui dipende
« <b>process</b> »	Un componente basato sulle transazioni
« <b>service</b> »	Un componente funzionale senza stato che calcola un valore
« <b>subsystem</b> »	Un'unità di decomposizione gerarchica per sistemi complessi.

## Sottosistemi

Un sottosistema è un componente che agisce come un'unità di decomposizione per un sistema più grande.

ATTENZIONE: i sottosistemi sono costrutti logici che non possono essere istanziati a tempo di esecuzione.



Il sottosistema InterfacciaGraficaUtente conosce solo le interfacce e non gli aspetti interni della logica del servizio. L'uso dei sottosistemi e delle interfacce disaccoppia i sottosistemi e crea una certa flessibilità architetturale.

## Come trovare le interfacce

Durante il workflow Progetto, il modello di progetto dovrebbe essere esaminato attentamente per trovare le interfacce. In particolare,

- Esamina ogni associazione, chiedendoti “Dovrebbe questa associazione effettivamente essere associata ad una classe particolare o dovrebbe essere più flessibile?”
- Esamina ogni messaggio, chiedendoti “Dovrebbe questo messaggio essere inviato ad oggetti di solo una classe o dovrebbe essere più flessibile?”
- Raggruppa operazioni che potrebbero essere riusabili altrove;
- Raggruppa insieme di attributi che si ripetono in più di una classe;
- Esamina classi che svolgono lo stesso ruolo nel sistema – il ruolo può indicare un'interfaccia;
- Analizza la possibilità di espansioni future – Se si prevede che nel futuro altre classi verranno aggiunte al sistema, tenta di definire una o più interfacce il cui compito è quello di definire il protocollo per definire queste nuove classi.

- Esamina la dipendenza tra componenti – utilizza connettori di assemblaggio dove possibile

## Progettare con le interfacce

Usando le interfacce , si possono stabilire protocolli comuni che potrebbero essere realizzati da molte classi e molti componenti.

Per esempio, in un sistema che modella un'organizzazione, ci sono molte classi che hanno un nome ed un indirizzo. Tutte queste classi possono interpretare il ruolo comune di `UnitaIndirizzabile`. Sembra naturale definire un'unica interfaccia `NomeEdIndirizzo`, che tutte le classi potrebbero realizzare. Una soluzione alternativa, ma meno flessibile sarebbe quella di utilizzare l'ereditarietà.

Le interfacce quindi forniscono la capacità di inserire facilmente nuove classi nel sistema. Se i sistemi sono progettati intorno alle interfacce, allora associazioni e invio di messaggi non sono più associati a specifici oggetti di classi particolari, ma piuttosto ad una particolare interfaccia.

Nascondere i dettagli implementativi di sottosistemi complessi dietro un'interfaccia semplice e ben definita è conosciuto come il pattern di progetto (design pattern) *Facade*.

Il pattern *Facade* (Facciata) permette l'occultamento dell'informazione e la separazione dei concetti. Il pattern consente di ridurre la complessità del sistema e controllare e gestire l'accoppiamento tra i sottosistemi.

### *Architettura e livelli*

L'insieme dei sottosistemi e delle interfacce di progetto costituisce l'architettura di alto livello di un sistema.

Per capire e mantenere questa architettura, l'insieme dovrebbe essere organizzato in modo coerente, applicando un pattern architetturale conosciuto come stratificazione ("layering"). Il pattern organizza i sottosistemi e le interfacce in strati, dove i sottosistemi in ogni strato sono semanticamente coesivi.

L'organizzazione a strati permette di gestire l'accoppiamento tra sottosistemi:

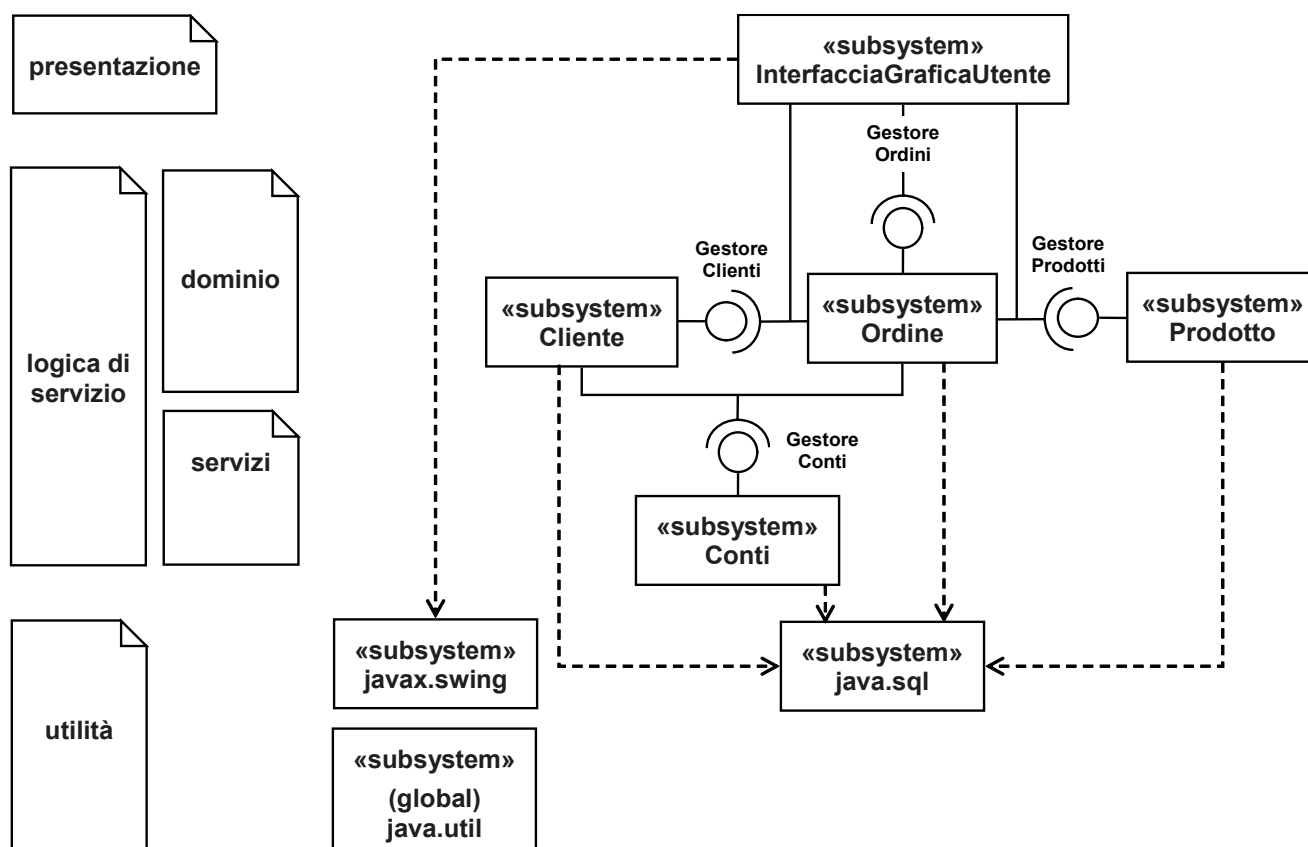
- Introducendo nuove interfacce quando necessario;
- Inserendo classi in nuovi sottosistemi in modo da ridurre l'accoppiamento tra sottosistemi.

Le dipendenze tra strati devono essere gestite molto attentamente. Idealmente, gli strati dovrebbero essere disaccoppiati il più possibile, in modo da assicurare che:

- Le dipendenze vadano in un solo verso
- Tutte le dipendenze siano mediate da interfacce.

Tipicamente, gli strati sono suddivisi in presentazione, logica di business e utilità. La logica di business è tipicamente separata in due strati: lo strato *dominio* che contiene i sottosistemi specifici all'applicazione e lo strato *servizi* che contiene i sottosistemi che possono essere riutilizzati in altre applicazioni.

Nella figura seguente, i sottosistemi Java non sono connessi attraverso un'interfaccia, sebbene tali sottosistemi le rendano disponibili. La ragione è che non sembra molto utile presentare queste interfacce nel modello. Nota anche che il sottosistema `java.util` è globale e quindi i contenuti pubblici del sottosistema sono visibili ovunque. Di nuovo, le dipendenze a questo sottosistema non sono presentate perché poco informative.



## *Vantaggi e Svantaggi nell'uso delle interfacce*

Usare le interfacce nel progetto fornisce una serie di **vantaggi**:

- Protocolli comuni che possono essere implementati da molte classi o componenti. Per esempio, un'interfaccia NomeEIndirizzo per un sistema di gestione delle risorse umane.
- Flessibilità
- Estendibilità – associazioni e messaggi non sono legati ad oggetti di una classe particolare, ma piuttosto ad una particolare interfaccia. Diventa così più facile aggiungere nuove classi ad un sistema seguendo i protocolli stabiliti dalle interfacce.
- Riduzione del numero di dipendenze tra classi, sottosistemi e componenti e quindi maggior controllo sulla quantità di accoppiamento in un modello.

Ci sono comunque anche degli **svantaggi**:

- Maggiore flessibilità significa anche più complessità. In teoria, ogni operazione di ogni classe potrebbe essere un'interfaccia, ma in questo modo il sistema diventerebbe incomprensibile.
- Maggior costo a livello di prestazioni.

**IMPORTANTE:** il progetto di un sistema cerca di catturare un ben definito insieme di semantiche di business. Alcune di queste semantiche sono fluide e cambiano rapidamente, mentre altre sono relativamente stabili.

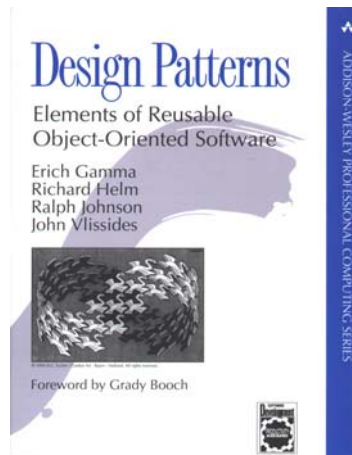
Semantiche fluide -> interfacce

Semantiche stabili -> associazioni e classi

**IMPORTANTE:** tieni in mente la regola KISS (Keep Interfaces Sweet and Simple)

# Design Patterns

I pattern di progetto sono documentati in Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, Addison-Wesley, 1995.



“Ogni pattern descrive un problema specifico che ricorre più volte nel nostro ambiente e poi descrive il nucleo della soluzione a quel problema, in modo da poter utilizzare tale soluzione un milione di volte, senza mai farlo allo stesso modo”.

C. Alexander et al. *A Pattern Language*, 1977

Le soluzioni nell'OO sono espresse in termini di oggetti, classi e relazioni. Per Alexander sono espresse in termini di muri e porte.

I design pattern quindi rappresentano soluzioni *assodate* a problemi *ricorrenti* in contesti *specifici*.

Catturano soluzioni che sono già state adottate e che si sono evolute nel tempo.

Catturano le strutture statiche e dinamiche di soluzioni che ricorrono più volte durante lo sviluppo di applicazioni in un particolare contesto.

Un design pattern:

- Astrae una struttura ricorrente nel progetto;
- Comprende classi e/o oggetti
  - Dipendenze
  - Strutture
  - Interazioni
- Descrive ed assegna nomi ai vari componenti;
- “astrae” l'esperienza nella progettazione.

Un design pattern è caratterizzato da:

- un **nome** – deve descrivere il problema di progetto, le sue soluzioni e conseguenze in una parola o due. Nominare un design pattern consente di avere un vocabolario comune di progetto e poter quindi condividere il pattern con gli altri;
- un **problema** – descrive quando il pattern può essere applicato. Spiega il problema ed il suo contesto. A volte il problema includerà una lista di condizioni che devono essere soddisfatte prima di applicare il pattern;
- la **soluzione** – descrive gli elementi che costituiscono il progetto, le loro relazioni, le loro responsabilità e collaborazioni. **NOTA BENE:** la soluzione non descrive un progetto o un'implementazione concreti, perché un pattern è come un modello che può essere applicato in molte situazioni differenti;
- le **conseguenze** – sono i risultati raggiunti ed i compromessi effettuati applicando il pattern. Le conseguenze includono l'impatto che avrà il pattern sulla flessibilità, estendibilità e portabilità del software.

Gli obiettivi dei design pattern sono:

- Creare un vocabolario comune di progetto all'interno della comunità Object-Oriented;
- Aiutare gli sviluppatori a risolvere problemi già trattati;
- Creare un linguaggio per comunicare intuizioni ed esperienza su problemi e sulle loro soluzioni.

Un buon design pattern:

- Deve risolvere un problema reale;
- Deve essere un concetto provato;
- La soluzione non deve essere ovvia;
- Deve descrivere relazioni, strutture e meccanismi;
- “Se non lo puoi disegnare non è un pattern” (Alexander)
- Deve mostrare che la soluzione descritta è:
  - a. *Utile* (ricorrenza)
  - b. *Utilizzabile* (contesto specifico)
  - c. *Usata* (assodata)



Nel catalogo proposto da Gamma e gli altri, vengono definiti 23 design pattern organizzati secondo due criteri: il proposito ed il campo di azione.

Il proposito può essere *di creazione* (creational), *strutturale* (structural) o *comportamentale* (behavioral).

Il campo d'azione specifica se il pattern viene applicato principalmente alle classi oppure agli oggetti.

I Creational pattern si concentrano sul processo di creazione degli oggetti.

Gli structural pattern trattano la composizione di classi ed oggetti.

I behavioral pattern caratterizzano come le classi o gli oggetti interagiscono e distribuiscono responsabilità.

La tabella seguente riassume l'organizzazione dei Design Pattern.

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

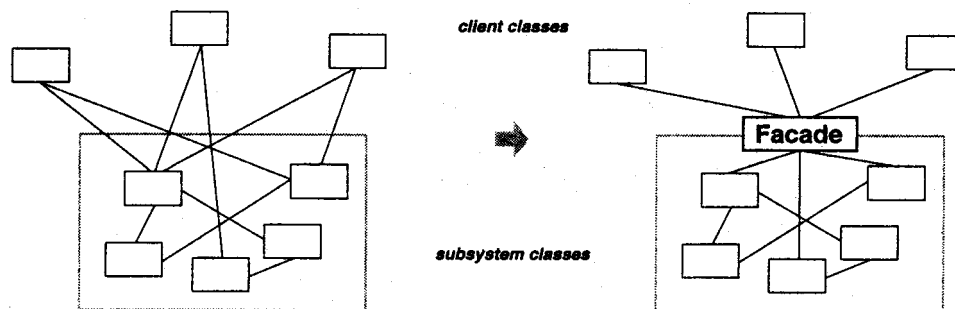
# Design Pattern FACADE

## Intento:

Fornire un'interfaccia unificata ad un insieme di interfacce in un sottosistema. Il design pattern Facade definisce un'interfaccia di più alto livello che rende il sottosistema più facile da usare.

## Motivazione:

Strutturare un sistema in sottosistemi aiuta a ridurre la complessità. Un obiettivo comune è quello di minimizzare la comunicazione e le dipendenze tra sottosistemi. Un modo per raggiungere questo obiettivo è quello di fornire un'unica interfaccia semplificata alle funzioni più generali di un sottosistema.

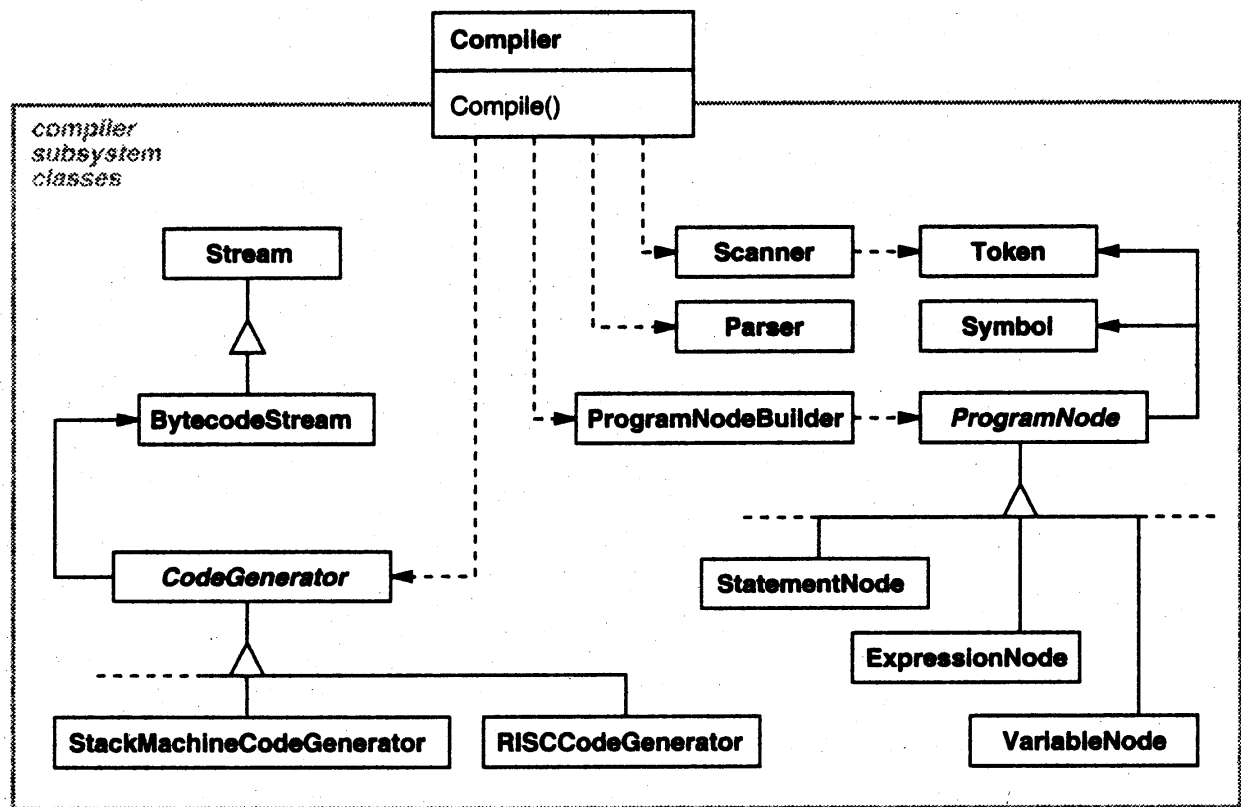


Consideriamo come esempio un ambiente di programmazione che permette ad alcune applicazioni di accedere al sottosistema di compilazione. Questo sottosistema contiene un numero di classi che realizzano il compilatore.

Alcune applicazioni molto specializzate potrebbero accedere a queste classi direttamente, ma la maggior parte dei clienti di un compilatore non vuole conoscere i dettagli della compilazione, ma vuole solo compilare del codice.

Per rendere trasparenti gli aspetti implementativi della compilazione ai suoi clienti, il sottosistema introduce la classe Compilatore.

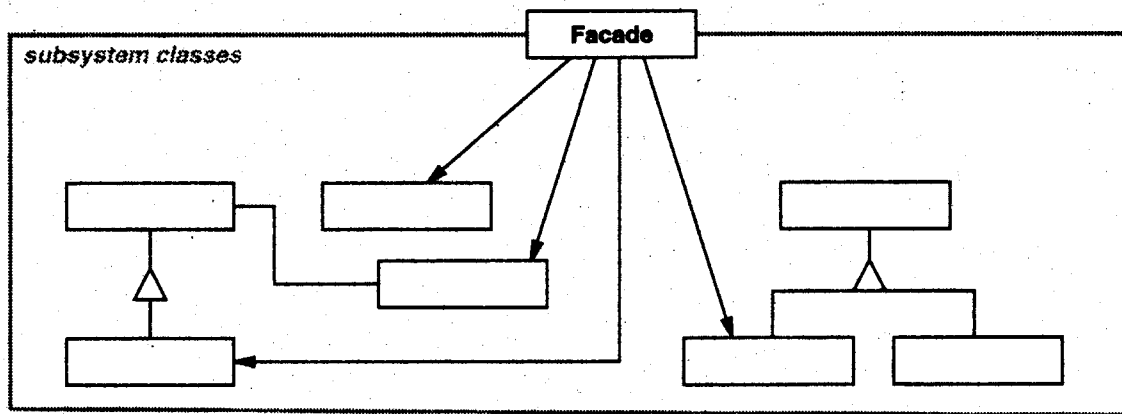
Questa classe costituisce un'interfaccia alle funzionalità del compilatore. La classe Compilatore rende la vita più facile alla maggior parte dei programmatori senza nascondere le funzionalità di più basso livello.



## Applicabilità:

Usa il pattern Facade quando:

- Vuoi fornire un'interfaccia semplice ad un sottosistema complesso;
- Ci sono molte dipendenze tra i clienti e le classi di implementazione di un'astrazione; il pattern Facade permette di disaccoppiare il sottosistema dai clienti, promuovendo l'indipendenza e quindi la portabilità;
- Vuoi organizzare i tuoi sottosistemi a livelli; usa un pattern Facade per definire un punto di ingresso per ogni livello di sottosistema. Se i sottosistemi sono dipendenti, puoi semplificare le dipendenze tra loro facendoli comunicare solo attraverso i Facade.



## Partecipanti:

### ➤ Facade (Compilatore)

- Conosce quali classi del sottosistema sono responsabili per soddisfare una richiesta;
- Delega le richieste del cliente ad appropriati oggetto del sottosistema;

### ➤ Classi del Sottosistema (Scanner, Parser, ProgramNode, ecc.)

- Implementano le funzionalità del sottosistema;
- Gestiscono il lavoro assegnato dall'oggetto Facade;
- Non hanno conoscenza dell'oggetto Facade; non mantengono nessun riferimento ad esso.

## Collaborazioni:

- I clienti comunicano con il sottosistema inviando richieste al Facade, che le inoltra agli oggetti appropriati del sottosistema. Il Facade potrebbe svolgere del lavoro per tradurre la sua interfaccia alle interfacce del sottosistema;
- I clienti che usano il Facade non devono accedere agli oggetti del sottosistema direttamente.

## Conseguenze:

Il pattern Facade produce i benefici seguenti:

- Inserisce uno schermo tra i clienti ed i componenti del sottosistema, riducendo il numero di oggetti visti dai clienti e rendendo il sottosistema più facile da usare;

- Promuove un accoppiamento debole tra il sottosistema ed i suoi clienti. Questo permette di variare i componenti del sottosistema senza influenzare i suoi clienti. Possono eliminare dipendenze circolari e complesse. Possono ridurre le dipendenze di compilazione permettendo di minimizzare i tempi di ricompilazione.
- Non proibisce alle applicazioni di usare le classi del sottosistema direttamente, se strettamente necessario.

### **Implementazione:**

Considera le questioni seguenti nell'implementazione di un Facade

- Ridurre l'accoppiamento cliente-sottosistema. L'accoppiamento cliente-sottosistema può essere ulteriormente ridotto rendendo la classe che realizza il pattern Facade astratta. In questo modo, i clienti comunicano con il sottosistema attraverso l'interfaccia della classe astratta e le diverse sottoclassi concrete consentono di avere differenti implementazioni del sottosistema;
- Classi pubbliche e classi private. Sarebbe utile implementare alcune classi come private al sottosistema. Questo può essere fatto utilizzando i namespace che ormai diversi linguaggi object-oriented mettono a disposizione.

### **Codice campione:**

Viene data una descrizione di una possibile implementazione del pattern Facade per l'esempio del compilatore (si rinvia al testo di riferimento per un esempio).

### **Usi conosciuti:**

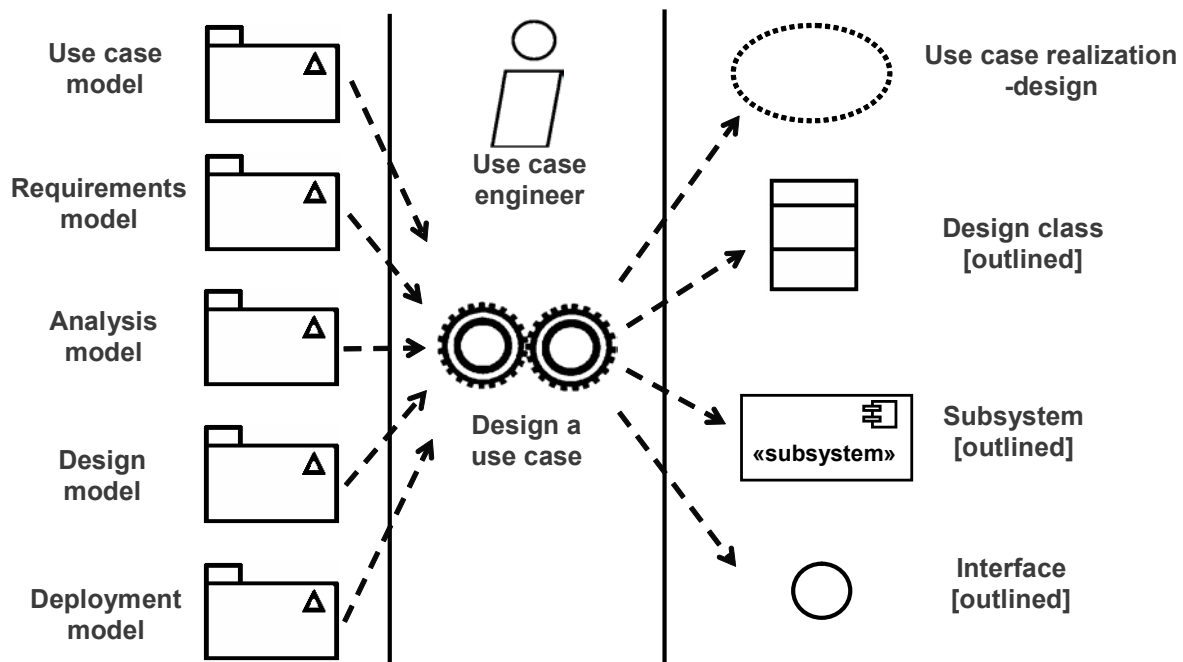
Compilatori, gestori della memoria, ecc.

### **Pattern correlati:**

Abstract Factory, Mediator.

## Realizzazioni dei casi d'uso nel Progetto

La figura seguente mostra l'attività di Progetto di un caso d'uso.



Nella figura, il modello di Progetto ed il modello di dislocazione sono presentati come ingressi dell'attività, sebbene siano fondamentalmente le principali uscite. Questo a dimostrare la natura iterativa del processo UP.

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;
- Diagrammi di classi contenenti le classi di progetto.

Le realizzazioni dei casi d'uso nel progetto sono molto più dettagliate rispetto all'analisi. In ogni caso, limita il lavoro a ciò che è utile (progetto strategico) e lascia il resto all'implementazione (progetto tattico).

### ***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.

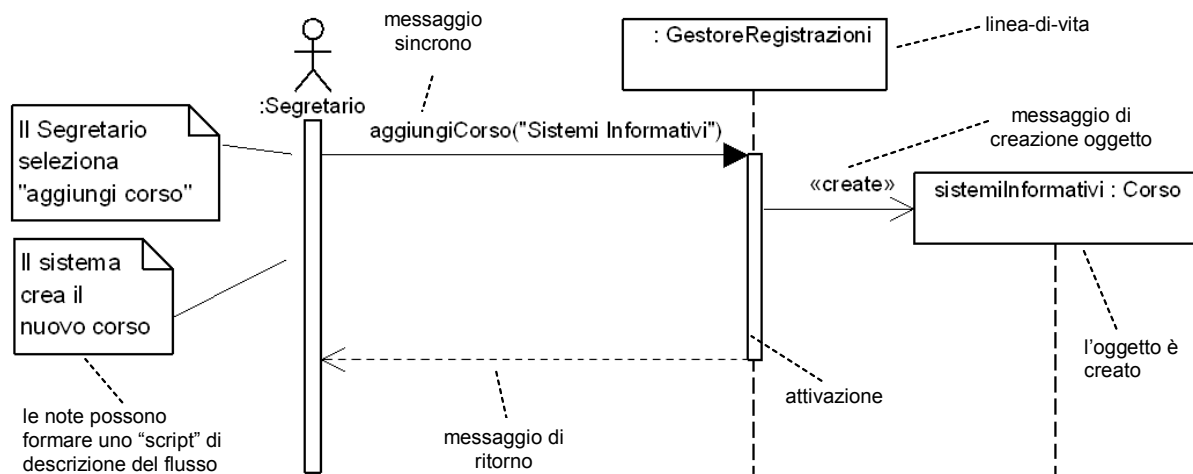
I diagrammi di interazione nel progetto possono essere una semplice rifinitura dei diagrammi di interazione nell'analisi oppure diagrammi completamente nuovi costruiti per illustrare aspetti tecnici che sono sorti durante il progetto.

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.

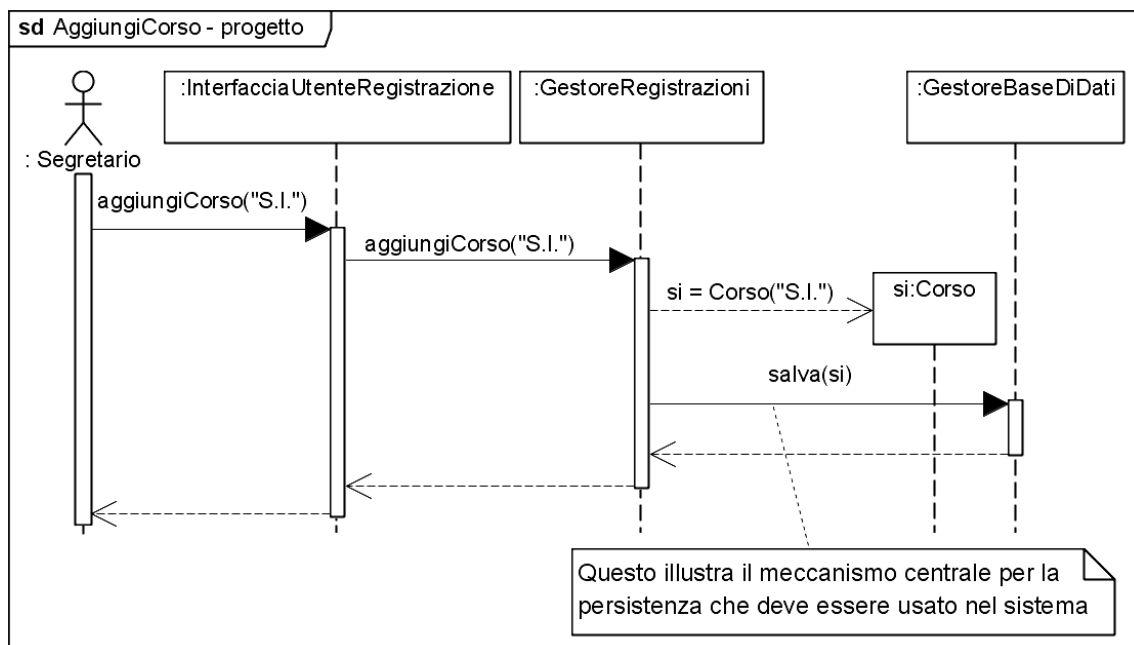
Per capire il ruolo dei diagrammi di sequenza, consideriamo il caso d'uso seguente.

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>

La figura seguente mostra il diagramma di sequenza introdotto nell'analisi.



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



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);



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.

## Modellare la concorrenza

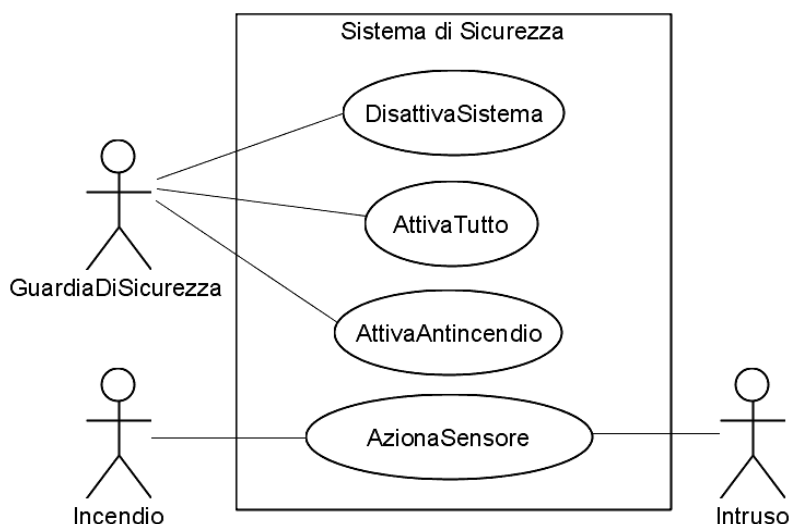
UML 2 fornisce una varietà di supporti per la concorrenza:

1. classi attive;
2. fork and join nei diagrammi di attività;
3. l'operatore `par` nei diagrammi di sequenza;
4. i prefissi del numero di sequenza nei diagrammi di comunicazione;
5. le tracce multiple nei diagrammi temporali;
6. stati compositi ortogonali nelle macchine a stati.

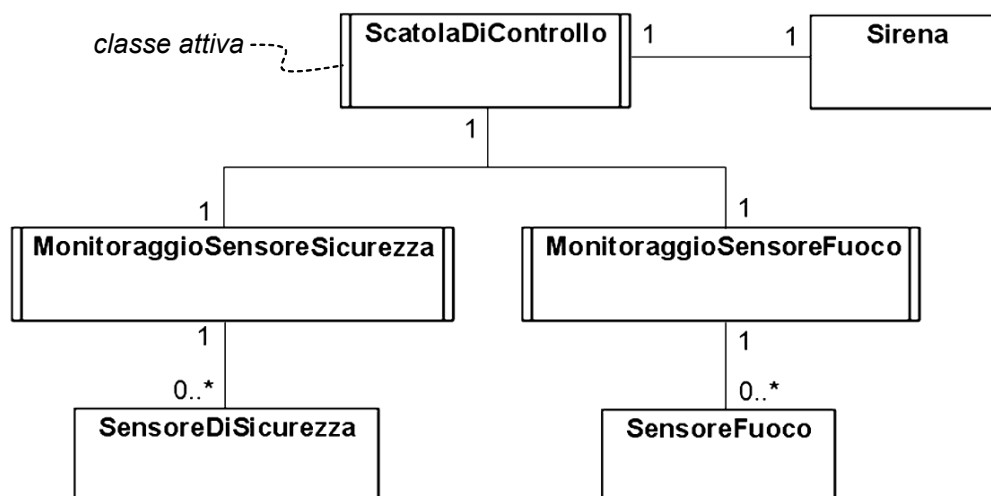
### *Le classi attive*

Il principio fondamentale per modellare la concorrenza è che ogni thread di controllo o ogni processo concorrente è modellato come un oggetto attivo, cioè un oggetto che incapsula il proprio thread di controllo. Oggetti attivi e classi attive sono modellati come oggetti e classi normali ma con un doppio bordo a sinistra ed a destra.

Modellare la concorrenza è molto importante per esempio nei sistemi embedded. Consideriamo un sistema di sicurezza, rappresentato dal diagramma dei casi d'uso seguente.



In generale, nei sistemi embedded l'hardware è una buona sorgente di classi. L'hardware in questo caso consiste nella scatola di controllo, la sirena, l'insieme di rivelatori di fuoco e di intrusione. La scatola di controllo contiene un controllore per ogni tipo differente di sensore. Quindi



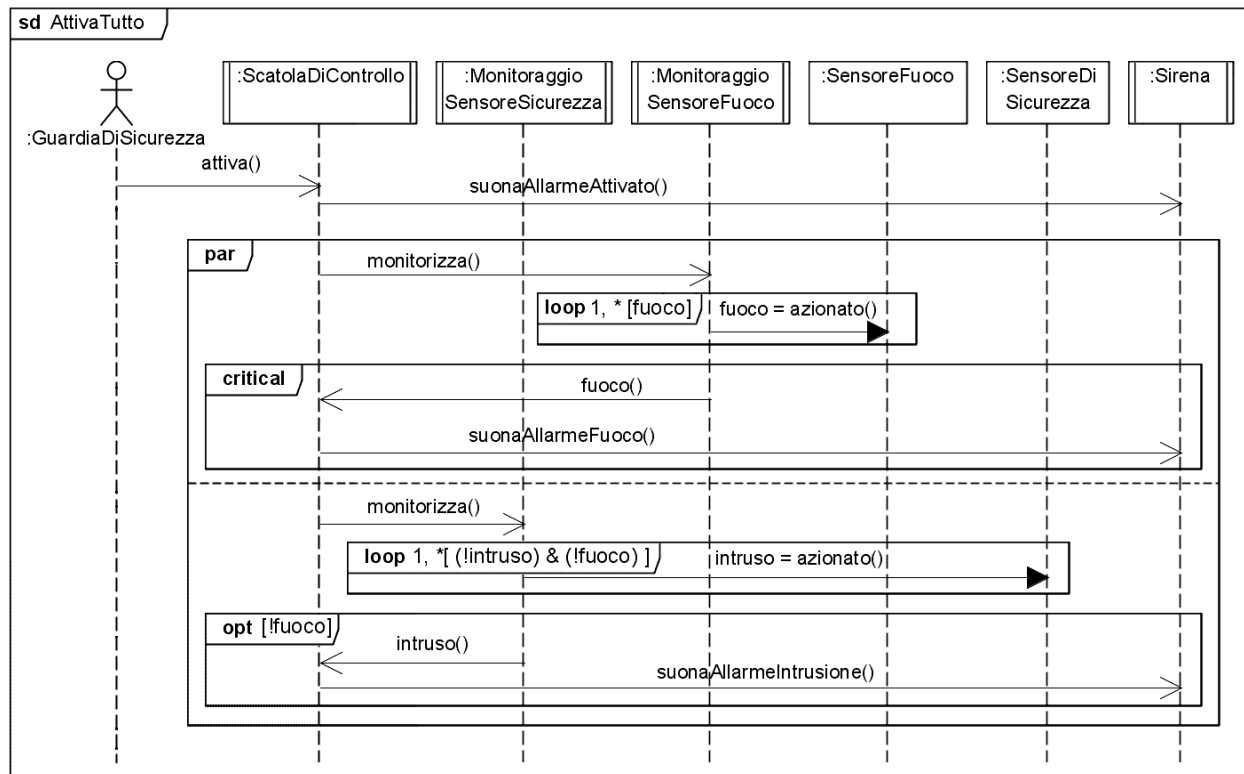
Le classi ScatolaDiControllo, MonitoraggioSensoreSicurezza e MonitoraggioSensoreFuoco sono classi attive in quanto devono monitorare continuamente i sensori relativi.

## Concorrenza nei diagrammi di sequenza

Per illustrare la concorrenza nei diagrammi di sequenza consideriamo il caso d'uso seguente:

Caso d'uso: <i>AttivaTutto</i>
ID: 2
Breve descrizione: <i>Attiva il sistema.</i>
Attori primari: <i>GuardiaDiSicurezza</i>
Attori secondari: <i>Nessuno</i>
Precondizioni: <i>1. La GuardiaDiSicurezza ha la chiave di attivazione</i>
Flusso principale: <i>1. La GuardiaDiSicurezza usa la chiave di attivazione per accendere il sistema</i> <i>2. Il sistema inizia a monitorare i sensori di sicurezza ed i sensori di fuoco</i> <i>3. Il sistema fa suonare la sirena per indicare che è in funzione.</i>
Postcondizioni: <i>1. Il sistema di sicurezza è attivato</i> <i>2. Il sistema di sicurezza sta monitorando i sensori</i>
Flussi alternativi: <i>Nessuno.</i>

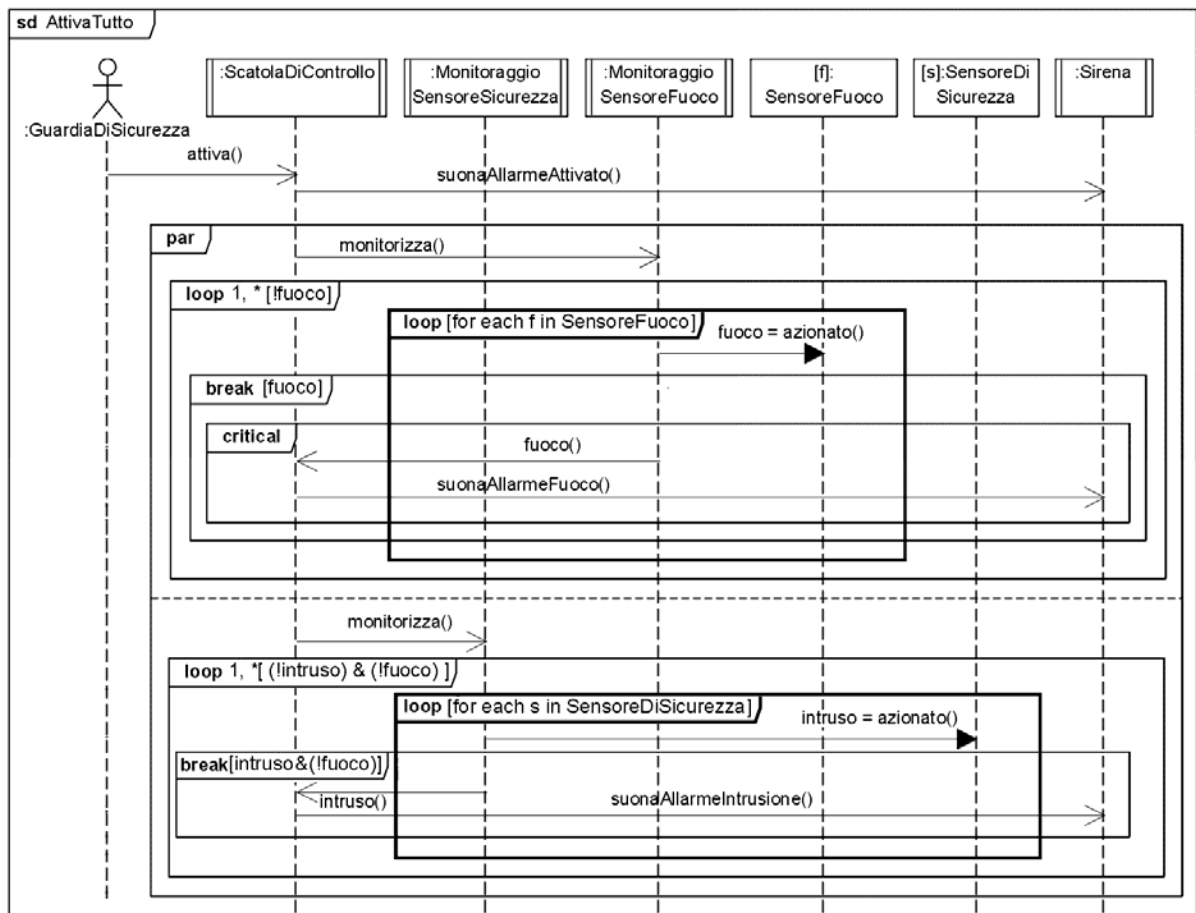
Il diagramma di sequenza relativo al caso d'uso è



La sezione critical rappresenta un frammento atomico.

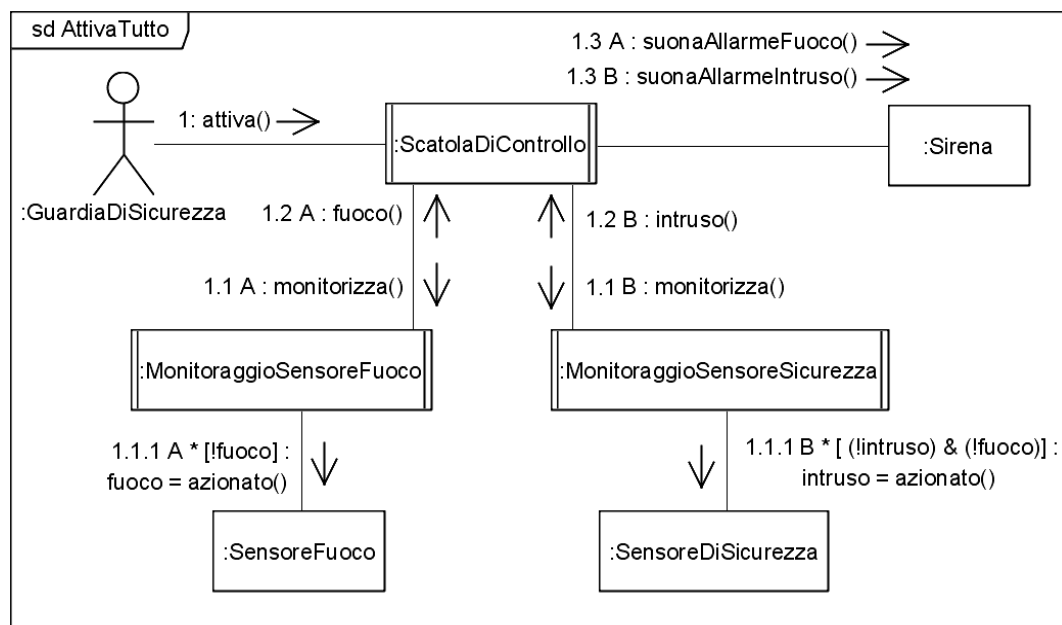
L'allarme incendio deve avere sempre precedenza sull'allarme intrusione. Questa è la ragione per cui il loop nell'operando 2 di par termina quando viene rilevato o un incendio o un'intrusione.

OSSERVAZIONE: nell'esempio abbiamo mostrato solo un rilevatore di fuoco ed uno di intrusione. Questo è sufficiente per rappresentare il comportamento. Se avessimo voluto un dettaglio maggiore avremmo dovuto usare il diagramma seguente.



## Concorrenza nei diagrammi di comunicazione

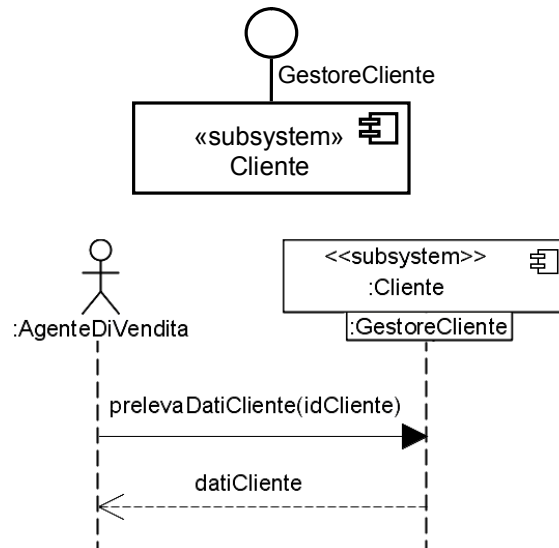
La concorrenza nei diagrammi di comunicazione è ottenuta facendo seguire un'etichetta al numero di sequenza: l'etichetta indica il thread di controllo. Nella figura seguente, che presenta la stessa informazione del diagramma di sequenza con un solo sensore, i thread identificati dalle lettere A e B sono concorrenti.



## Interazioni tra sottosistemi

Modellare le interazioni tra i sottosistemi può essere utile perché fornisce una vista ad alto livello di come l'architettura realizza i casi d'uso senza entrare nel dettaglio della comunicazione tra oggetti.

Ogni sottosistema è trattato come una scatola nera che semplicemente fornisce e richiede servizi specificati dalle sue interfacce pubbliche fornite e richieste.



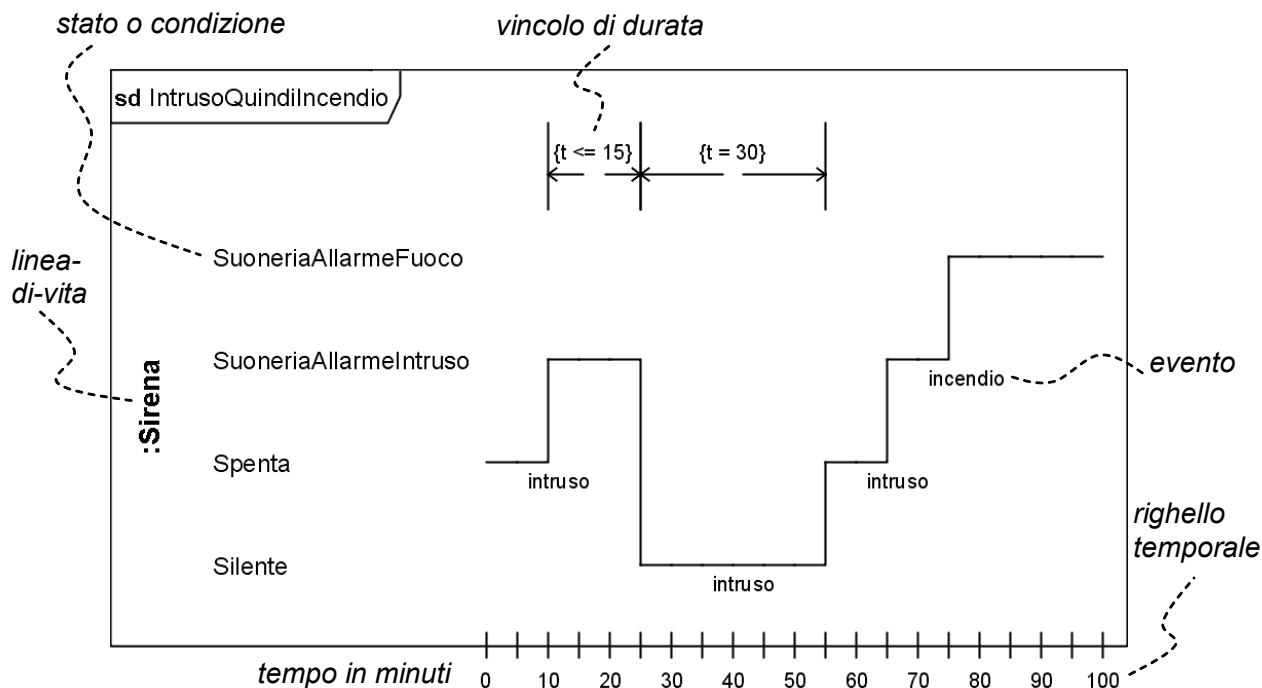
## Diagrammi temporali

Il diagramma temporale è un diagramma introdotto in UML 2.0 ed è un tipo di diagramma di interazione che si concentra sulla modellazione dei vincoli temporali.

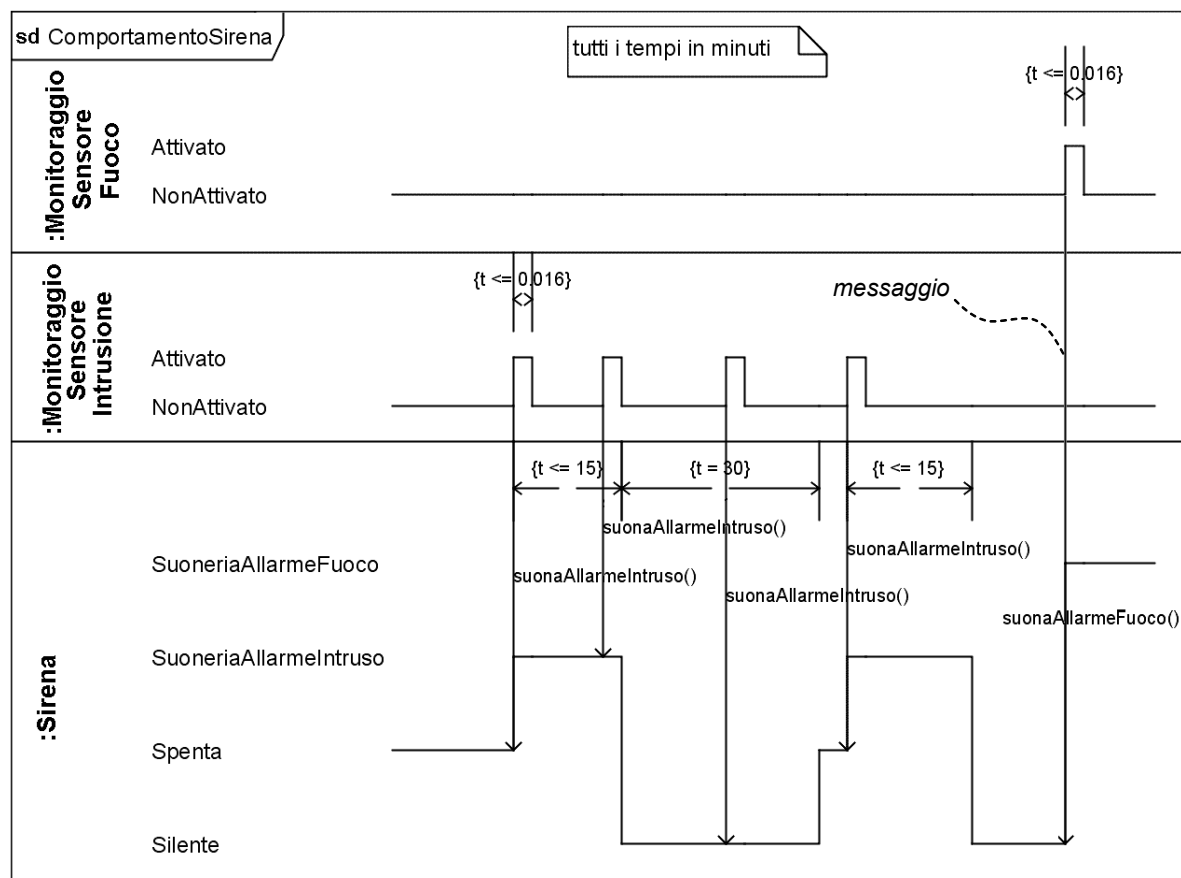
Questi diagrammi risultano particolarmente importanti nelle applicazioni real-time.

Nei diagrammi temporali il tempo cresce da sinistra verso destra e le linee di vita con i loro stati e condizioni sono presentate verticalmente.

Facendo riferimento ai casi d'uso introdotti precedentemente, il diagramma seguente illustra ciò che accade quando c'è un'intrusione, seguita da un incendio.



Il diagramma seguente mostra come il diagramma temporale possa illustrare vincoli temporali nell'interazione tra due o più linee di vita. Nel diagramma si nota come la sirena risponda sempre quando si verifica un incendio e solo se si trova nello stato spenta ad un'intrusione.



## Macchine a Stati

Sia i diagrammi di attività che i diagrammi di macchine a stati modellano il comportamento dinamico del sistema, ma hanno semantiche ed obiettivi differenti. I diagrammi di attività tendono ad essere usati per modellare processi di business in cui partecipano diversi oggetti. Le macchine a stati tendono ad essere usate per modellare la storia del ciclo di vita di un singolo oggetto reattivo come una macchina a stati finiti, cioè una macchina che può esistere in un numero finito di stati.

Tre elementi chiave:

- **stato** – condizione o situazione nella vita di un oggetto durante la quale soddisfa qualche condizione, esegue qualche attività o aspetta qualche evento;
- **evento** – specifica di un'occorrenza rilevante che può essere localizzata nel tempo e nello spazio
- **transizione** – passaggio da uno stato ad un altro in risposta ad un evento.

Un **oggetto reattivo** è un oggetto che risponde ad eventi sia interni che esterni, può generare eventi interni, ha un ciclo di vita modellato come sequenza di stati, transizioni ed eventi, può avere il comportamento attuale dipendente dal comportamento passato.

Le macchine a stati possono essere usate per modellare il comportamento di classificatori quali classi, casi d'uso, sottosistemi e sistemi interi.

### *Macchine a stati per modellare comportamento e protocolli*

Le macchine a stati per modellare il comportamento (behavioral state machine) usano stati, transizioni ed eventi per definire il comportamento del classificatore di contesto. Queste macchine a stati possono essere usate solo per classificatori che hanno un comportamento (non, per esempio, per interfacce o porte).

Le macchine a stati per modellare protocolli (protocol state machine) usano stati, transizioni ed eventi per definire il protocollo del classificatore di contesto. Questo protocollo include:

- le condizioni che permettono di invocare le operazioni sul classificatore o sulle sue istanze;
- i risultati delle invocazioni delle operazioni;
- l'ordine delle invocazioni.

Nelle macchine a stati per modellare protocolli gli stati non possono specificare azioni (è un compito delle macchine a stati per modellare comportamento).

NOTA BENE: normalmente non viene fatta distinzione tra le due macchine a stati. Se si rendesse necessaria questa distinzione, si potrebbe usare la parola chiave {protocol} dopo il nome della macchina a stati per protocolli.

Le macchine a stati vengono usate principalmente per modellare il comportamento dinamico dei classificatori, in particolare delle classi. Una classe può avere una macchina a stati per modellare il comportamento ed una o più macchine a stati per modellare il protocollo.

Le macchine a stati associate ad una classe dovrebbero specificare il comportamento od i protocolli richiesti da tutti i casi d'uso a cui partecipano istanze della classe.

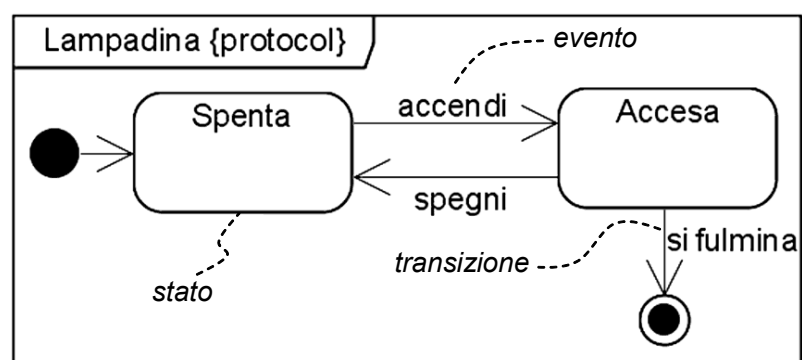
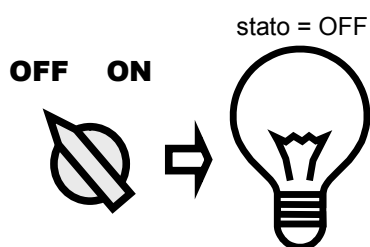
Tipicamente, le macchine a stati vengono usate nella fase di Elaborazione e nei primi stadi della fase di Costruzione, quando si tenta di capire le classi in sufficiente dettaglio.

Il problema più grande che si incontra lavorando con le macchine a stati è testarle. Gli strumenti di ausilio alla modellazione UML non offrono in generale nessun aiuto alla verifica automatica delle macchine a stati. Il modo migliore per verificarle è quindi simularle con degli strumenti appropriati.

### ***Diagrammi di macchina a stati***

Un diagramma di macchina a stati contiene esattamente una macchina a stati per un singolo oggetto reattivo.

Esempio: sistema composto da lampadina ed interruttore.



- Gli stati sono rappresentati come rettangoli arrotondati, ad eccezione dello stato iniziale (cerchio pieno) e lo stato finale (occhio di toro);
- Le transizioni indicano percorsi possibili tra gli stati e sono modellate da una freccia;
- Gli eventi sono scritti sopra alla transizione che attivano.

Ogni macchina a stati deve avere uno stato iniziale e, a meno di cicli infiniti degli stati, uno stato finale. Lo stato iniziale e lo stato finale sono degli pseudo-stati (nessun stato reale).



Nelle macchine a stati, gli eventi sono considerati istantanei. Questo evita dover trattare corse critiche.

## Stati

Lo stato è determinato da:

- I valori degli attributi dell'oggetto di contesto;
- Le relazioni che questo oggetto ha con altri oggetti;
- Le attività che l'oggetto sta eseguendo.

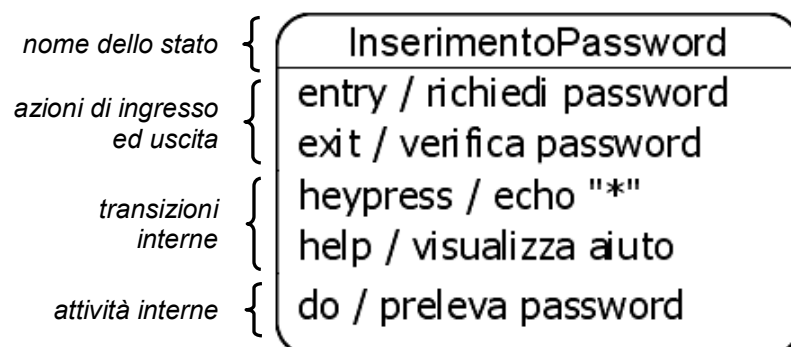
ATTENZIONE: identifica gli stati che hanno senso nel sistema. Nell'esempio della lampadina, certo non è il movimento di un atomo a generare un cambio di stato.

Quando conviene usare una macchina a stati pensando allo stato di un oggetto? Quando il numero dei possibili stati è limitato e quando è significativo capire come si passa da uno stato ad un altro. Consideriamo la classe seguente:

Colore
rosso : int verde : int blu : int

Assumendo che rosso, verde e blu possano assumere valori tra 0 e 255, che macchina a stati dovremmo definire? I possibili stati sono  $256*256*256 = 16777216$ . Quale semantica avrebbe uno stato?

La sintassi di uno stato è la seguente



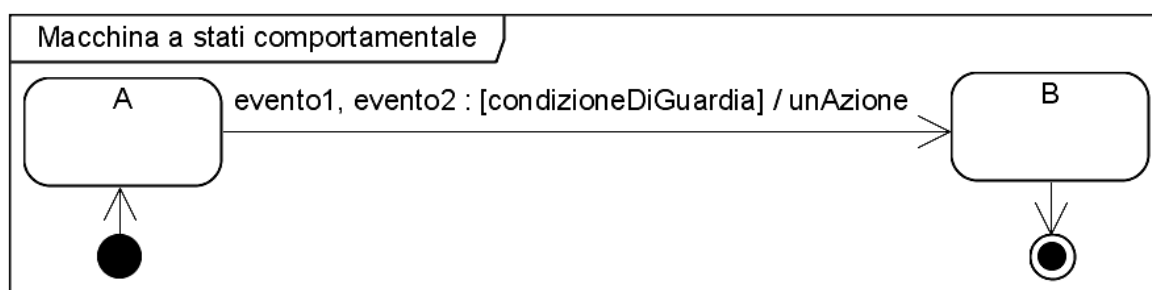
Le azioni sono considerate istantanee e non interrompibili, mentre le attività possono essere interrotte e durano un tempo finito. Ogni azione in uno stato è associata con una transizione interna attivata da un evento. In uno stato, ci può essere un numero qualsiasi di transizioni interne ed azioni.

Le transizioni interne catturano eventi che sono importanti da modellare, ma non provocano un cambiamento di stato. Per esempio, l'inserimento dei caratteri da parte dell'utente nello stato rappresentato nel diagramma precedente.

Due azioni speciali, l'azione di ingresso e di uscita sono associate con gli eventi speciali entry e exit. Le attività sono introdotte dalla parola chiave do.

## Transizioni

La figura seguente riassume la sintassi per le transizioni



Ogni transizione ha tre elementi opzionali:

- Zero o più eventi –specificano occorrenze interne o esterne che possono attivare la transizione;
- Zero o più condizioni di guardia – espressioni booleane che devono essere vere prima che transizione possa occorrere.
- Zero o più azioni – sono parti di lavoro associate alla transizione e vengono eseguite quando la transizione viene attivata.

Nella figura precedente gli elementi opzionali associati alla transizione hanno il significato seguente:

“Quando l’evento 1 o l’evento 2 si verificano, se la condizione di guardia è vera, allora esegui l’azione e subito dopo vai nello stato B.”

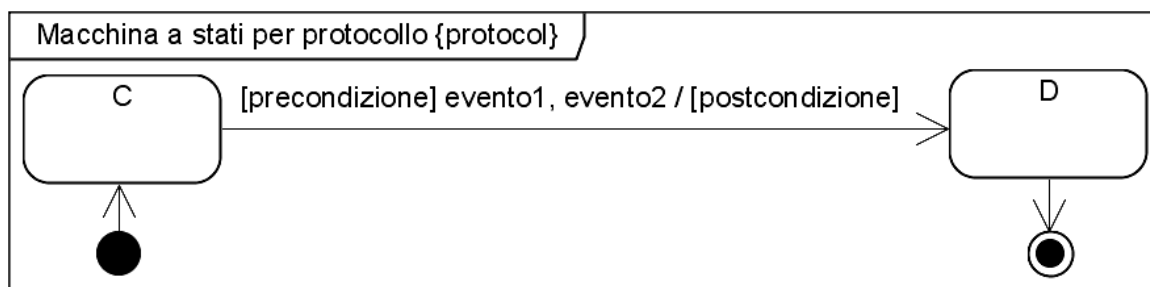
L’azione può coinvolgere variabili che sono visibili nella macchina a stati. Per esempio:

azioneEseguita(eventoAzione)/comando = eventoAzione.ottieniComandoAzione()

Il nome dell’azione viene memorizzato nella variabile comando.

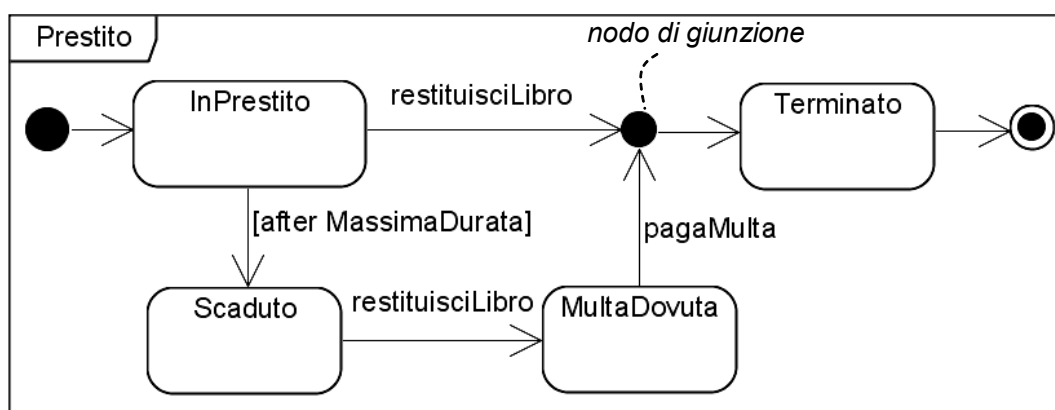
Per le macchine a stati per protocolli, le transizioni hanno una sintassi leggermente differente:

- Non c’è alcuna azione;
- La condizione di guardia viene rimpiazzata da precondizioni e postcondizioni.

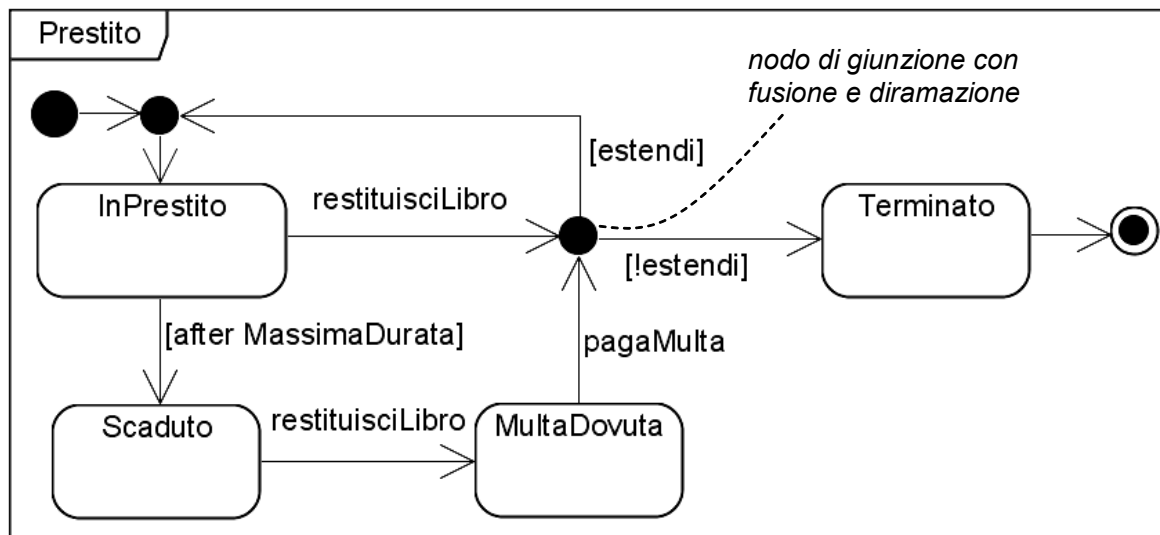


Se una transizione non ha nessun evento, è una *transizione automatica*.

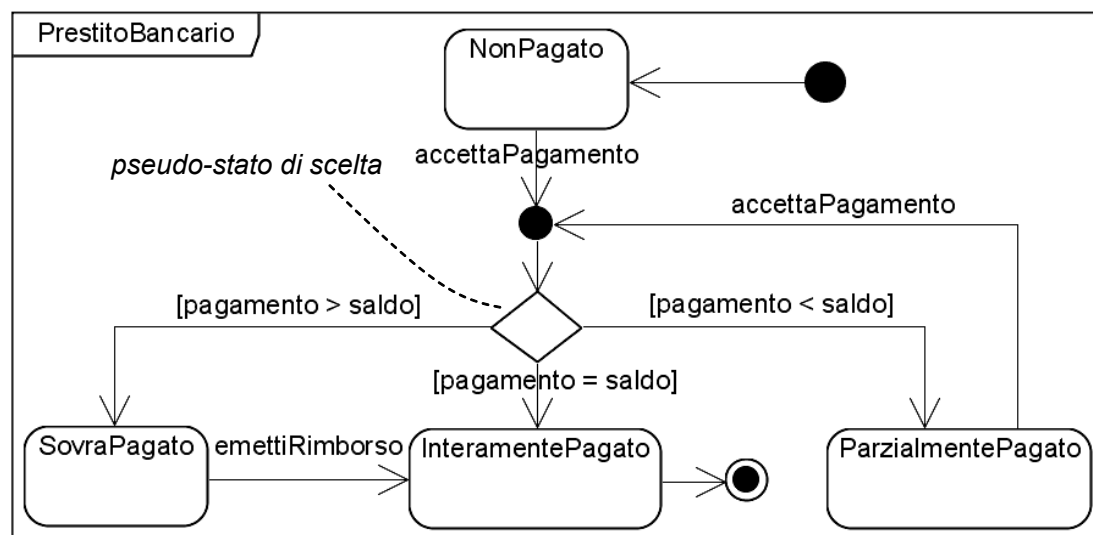
Le transizioni possono essere connesse da uno **pseudo-stato di giunzione**. La figura presenta la macchina a stati per la classe Prestito



Uno pseudo-stato di giunzione può avere più di una transizione di uscita. In questo caso, ogni transizione di uscita deve essere protetta da una condizione di guardia mutuamente esclusiva.

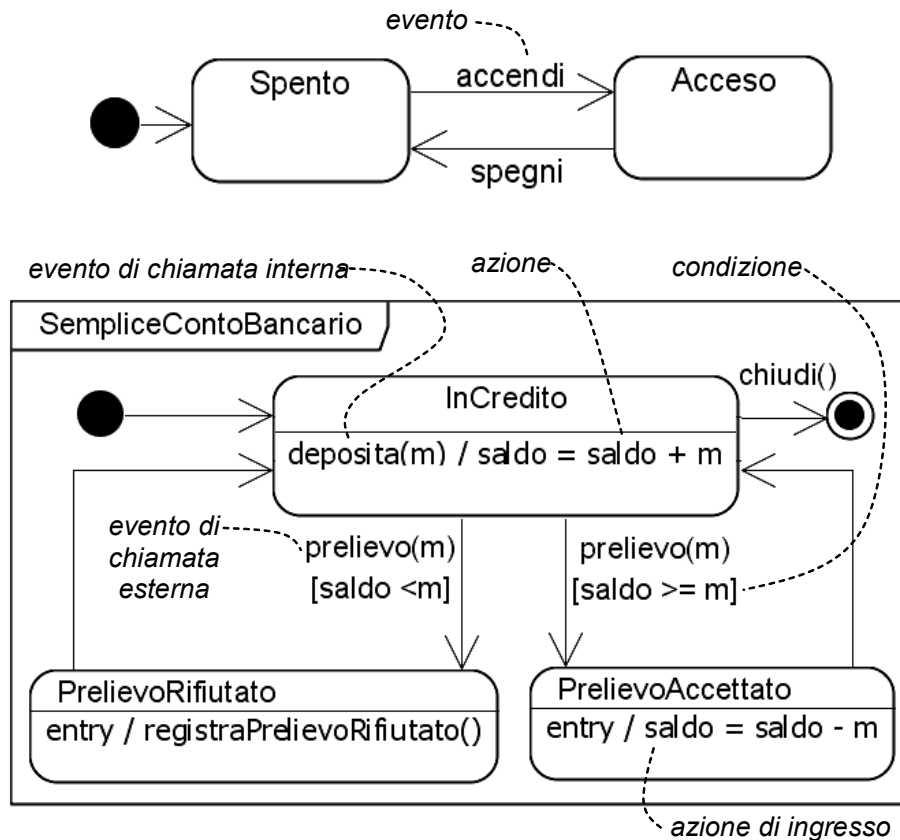


Per diramazioni semplici si può usare lo **pseudo-stato di scelta** mostrato nella figura seguente.



## Eventi

Un evento per UML è “la specifica di un’occorrenza degna di nota che si verifica nel tempo o nello spazio”. Eventi attivano transizioni nelle macchine a stati. Gli eventi possono essere presentati esternamente sulle transizioni (prima figura seguente) o all’interno degli stati (seconda figura).

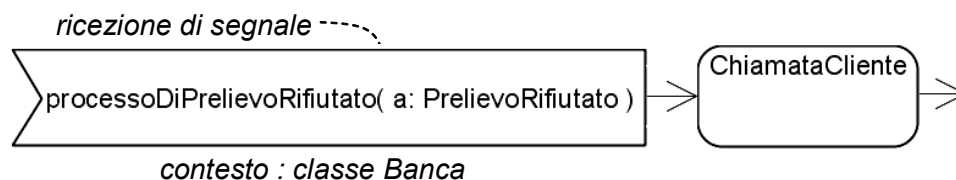
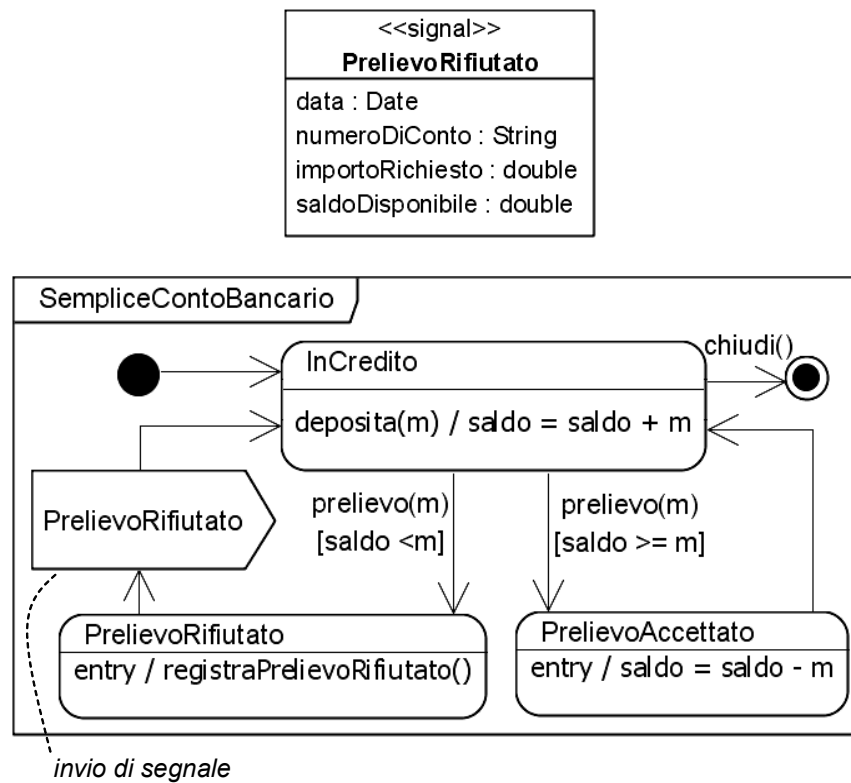


La classe `SempliceContoBancario` è soggetta ai vincoli di business seguenti:

- Il saldo del conto deve essere sempre superiore o uguale a zero;
- Un prelievo non verrà effettuato nel caso in cui porterebbe il saldo del conto sotto allo zero

Quattro tipi di evento:

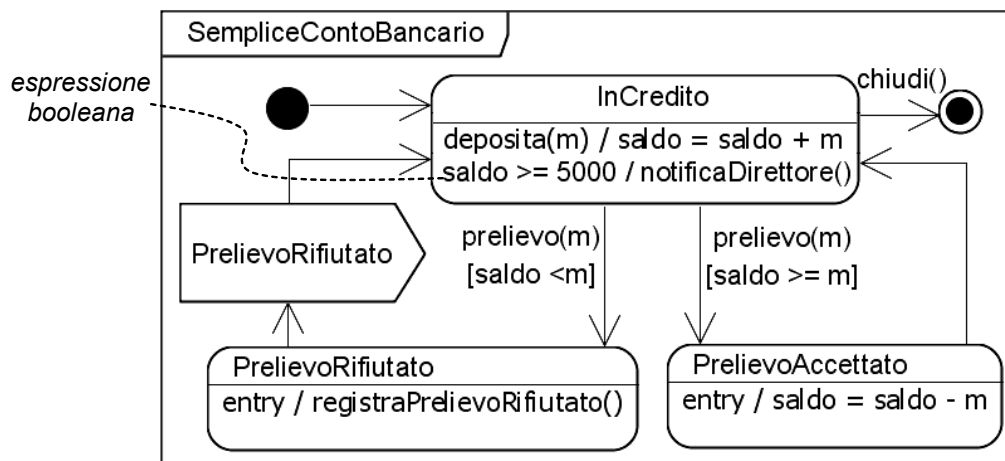
- **Call** – è una richiesta di invocazione di un'operazione su un'istanza della classe di contesto. Deve avere la stessa intestazione dell'operazione della classe di contesto. La ricezione dell'evento attiva l'esecuzione dell'operazione. La figura precedente presenta eventi call sia interni che esterni. Può essere specificata una sequenza di azioni separate da punto e virgola per ogni evento call.
- **Signal** – è un pacchetto di informazioni inviato in modo asincrono tra oggetti. Il segnale è modellato con una classe, tipicamente senza operazioni, stereotipata come «signal».



La ricezione di un segnale può anche essere modellata usando la notazione standard già vista `eventName/ action`.

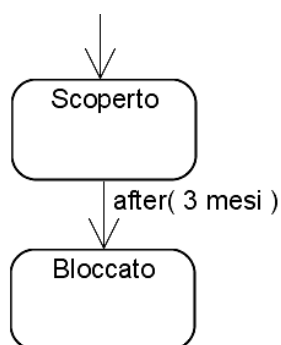
- **Change** – l'evento è specificato da un'espressione booleana: quando l'espressione diventa vera, viene eseguita l'azione associata all'evento. Tutti i valori usati nell'espressione booleana devono essere visibili dalla classe di contesto. Dal punto di vista dell'implementazione, un evento **change** richiede di testare continuamente la condizione booleana.

**ATTENZIONE:** gli eventi **change** sono *positive edge triggered*, ossia sono attivati ogni volta che il valore dell'espressione booleana cambia da *false* a *true*.



- **Time** – sono eventi dipendenti dal tempo e vengono indicati spesso con la parola chiave *when* o *after*. La parola chiave *when* specifica un istante particolare in cui l'evento è attivato; *after* specifica una soglia dopo la quale l'evento è attivato.

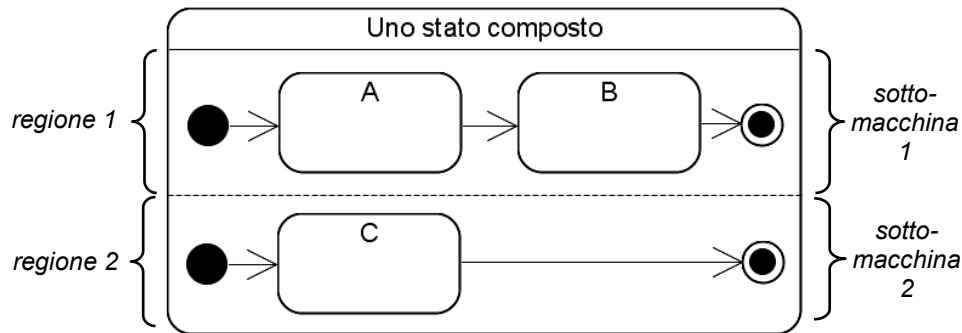
L'esempio seguente mostra un frammento di una macchina a stati di una classe ContoCorrente.



context: classe ContoDiCredito

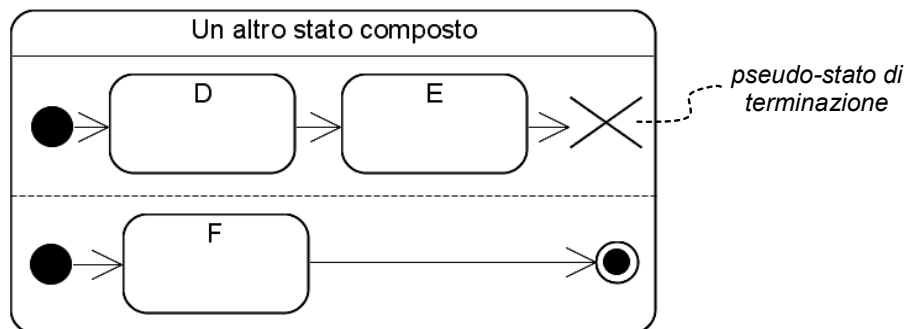
## Stati composti

Uno stato composto è uno stato che contiene stati annidati. Questi stati annidati sono organizzati in una o più macchine a stati chiamate *sottomacchine a stati*. Ogni sottomacchina esiste nella sua propria regione all'interno dell'icona dello stato composto.



**IMPORTANTE:** gli stati annidati ereditano tutte le transizioni degli stati che li contengono: se lo stato composto ha una transizione, ognuno degli stati annidati ha questa transizione.

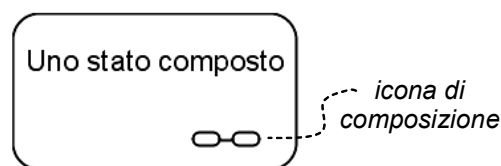
Lo pseudo-stato finale di una sottomacchina termina solo la sottomacchina. Se si vuole terminare l'esecuzione dell'intero stato composto, si deve usare lo pseudo-stato terminate.



**NOTA BENE:** stati annidati possono anche essere stati composti.

**ATTENZIONE:** evita che l'annidamento superi i due o tre livelli.

Uno stato composto si può rappresentare con l'icona seguente:

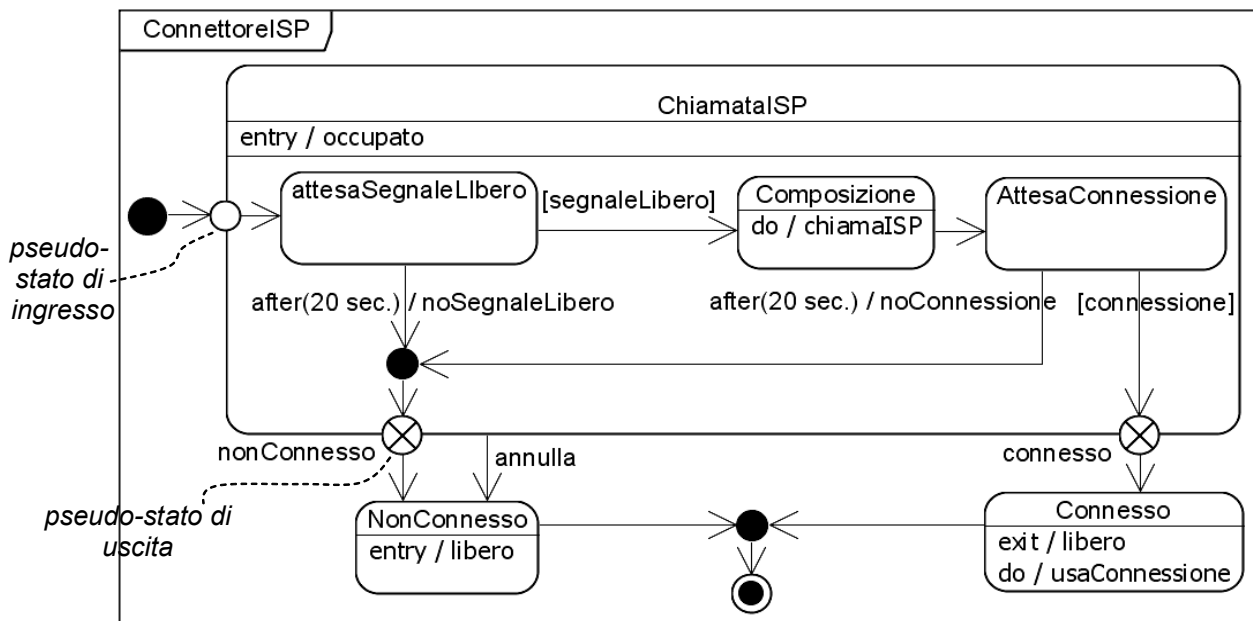


Ci sono due tipi di stati composti: stati composti semplici ed ortogonali.



## Stati composti semplici

Uno stato composto semplice è uno stato composto che contiene una singola regione.



NOTA: La transizione attivata dall'evento "annulla" è ereditata da tutti i sottostati nella sottomacchina a stati.

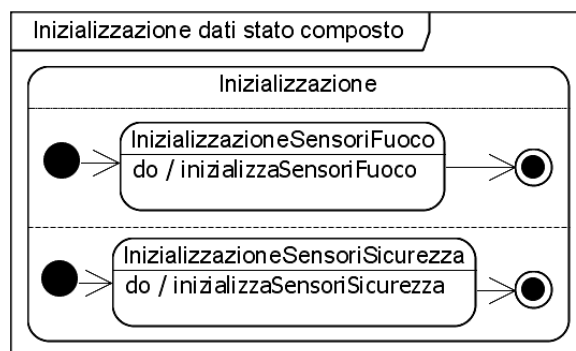
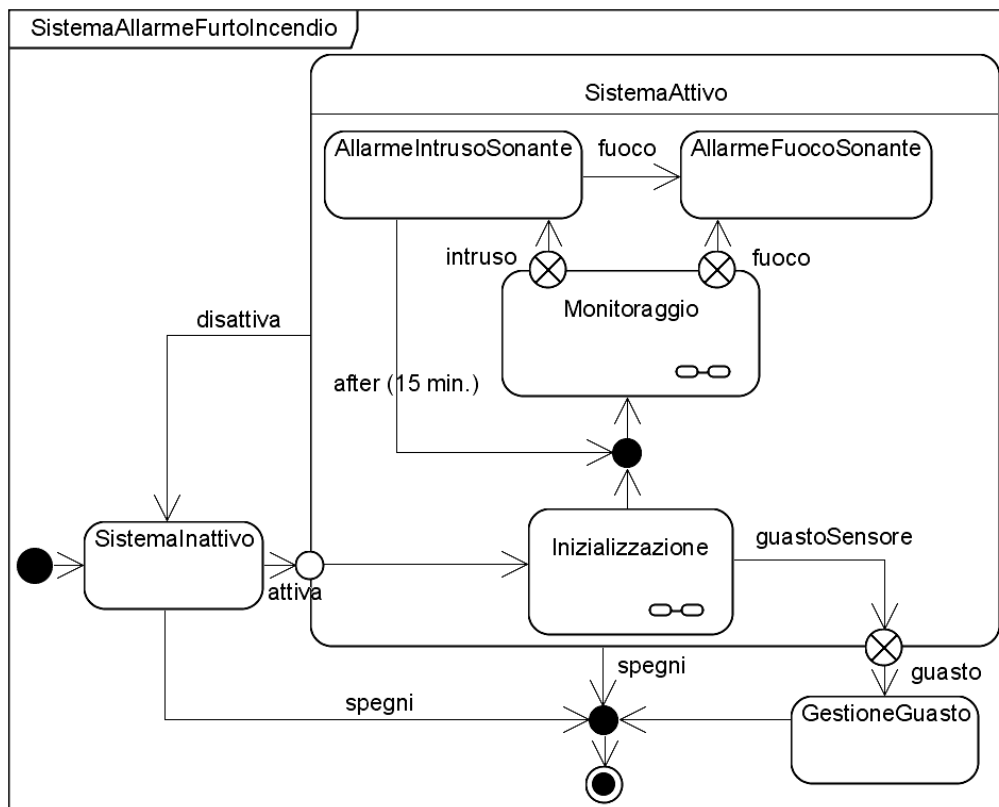
NOTA: la sottomacchina a stati ha un pseudostato di ingresso (cerchio vuoto) e due pseudostati di uscita (cerchio vuoto con croce).

## Stati composti ortogonali

Gli stati composti ortogonali contengono due o più sottomacchine a stati che vengono eseguite concorrentemente. Quando si entra nello stato composto, tutte le sue sottomacchine a stati iniziano subito l'esecuzione (fork implicito).

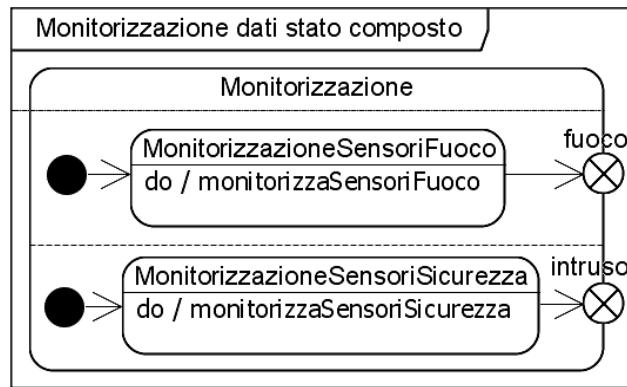
Ci sono due modi per uscire dallo stato composto:

- Tutte le sottomacchine terminano (join implicito);
- Una delle sottomacchine esegue una transizione verso uno stato esterno allo stato composto, per esempio usando uno pseudo-stato di uscita; in questo caso, tutte le altre sottomacchine vengono abortite.



In condizioni normali (quando non si verifica un errore nell'inizializzazione), si esce dallo stato composto Inizializzazione quando entrambe le sottomacchine terminano. Questo comportamento corrisponde ad un join tra le due sottomacchine.

Per quanto riguarda lo stato composto Monitoraggio, non c'è nessuna sincronizzazione tra le due sottomacchine: quando una sottomacchina termina, l'altra continua la sua normale esecuzione.



L'esempio visto prima evidenzia come gli stati composti riescano a modellare molto bene la concorrenza.

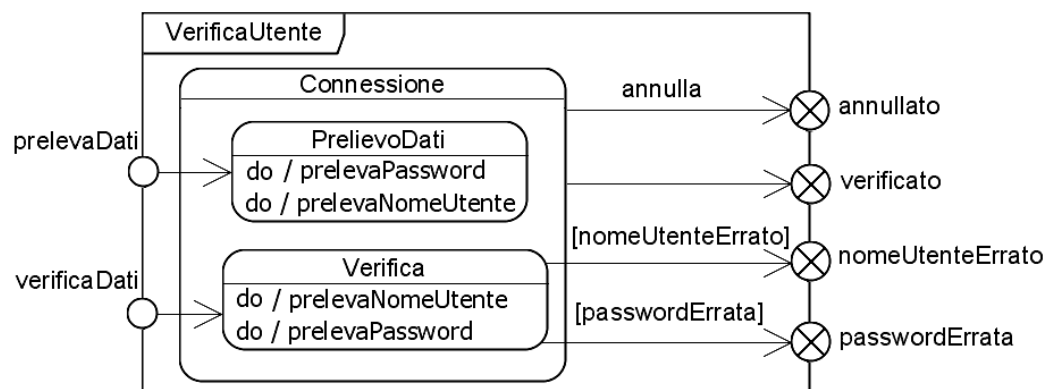
## Stati di una sottomacchina (Submachine states)

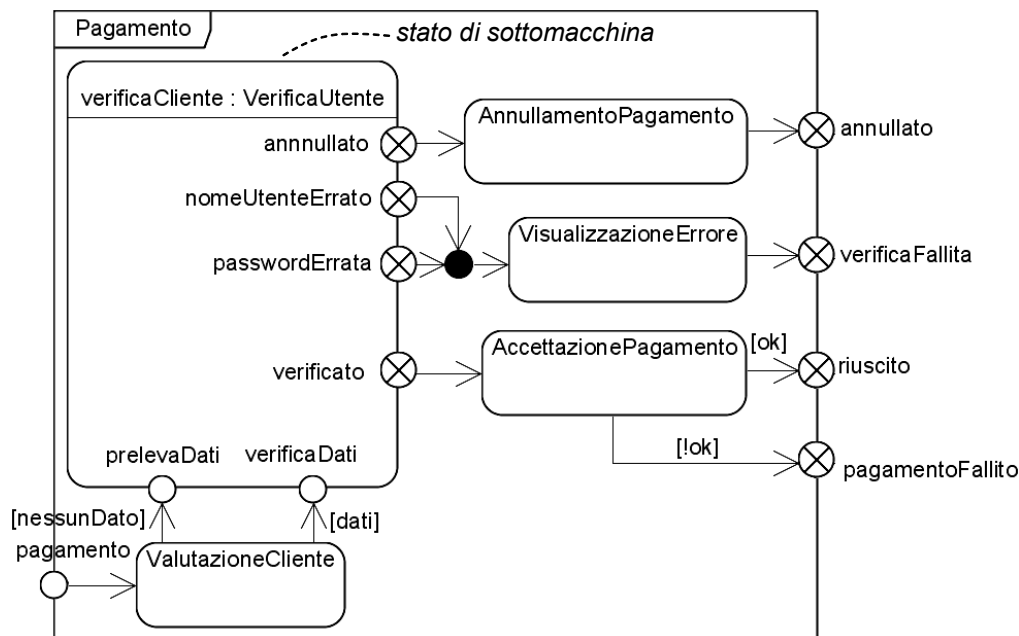
Uno stato di una sottomacchina è uno stato speciale che si riferisce ad una macchina a stati memorizzata in un diagramma separato. È come una chiamata a funzione da una macchina a stati ad un'altra. Gli stati di una sottomacchina sono semanticamente equivalenti a stati composti.

Gli stati di una sottomacchina sono nominati come segue:

macchina a stati: nome della macchina a stati riferita

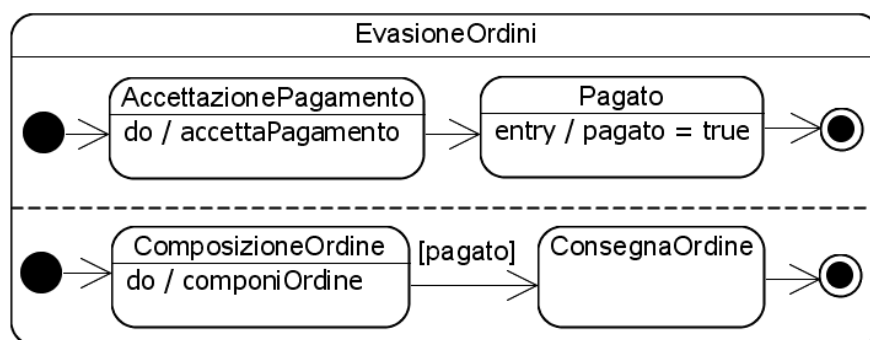
Gli stati di una sottomacchina forniscono un modo per riutilizzare il comportamento.





## Comunicazione tra sottomacchine (Submachine communication)

La comunicazione asincrona può essere raggiunta permettendo ad una sottomacchina di lasciare messaggi o flag quando viene eseguita. Altre sottomacchine possono prelevare questi flag quando sono pronte. Questi flag sono creati usando i valori degli attributi dell'oggetto di contesto della macchina a stati. In pratica, una sottomacchina inizializza i valori degli attributi, ed altre sottomacchine usano questi valori nelle condizioni di guardia sulle loro transizioni.



Nel diagramma seguente, l'attributo **pagato** consente di rappresentare che un ordine non può essere consegnato finché non sia stato pagato ed assemblato. L'attributo consente di realizzare una comunicazione asincrona tra le due sottomacchine usando un attributo come flag che una sottomacchina inizializza e l'altra legge.

## Storia (history)

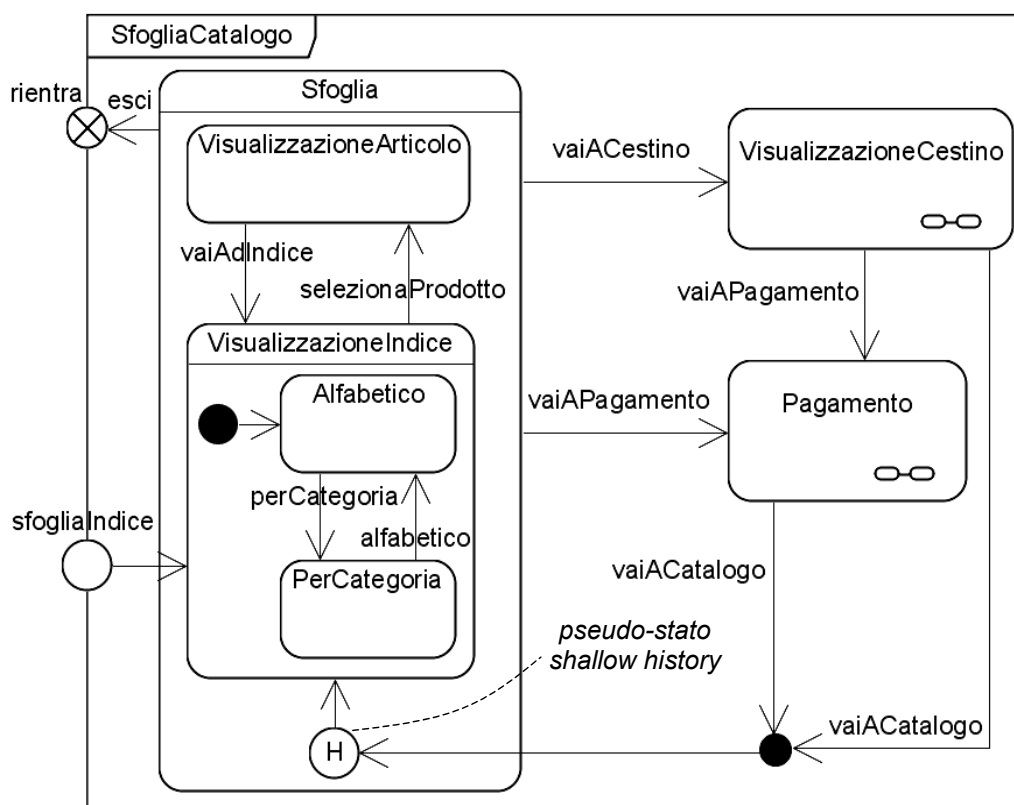
Quando si modella usando le macchine a stati, si incontra spesso la situazione seguente:

- Supponiamo di essere all'interno di un sottostato A di uno stato composto;
- Una transizione produce l'uscita dal sottostato A;
- Si passa attraverso uno o più stati esterni;
- Si ritorna con una transizione allo stato composto, ma si vorrebbe continuare dal sottostato A da dove si era lasciato lo stato composto.

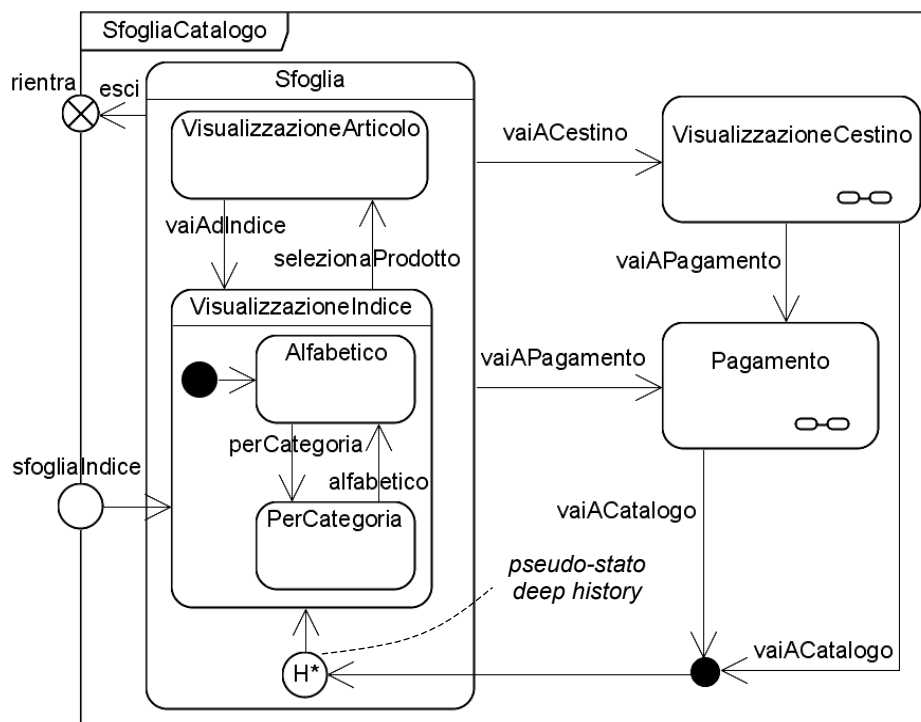
Come si fa a realizzare questo scenario?

Con lo pseudo-stato storia (history) che consente di dare ai superstati memoria dell'ultimo sottostato attivo prima che il superstato terminasse. Ci sono due tipi di pseudo-stati history: *shallow history* and *deep history*.

### Shallow and Deep history



Nella figura, lo pseudo-stato history è rappresentato da un cerchio con una lettera H all'interno. Lo pseudo-stato ricorda in quale sottostato ci si trovava quando il superstato è stato abbandonato. Quando da uno stato esterno si torna allo pseudo-stato history, la transizione viene inoltrata automaticamente all'ultimo stato memorizzato. Con lo pseudo-stato shallow history, l'ultimo stato memorizzato è allo stesso livello dello pseudo-stato history. Questo implica che se l'ultimo stato è uno stato composto, lo shallow history non si ricorderà i sottostati all'interno di questo stato. Il problema viene risolto con lo pseudo-stato **deep history**.



Sia gli pseudo-stati shallow history che gli pseudo-stati deep history possono avere molte transizioni entranti, ma un'unica transizione uscente. La transizione uscente viene attivata se è la prima volta che si entra nel superstato e quindi non c'è nessun sottostato memorizzato.

# Sistemi Informativi

Francesco Marcelloni – Mario G. Cimino

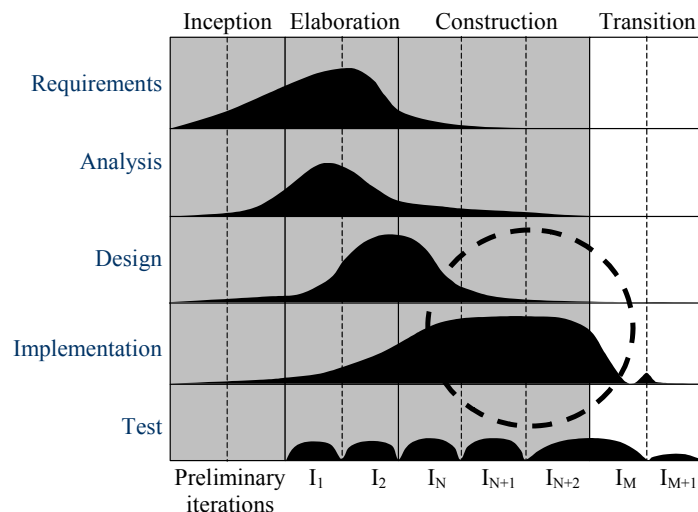


## Workflow Implementazione

395

### 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



396

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

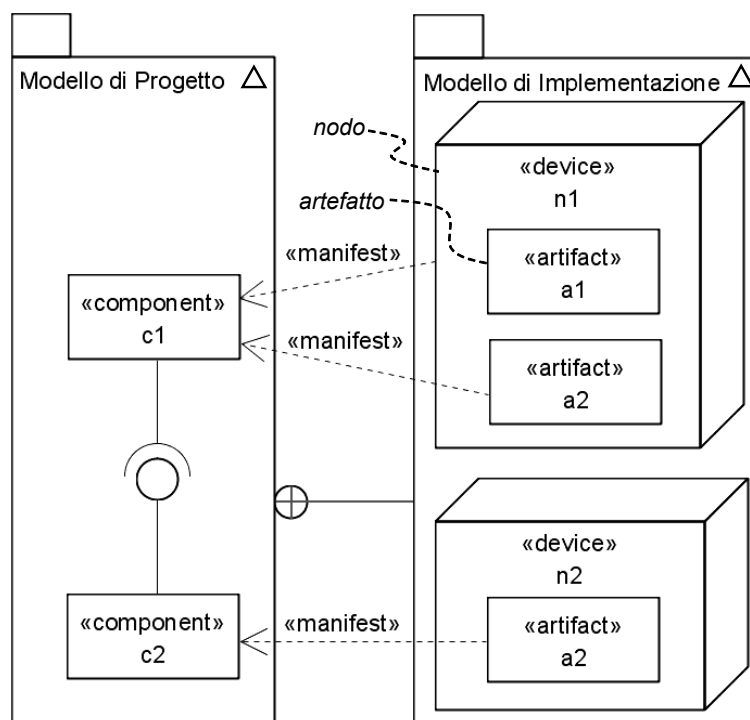
1. quando si vuole generare il codice direttamente dal modello;
2. se si sta realizzando uno sviluppo basato sul componente (component-based development) in modo tale da riusare componenti già disponibili: l'allocazione ai componenti di classi di progetto ed interfacce diviene un problema strategico.

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.

397

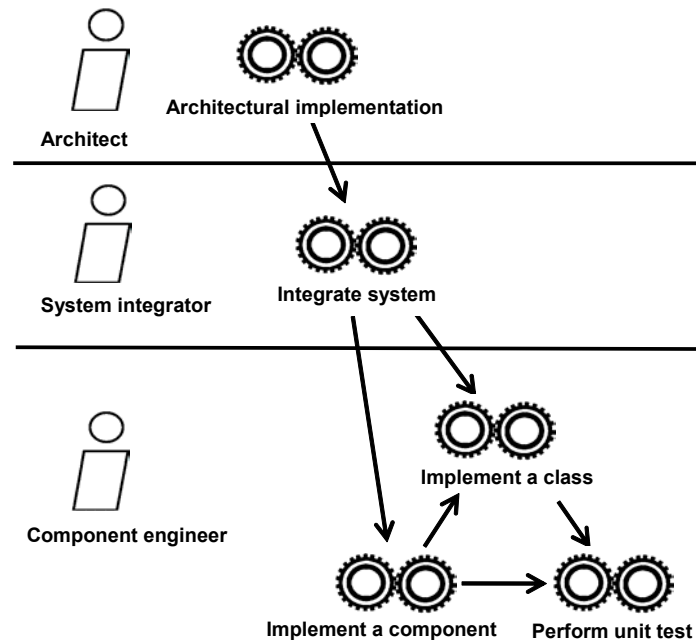


La relazione «manifest» tra artefatti e componenti indica che gli artefatti sono rappresentazioni fisiche dei componenti.

La figura seguente mostra le attività principali del workflow Implementazione.

398





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

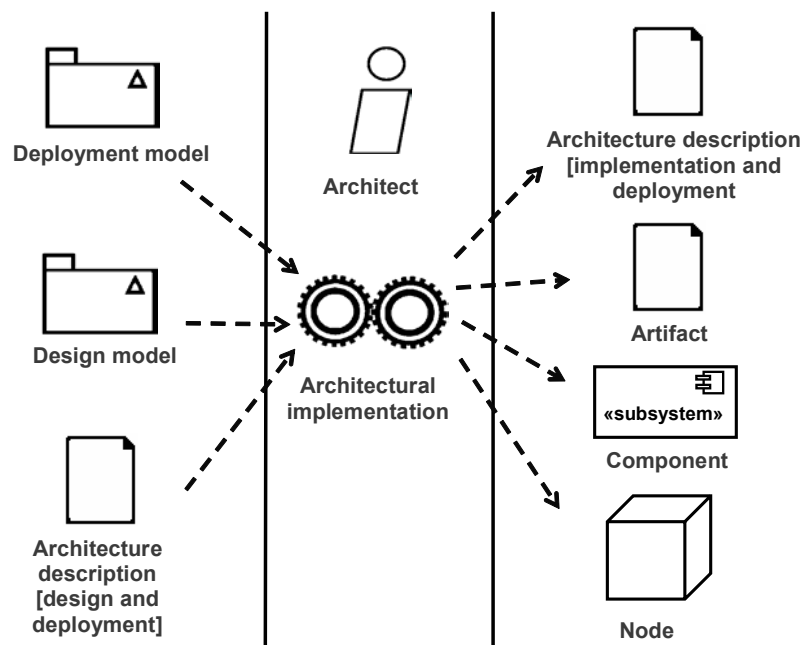
399

- 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.

400



## 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

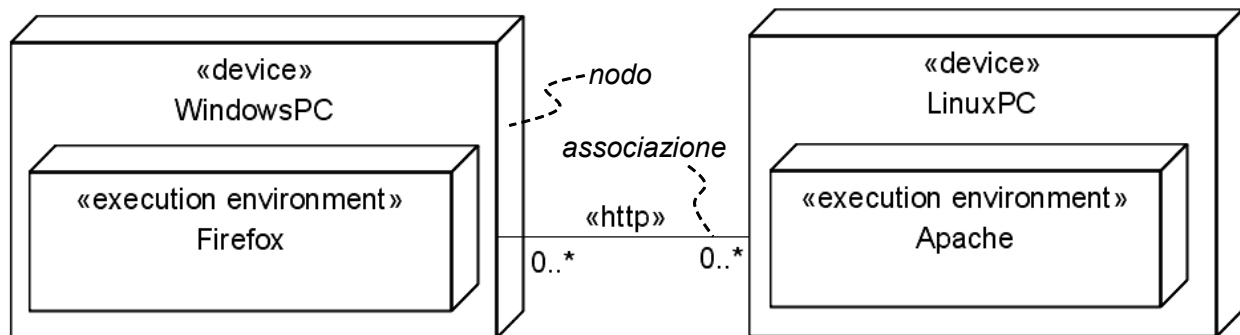
UML 2: “Un nodo rappresenta un tipo di risorsa computazionale su cui possono essere dislocati artefatti per l'esecuzione.”

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.

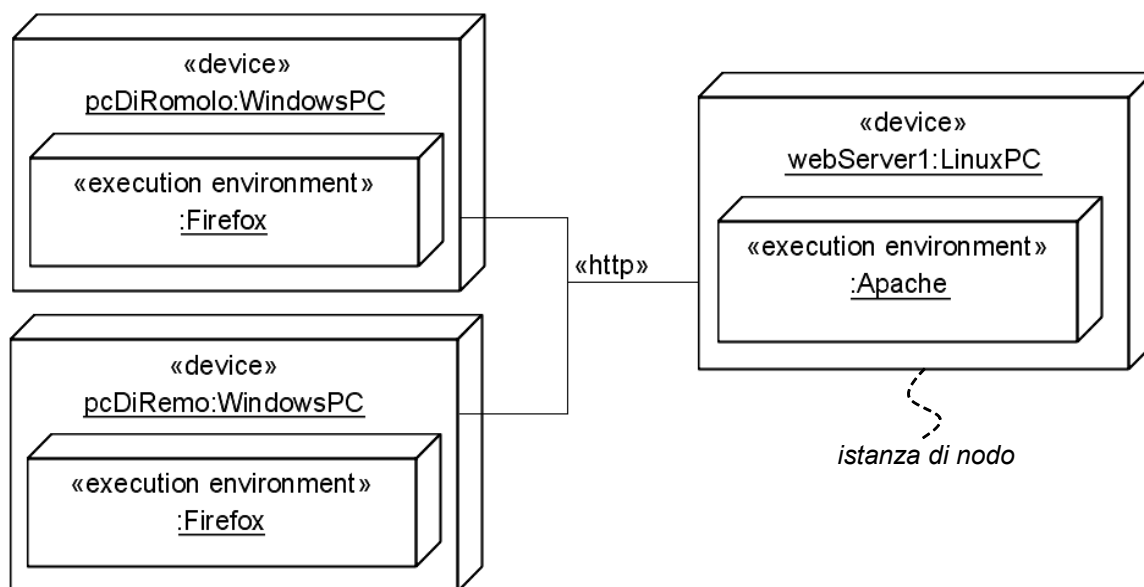
La figura seguente presenta un diagramma di dislocazione in forma descrittiva. Come è pratica comune, nella figura sono stati inclusi nello stesso diagramma sia il tipo di hardware che l'ambiente di esecuzione.



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.

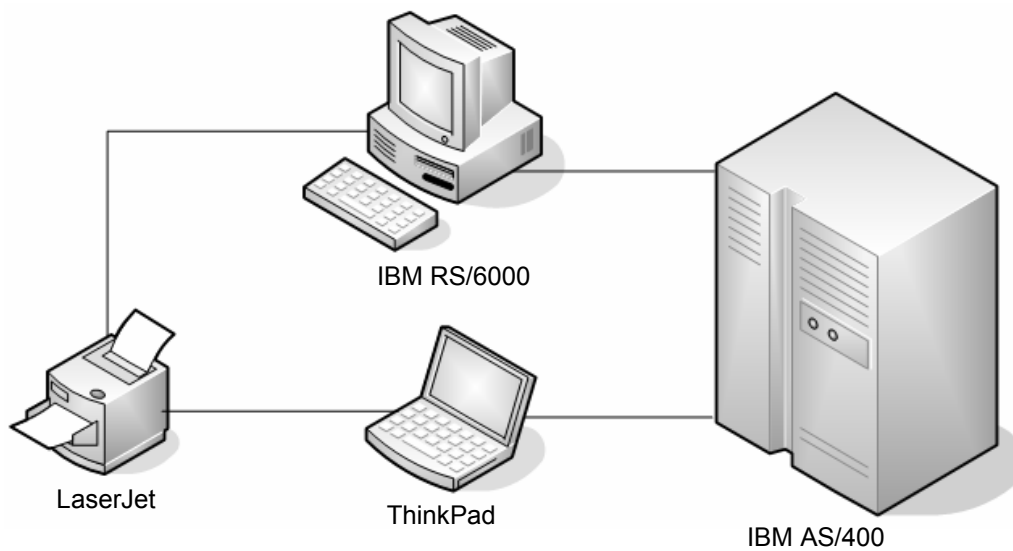
405



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.

406



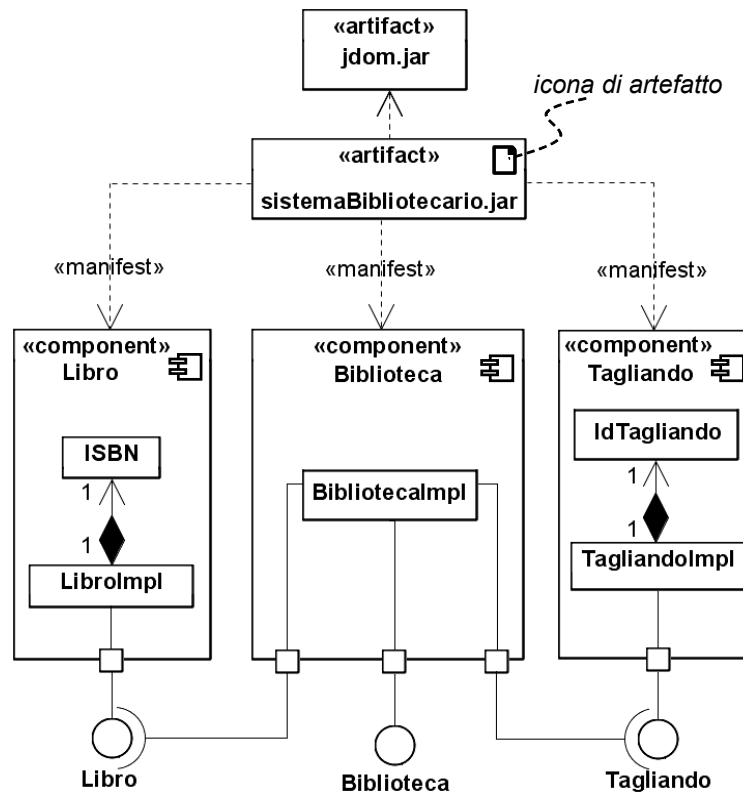
407

## 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.

408



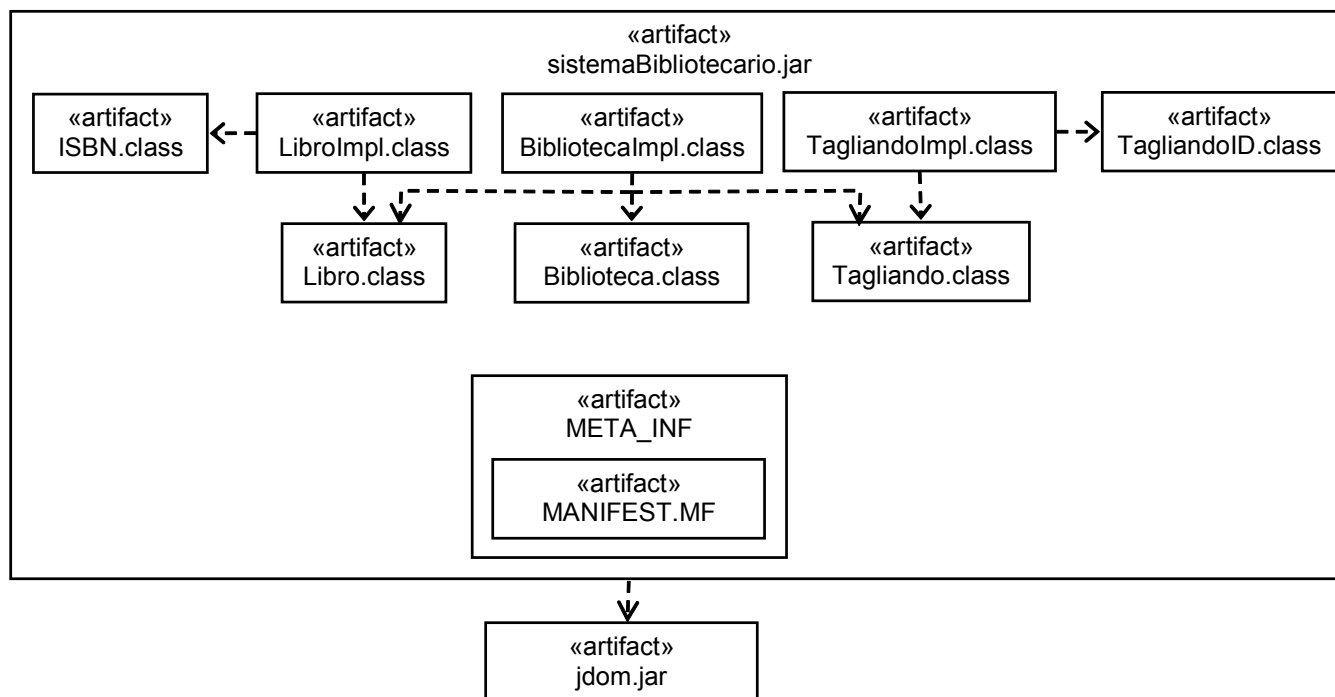
Un artefatto può fornire la manifestazione fisica di ogni tipo di elemento UML. Tipicamente, esso manifesta uno o più componenti. Nella figura, sistemaBibliotecario.jar manifesta tre

409

componenti: Libro, Biblioteca, Tagliando. Oltre al nome, un artefatto possiede nella sua specifica anche un nome di file che indica la locazione fisica dell'artefatto.

Per esempio, il nome del file potrebbe specificare un URL dove può essere recuperata la copia originale dell'artefatto.

Dopo aver compilato i file sorgenti per i componenti mostrati nella figura precedente, viene usato uno strumento per creare un file JAR da questi file compilati. Il risultato è mostrato nella figura seguente:



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:

411

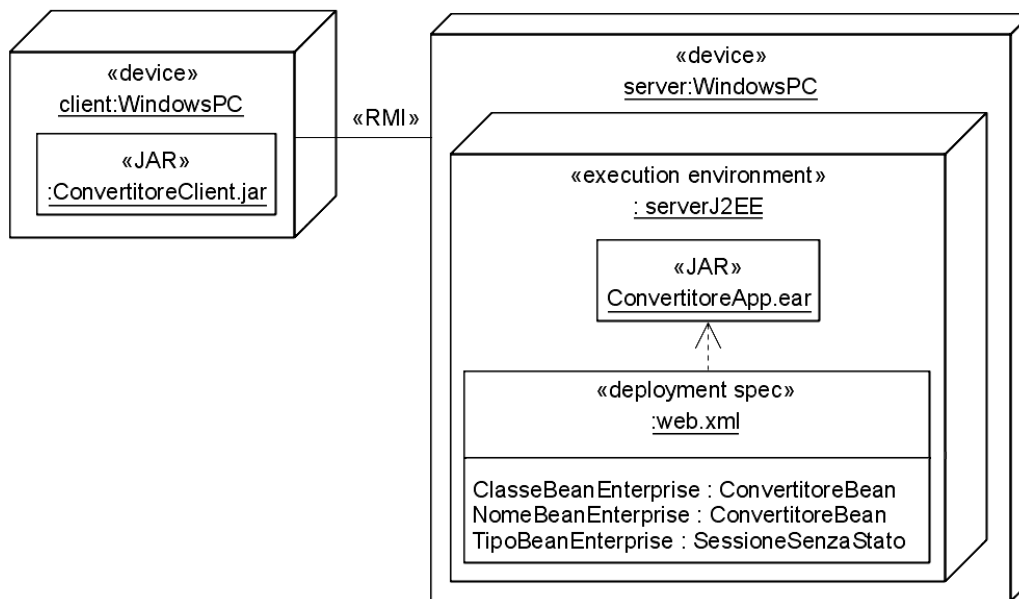
Stereotipi di Artefatto	Semantica
«file»	Un file fisico
«deployment spec»	Una specifica di dettagli di dislocazione
«document»	Un file generico che mantiene qualche informazione
«executable»	Un file eseguibile
«library»	Una libreria statica o dinamica
«script»	Uno script che può essere eseguito da un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

412

## Dislocazione

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



413

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.

414