

# Analisi e Progettazione del Software

UniShare

Davide Cozzi  
@dlcgold

Gabriele De Rosa  
@derogab

Federica Di Lauro  
@f\_dila

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Introduzione all’Ingegneria del software</b>	<b>4</b>
2.1	Modelli di Processo . . . . .	8
2.2	Modello UP . . . . .	16
2.2.1	Modello UP o RUP . . . . .	16
2.2.2	Processo Scrum . . . . .	21
<b>3</b>	<b>Casi d’uso</b>	<b>26</b>
3.1	Diagrammi dei casi d’uso . . . . .	36
<b>4</b>	<b>Contratti delle Operazioni</b>	<b>43</b>
4.1	Operazioni e UML . . . . .	46
<b>5</b>	<b>Modellazione di Dominio</b>	<b>48</b>
5.1	Associazioni . . . . .	53
5.1.1	Notazione per le Associazioni . . . . .	54
5.1.2	Aggregazione e composizione . . . . .	56
5.2	Attributi . . . . .	56
<b>6</b>	<b>Progettazione a Oggetti</b>	<b>62</b>
6.1	Diagrammi di Interazione di UML . . . . .	64
6.1.1	Notazione in UML . . . . .	67
6.1.2	Notazione dei Diagrammi di Sequenza . . . . .	69
6.1.3	Notazione nei Diagrammi di Comunicazione . . . . .	78
<b>7</b>	<b>Diagrammi delle Classi</b>	<b>83</b>
<b>8</b>	<b>Diagrammi di Attività in UML</b>	<b>98</b>
<b>9</b>	<b>Diagrammi di Macchina a Stati in UML</b>	<b>104</b>

<b>10 Architettura Logica</b>	<b>108</b>
10.1 Diagramma dei Package . . . . .	111
10.1.1 Principio di separazione Modello-Vista . . . . .	114
10.1.2 SSD e Operazioni di Sistema . . . . .	116
<b>11 GRASP</b>	<b>117</b>
<b>12 Applicare i design pattern GoF</b>	<b>132</b>
12.1 Basi per la Progettazione di un Framework . . . . .	144
<b>13 Refactoring e Code Smell</b>	<b>147</b>
13.1 refactoring . . . . .	147
13.2 Code Smell . . . . .	149

# Capitolo 1

## Introduzione

Questi appunti sono presi durante le lezioni in aula. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: <https://github.com/dlcgold/Appunti>. Grazie mille e buono studio!

## Capitolo 2

# Introduzione all'Ingegneria del software

Durante il corso di Analisi e Progettazione del software si analizzano i modelli e i principi per lo sviluppo di un software mantenibile, andando anche ad analizzare, in maniera sommaria, anche tutti gli strumenti di ingegneria del software, necessari per lo sviluppo ottimale di un software.

In questo corso studieremo in dettaglio i seguenti argomenti:

- introduzione all'ingegneria del software
- progettazione sistemi orientati ad oggetti
- modellazione a dominio
- UML e analisi dei casi d'uso
- design pattern
- sviluppo test-driven(cenni)
- code smell e refactoring(cenni)

Il software, è l'insieme delle componenti modificabili e non fisiche di un calcolatore, viene diviso in due categorie:

**generici** , per un ampio range di clienti, come ad esempio gli elaboratori di testi

**custom** , per un singolo cliente, come ad esempio i gestionali specifici di un'impresa

Questa differenza si sta sempre più assottigliando in quanto recentemente varie aziende stanno sviluppando un software generico, che viene poi adattato in base alle esigenze del singolo cliente, come ad esempio i software Oracle e Sas.

Nello sviluppo di software si utilizza spesso una base pre-esistente, infatti l'ingegneria del software si occupa di tutti gli aspetti per lo sviluppo del software, sfruttando di solito una base preesistente.

Di solito quando si sviluppa un software si presuppone che abbia una durata di alcuni anni, con la predisposizione al cambiamento e all'introduzione di nuove features infatti un software per essere utile deve essere continuamente cambiato.

Si hanno due tipologie di progetti:

1. **progetti di routine**, con soluzione di problemi e riuso di vecchio codice
2. **progetti innovativi**, con soluzioni nuovi

e solitamente l'ingegneria del software si occupa principalmente di progetti innovativi con specifiche del progetto variabili, con la presenza di cambiamenti continui e ovviamente non è uguale né similare con l'ingegneria tradizionale.

Un programmatore normalmente lavora da solo, su un programma completo con specifiche note mentre un ingegnere del software lavora in gruppo, progetta componenti e l'architettura e identifica requisiti e specifiche. Il costo del software spesso supera quello hardware

Nello sviluppo di un software si hanno le seguenti fasi di sviluppo:

- **analisi dei requisiti**, che indica cosa deve fare il sistema
- **progettazione**, progetto del sistema da implementare
- **sviluppo**, produzione del sistema software
- **convalida**, verifica dei requisiti del cliente
- **evoluzione**, evoluzione al cambiare di requisiti del cliente

Esistono dei sistemi software per l'automazione delle attività svolte nel progetto software, i cosiddetti **CASE** (Computer-Aided Software Engineering) che si dividono in:

**CASE di alto livello** per il supporto alle prime attività di processo come la raccolta requisiti e la progettazione

**CASE di basso livello** per il supporto alle ultime attività di processo come la programmazione, il debugging, testing e reverse engineering

Nella figure X1 e X2 si hanno delle risposte alle comuni domande sull'ingegneria del software e le caratteristiche di un ottimo software, necessario per mantenerlo facilmente modificabile nel tempo.

Nello sviluppo di un software, qualsiasi tipologia esso sia, prevede i seguenti aspetti in comune:

**specifiche del software** vengono definite le specifiche e analizzato in dettaglio il comportamento e le funzionalità richieste da sviluppare

**sviluppo del software** viene effettivamente implementato e progettato il codice, attraverso i tools di sviluppo, rispettando le specifiche previste nella fase precedente.

**convalida del software** viene testato il software sviluppato nella fase precedente, al f

**evoluzione del software** viene mantenuto ed evoluto il software, al fine di riflettere i cambiamenti e delle funzionalità richieste dal cliente; è la fase più critica e costosa, dato che a volte la manutenzione richiede di sistemare e correggere gli errori e/o il cattivo design del progetto.

Il software deve essere accettato dagli utenti per i quali è stato sviluppato per cui deve essere comprensibile, usabile e compatibile con altri sistemi, oltre a dover essere mantenibile, affidabile ma soprattutto efficiente.

Esiste un codice etico, **ACM/IEEE**, con otto principi legati al comportamento degli ingegneri del software ed è fondamentale seguirlo dato che fornisce informazioni su come risolvere i problemi etici collegati a tutti i progetti software, come le informazioni da fornire all'esterno ed altri aspetti.

Un **sistema** è una collezione significativa di componenti interrelati che lavorano assieme per realizzare un obiettivo comune, quindi include software e parti meccaniche o elettriche; inoltre si ha che i vari componenti possono dipendere da altri componenti e che le proprietà e il comportamento dei vari componenti sono intrinsecamente correlati, quindi si hanno due macro categorie:

1. **sistemi tecnico-informatici**, in cui sono inclusi hardware e software ma non gli operatori e i processi operazionali, dove sono presenti le **proprietà emergenti**, dipendenti dalle sue componenti e le relazioni tra esse, misurabili soltanto sul sistema finale.

Inoltre i sistemi tecnici sono non-deterministici, in quanto dato lo stesso

input non è detto che restituisca lo stesso output, in quanto il risultato è spesso dipendente dal comportamento degli operatori umani.

2. **sistemi socio-tecnici**, comprendente uno o più sistemi tecnici, assieme ai processi operazionali e agli operatori, sono fortemente condizionati da politiche aziendali e regole.

I sistemi Socio-tecnici sono sistemi pensati per raggiungere obiettivi aziendali o organizzativi e bisogna comprendere a fondo l'ambiente organizzativo nel quale un certo sistema è usato.

Vediamo qualche proprietà emergente:

- **volume**: il volume di un sistema varia a seconda di come sono disposti e collegati i componenti che lo formano.
- **affidabilità**: l'affidabilità del sistema dipende dall'affidabilità dei suoi componenti, ma interazioni impreviste possono produrre nuovi fallimenti e quindi influenzare l'affidabilità dell'intero sistema
- **protezione**: la protezione del sistema è una proprietà complessa che non può essere facilmente misurata, in quanto in futuro potrebbero essere inventati nuove modalità di accesso e/o attacco che rendono meno protetto il software.
- **riparabilità**: questa proprietà riflette la facilità con cui è possibile correggere un problema e ciò dipende dalla possibilità di diagnosticare il problema e soprattutto di accedere, modificare o sostituire i componenti difettosi.
- **usabilità**: questa proprietà mostra la facilità d'uso del sistema e dipende dai componenti del sistema tecnico, dai suoi operatori e dal suo ambiente operativo.

Oltre a questi sistemi software standard, sono presenti i **sistemi critici**, in cui fondamentalmente non sono ammessi errori e malfunzionamenti altrimenti causano dei gravi problemi se non morti, per questo le specifiche avvengono in linguaggio il più possibile formale possibile e usano un modello di sviluppo a cascata, inefficiente per gli altri sistemi software.

Abbiamo tre esempi di sistemi critici:

1. **sistemi safety-critical** dove i fallimenti comportano rischi ambientali o perdite di vite umane, come ad esempio un sistema di controllo per un impianto chimico.

2. **sistemi mission-critical** dove i fallimenti possono causare il fallimento di attività a obiettivi diretti, come un sistema di navigazione di un veicolo spaziale.
3. **sistemi business-critical** dove i fallimenti possono risultare in perdite di denaro sostenute, come ad esempio un sistema bancario.

I fallimenti di un software, sia che fossero piccoli che grandi, possono essere di diversi tipi:

- **fallimenti hardware**, errori di progetto, produzione o "consumo"
- **fallimenti software**, errori di specifica, progetto, implementazione
- **errori operativi**, errori commessi da operatori umani (forse una delle maggiori cause di fallimenti)

Nei sistemi critici, generalmente la fidatezza è la più importante proprietà del sistema in quanto si ha un alto costo dei fallimenti ed esso riflette il livello di confidenza che l'utente ha verso il sistema, cosa leggermente diversa dall'utilità, ossia la sicurezza di non avere problemi e/o intrusioni da parte di persone esterne.

Il costo dei fallimenti nei sistemi critici è così alto che i metodi di ingegneria del software non sono cost-effective per cui si utilizza un modello a cascata, in quanto si devono fare prima tutte le analisi, prima di poter implementare e testare il sistema e soprattutto si spende un sacco di soldi per il testing e le analisi per convalidare la fidatezza necessaria da raggiungere.

## 2.1 Modelli di Processo

Il modello di un processo software è una rappresentazione semplificata del processo, basata su un aspetto specifico:

- **modello a flusso di lavoro (Workflow)** per una sequenza di attività
- **modello a flusso di dati (Data-flow)** per il flusso delle informazioni
- **modello ruolo/azione (role/action)** per decidere i vari ruoli

si hanno poi tre modelli generici:

1. **modello a cascata (waterfall)**: modello in cui le diverse fasi, come si nota nella figura Figura 2.1, avvengono in sequenza, dove i problemi e/o mancanze di interpretazioni si notano verso la fine dello sviluppo, nella

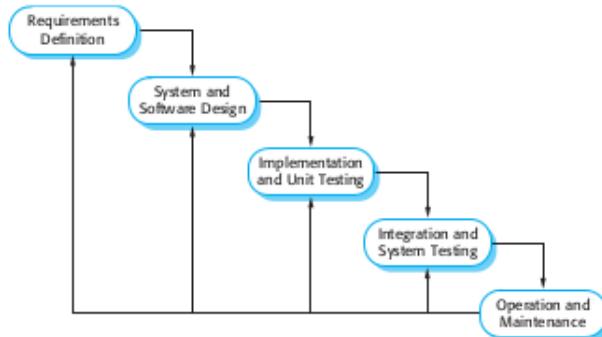


Figura 2.1: Modello a cascata fig:wf

fase di test e/o consegna, e ciò comporta una modifica di una o più fasi precedenti dello sviluppo, con notevoli aggravi di costi e tempestiche. È stato il primo modello di sviluppo, ma a partire dagli anni 90 si comprese che la maggioranza dei fallimenti dei software era da imputare al fatto di cercare di definire tutti i requisiti prima di sviluppare, cosa che è difficile da fare in maniera efficace in sistemi variabili nel tempo, come quasi tutti i sistemi software da sviluppare e/o già sviluppati. Vediamo una spiegazione più dettagliata:

- **Requirements analysis and definition:** si stabiliscono servizi, limiti e fini del software consultandosi col cliente
- **System and software design:** si stabilisce un'architettura di sistema complessiva stabilendo l'hardware e il software necessario
- **Implementation and unit testing:** si sviluppano le unità di test e si verifica che ogni parte del software risponda alle specifiche richieste presa singolarmente
- **Integration and system testing:** si testa il software nella sua interezza, si verifica che il software rispetti ogni requisito e infine si consegna il prodotto al cliente
- **Operation and maintenance:** una volta che il prodotto è stato consegnato si provvede a mantenerlo, risolvendo eventuali errori e aggiungendo eventuali funzionalità, soddisfacendo eventuali nuovi requisiti. Potrebbe non essere uno step presente ogni volta

2. **modello iterativo (Iterative development):** modello in cui le fasi di sviluppo avvengono nel corso del tempo, senza effettuare prima tutta la progettazione e poi l'implementazione, sviluppando versioni sempre più definite del software, portando subito in risalto i problemi e/o le

mancate interpretazioni del sistema da sviluppare.

Prevede diverse implementazioni di questo modello, come ad esempio l'UP, lo Scrum e la modellazione agile, in cui il concetto di fondo è quello di conoscere, sviluppare e progettare il sistema a passi successivi, al fine di avere ad ogni passo un sottoinsieme del sistema già funzionante e questo permette di poter reagire ai cambiamenti dei requisiti e permettere una manutenzione mantenibile nel tempo.

**3. ingegneria del Software basata sui componenti (CBSE)**

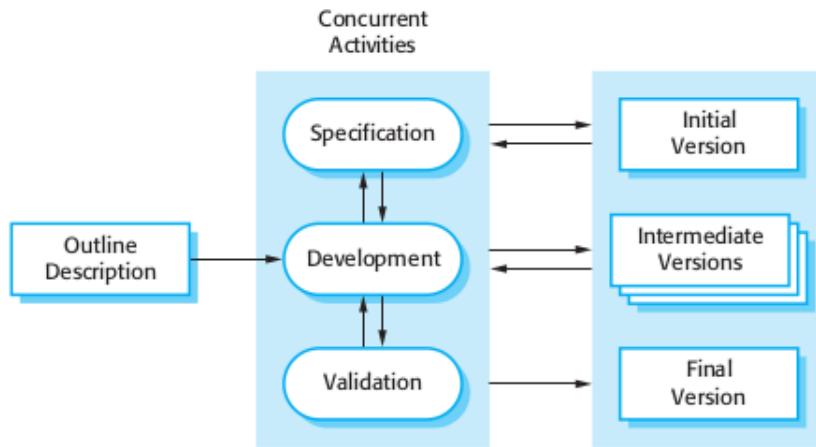


Figura 2.2: sviluppo incrementale `fig:inc`

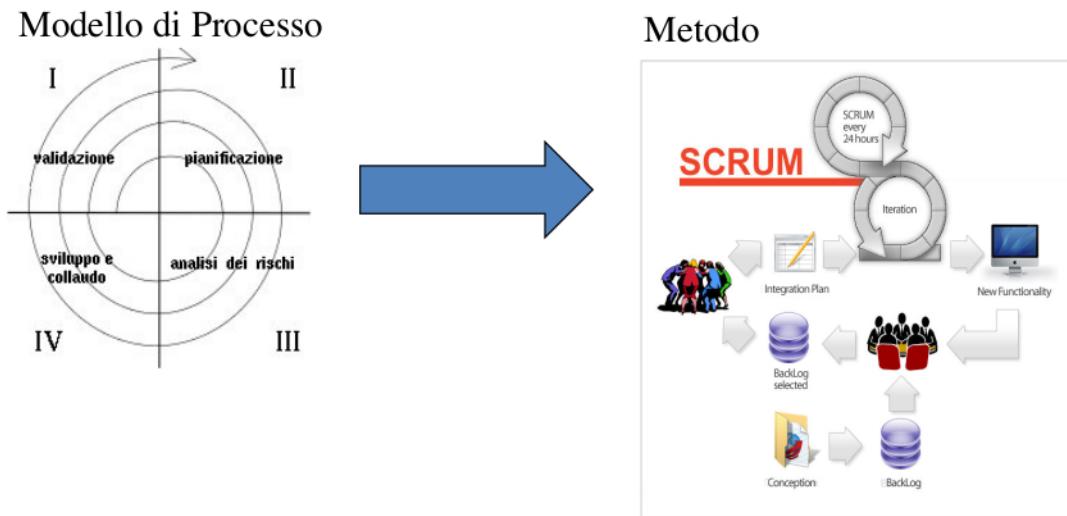
### Sviluppo incrementale

Lo sviluppo incrementale si basa sull'idea di sviluppare un'implementazione iniziale del software che viene revisionata dai futuri utenti dello stesso. Si procede nella stessa maniera con ulteriori versioni fino a giungere ad un'implementazione finale. Quindi sviluppo, indicato in figura Figura 2.2 e consegna sono strutturati in una sequenza di incrementi, ognuno delle quali corrisponde a parte delle funzionalità richieste, con ovviamente un ordine di priorità dato dal cliente (le prime parti conterranno le funzionalità principali del software). Lo sviluppo incrementale ha 3 vantaggi principali rispetto al modello a cascata:

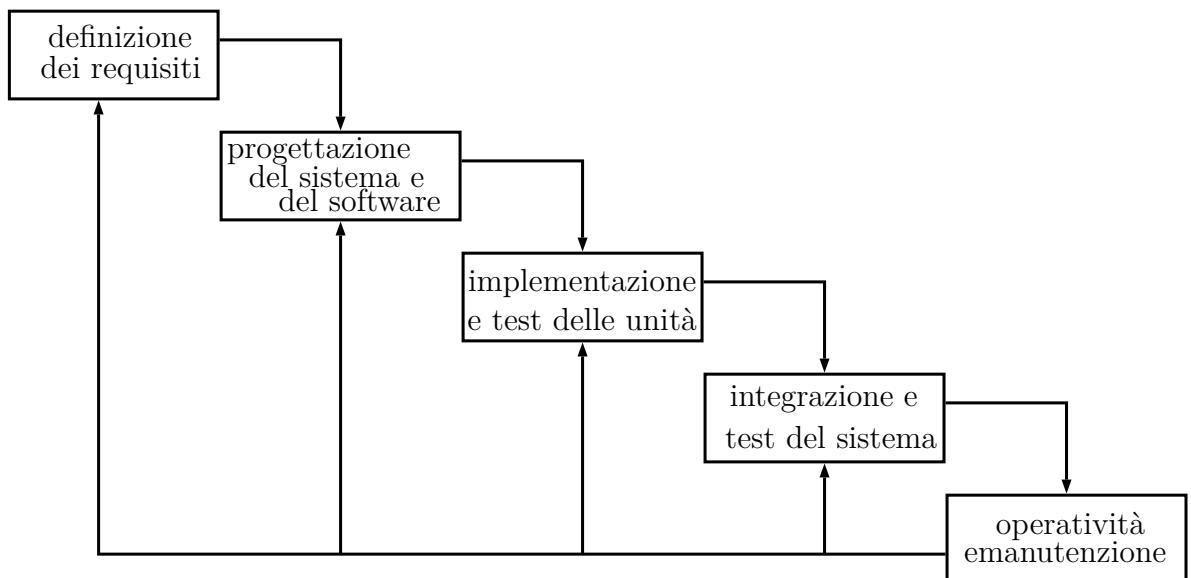
1. si riduce il costo relativo al cambio di specifiche del cliente, riducendo anche la quantità di analisi e documentazione che andrebbe rielaborata
2. si hanno feedback costanti sullo sviluppo grazie alla continua prova diretta delle varie implementazioni
3. si può fornire gradualmente al cliente un software funzionante anche se privo di alcune features, di modo che possa usare parte del software ben prima del suo completo sviluppo

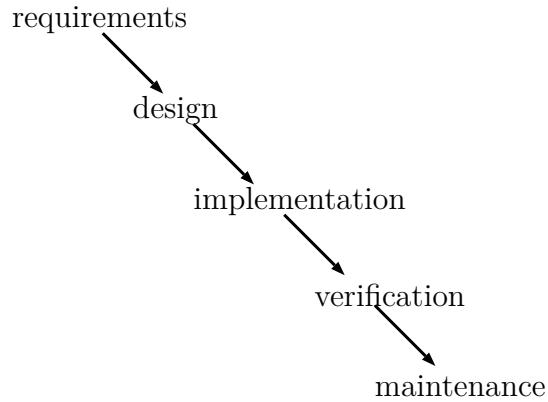
Inoltre i rischi di fallimento si abbassano e i servizi a priorità più alta sono testati più a fondo. Del resto questo modello può essere rallentato facilmente da problemi burocratici. Inoltre il processo di sviluppo è più difficile da controllare da parte di un manager e il continuo aggiungere features può danneggiare l'architettura generale del software **pagina 34**

I metodi di ingegneria del software sono l'implementazione dei seguenti modelli generici, e nella maggioranza si utilizza un modello iterativo, al fine di rendere incrementale lo sviluppo e ridurre il rischio di fallimento del progetto.



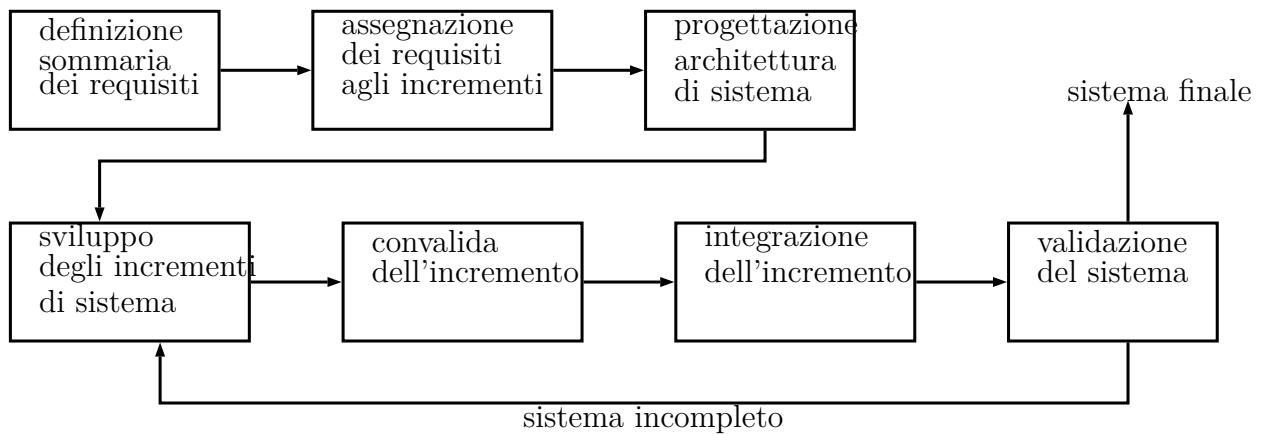
si ha il seguente feedback del modello a cascata:





Invece di rilasciare il sistema in una singola consegna, sviluppo e consegna sono strutturati in una sequenza di incrementi, ognuno dei quali corrispondenti a parte delle funzionalità richieste. Questa è la **consegna incrementale**. requisiti utente sono ordinati per priorità, i requisiti ad alta priorità sono inclusi nei primi incrementi.

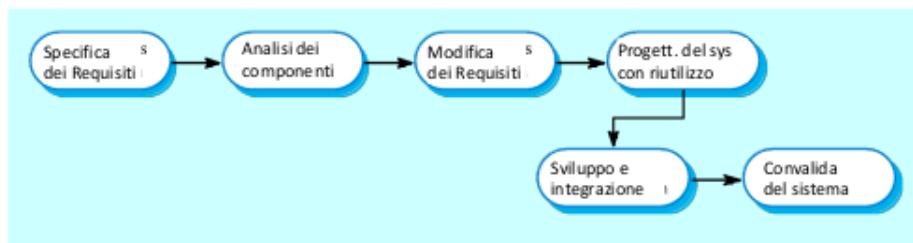
Una volta che lo sviluppo di un incremento inizia, i requisiti sono congegliati; invece possono evolvere i requisiti per gli incrementi successivi:



Con la consegna incrementale si hanno i seguenti vantaggi:

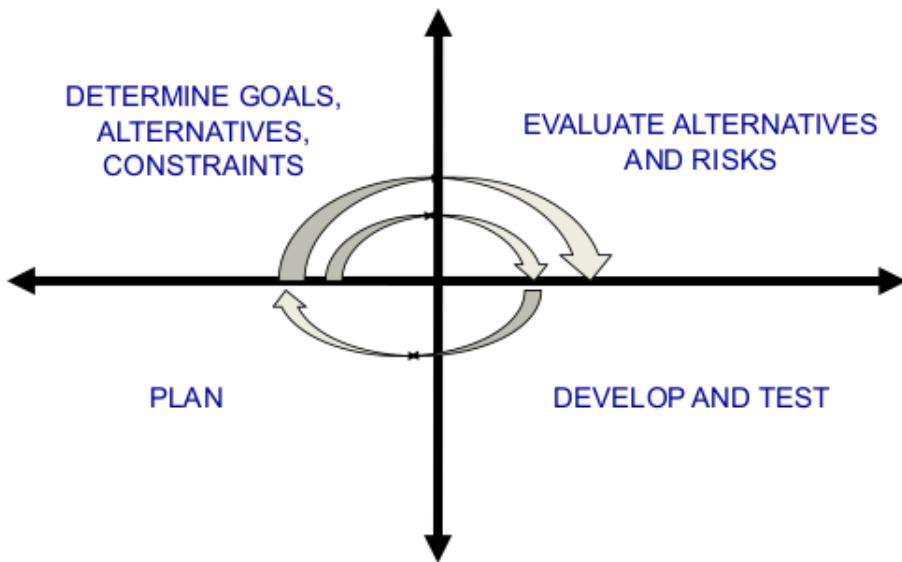
- funzioni utili per il cliente possono essere rilasciate ad ogni incremento, quindi alcune funzionalità di sistema sono disponibili sin dai primi incrementi
- i primi incrementi rappresentano dei prototipi che supportano la scoperta dei requisiti per i successivi incrementi
- i rischi di fallimento si abbassano
- i servizi a priorità più alta tendono ad essere collaudati più a fondo

Se si ha che i requisiti vengono scoperti attraverso lo sviluppo si ha lo **sviluppo evoluzionario**. Si usano prototipi che poi non verranno utilizzati nel corso del progetto vero e proprio

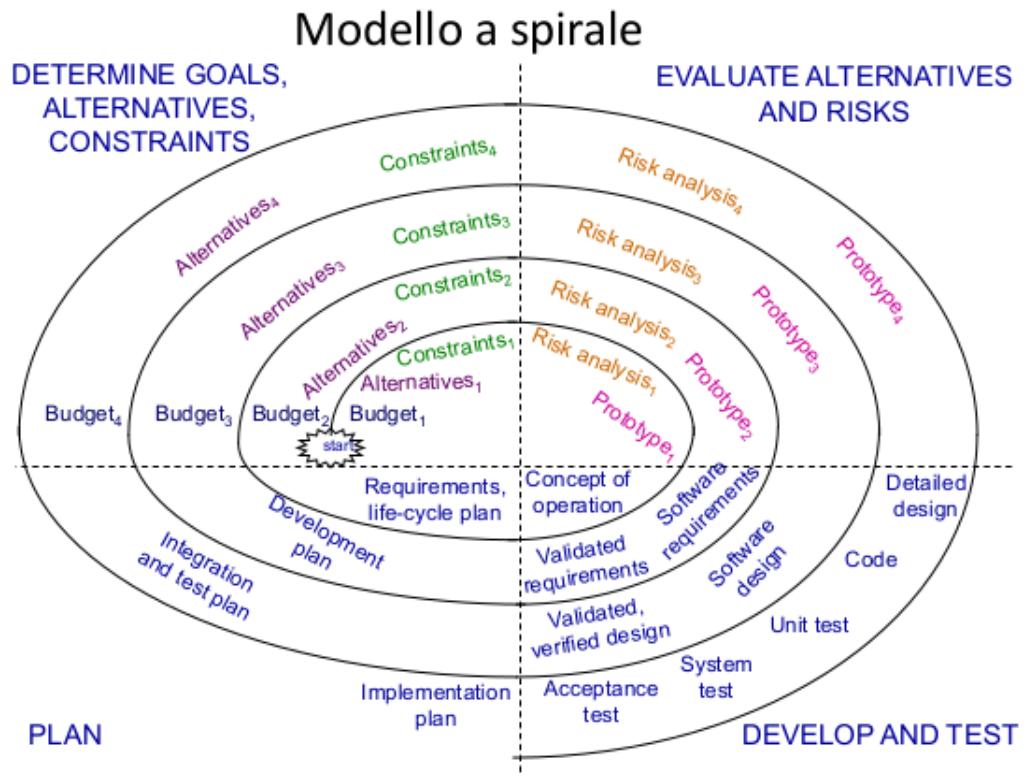


Si ha ingegneria del software basata su componenti se si hanno già librerie su cui lavorare.

Passiamo al primo modello iterativo, quello di B. Boehm.



si parte dal quadrante in alto a sinistra e si arriva al prodotto finale mediante una serie di iterazioni. Si sceglie ovviamente la soluzione più sicura. I vari passaggi sono qui rappresentati:



## 2.2 Modello UP

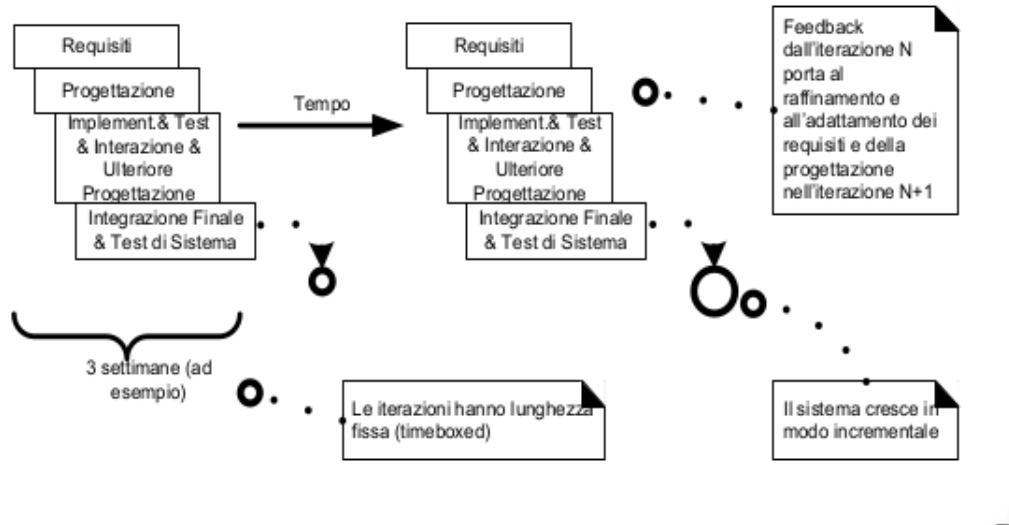
Il metodo di modellazione UP (Unified Process), conosciuto anche come RUP (Rational Unified Process), è un processo iterativo molto diffuso per lo sviluppo software, in cui sono presenti degli elementi tratti da altri metodi, come Extreme Programming, Scrum e così via.

Lo sviluppo iterativo Si hanno i modelli agili, che si basano sulla  **consegna incrementale**. Si hanno iterazioni brevi e timeboxed, con un raffinamento evolutivo di piani, dei requisiti e del progetto. Questi metodi favoriscono la collaborazione nei gruppi e la riduzione dei costi.

### 2.2.1 Modello UP o RUP

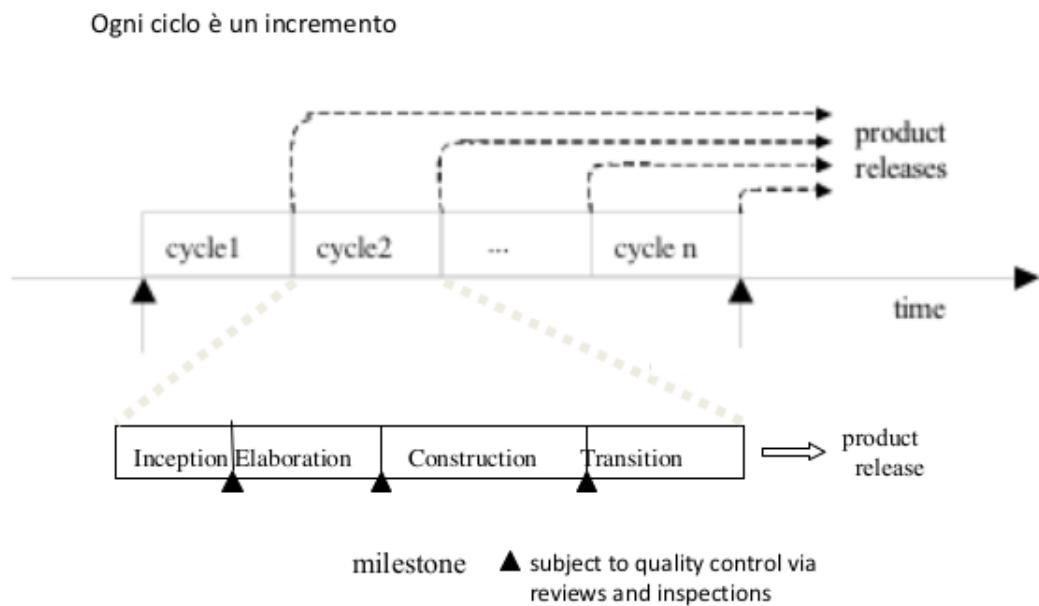
Il metodo UP, *unified process*, conosciuto anche come RUP, *Rational Unified Process*, usa la notazione UML, *unified modeling language* ed è uno dei più importanti modelli agili. È un processo iterativo e incrementale. Si basa sulla suddivisione di un grande processo in iterazioni controllate. Si hanno

passi piccoli, timeboxed da 2 a 6 settimane, feedback rapido e adattamento, di requisiti, modelli, stime di sviluppo e costi e priorità.



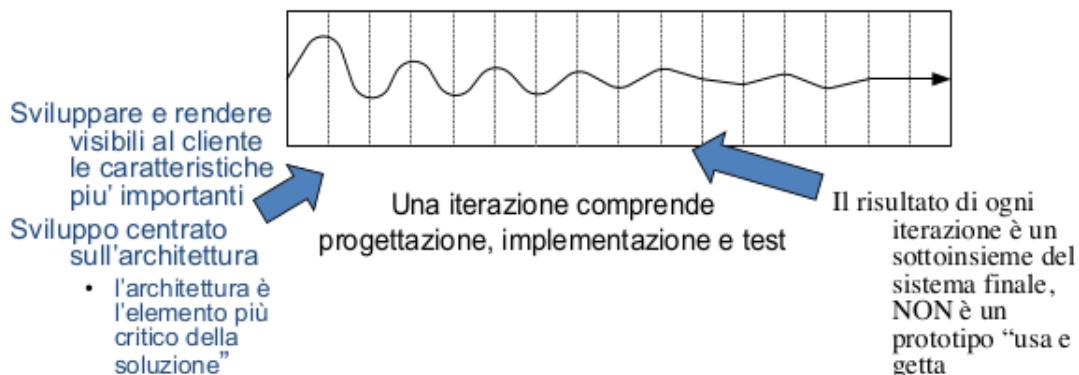
Si hanno 4 fasi nel modello RUP:

1. **avviamento:** dove viene stabilita la bussines rationale del progetto e stabiliti gli obiettivi, stime dei costi e dei tempi. Si ha uno studio di fattibilità, *Life-Cycle Objective Milestone*
2. **elaborazione:** dove si raccolgono i requisiti in modo più dettagliato, si procede con l'analisi ad alto livello per stabilire l'architettura di base e vengono analizzati i rischi principali. Si hanno stime più affidabili: *Life-Cycle Architecture Milestone*
3. **costruzione** che consiste di molte iterazioni, e ad ogni iterazione viene costruita una parte del sistema che soddisfa un sottoinsieme di requisiti. Viene effettuato il testing e l'integrazione. Si ha la preparazione al rilascio: *Initial Operational Capability Milestone*
4. **transizione:** dove vengono affrontati tutti gli aspetti legati al fine-tuning delle funzionalità, prestazioni, qualità, beta-testing, ottimizzazione, formazione degli utenti,... È la *Product Release Milestone*

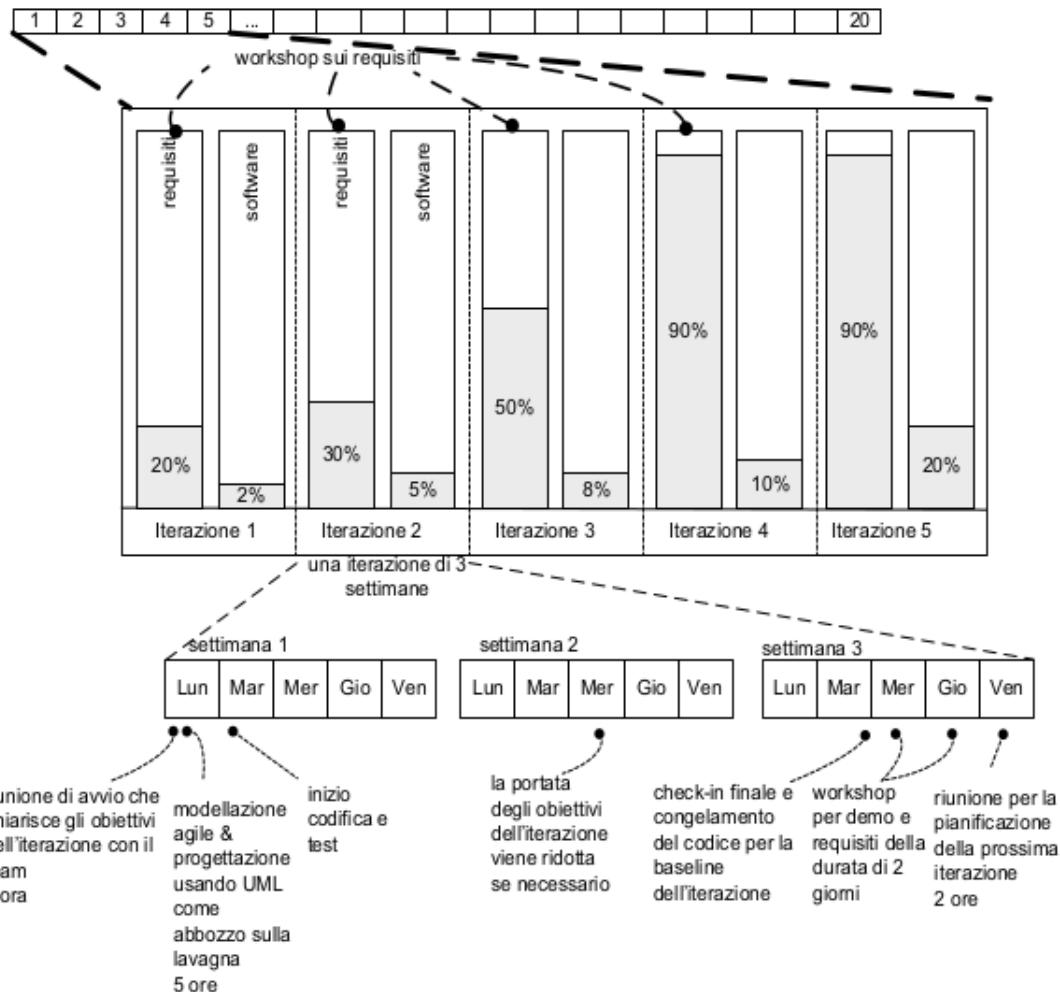


Si hanno i seguenti vantaggi dello sviluppo iterativo:

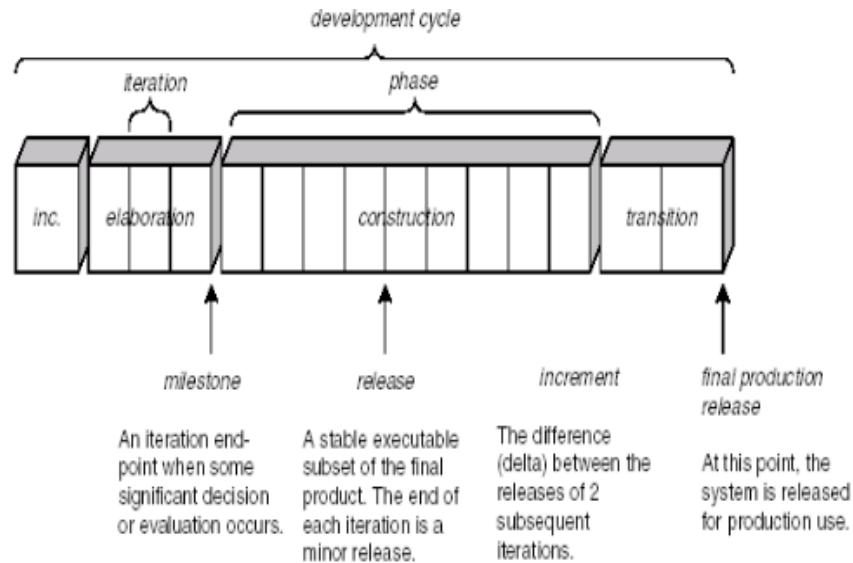
- minore chance di fallimento del progetto
- riduzione precoce dei rischi
- progresso visibile
- feedback precoce e coinvolgimento dell'utente
- "abbracciare il cambiamento"



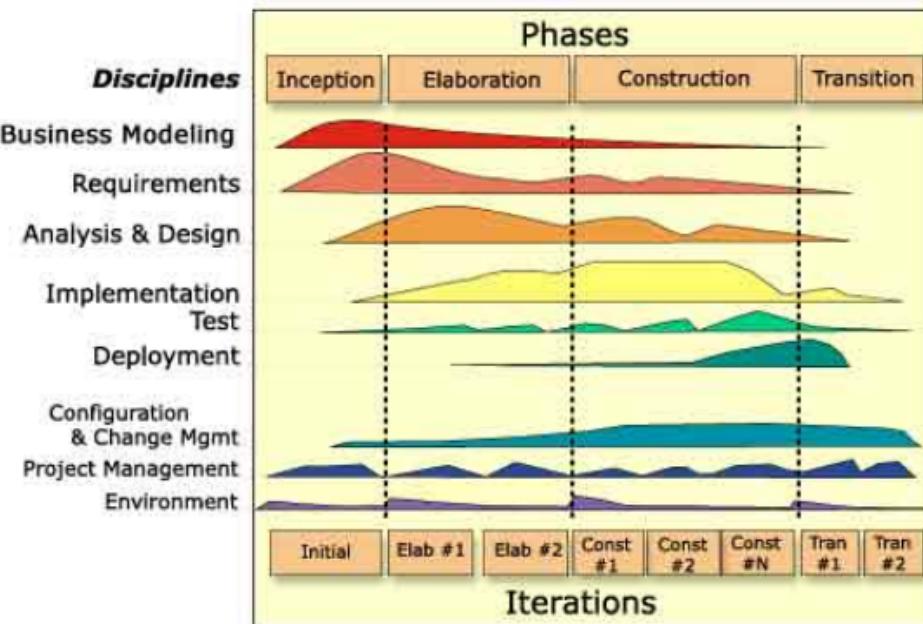
Vediamo un esempio con 5 iterazioni:



vediamo un'altra rappresentazione del ciclo di sviluppo:



vediamo anche un'immagine per rappresentare l'organizzazione del processo, con fasi, iterazioni e "discipline":

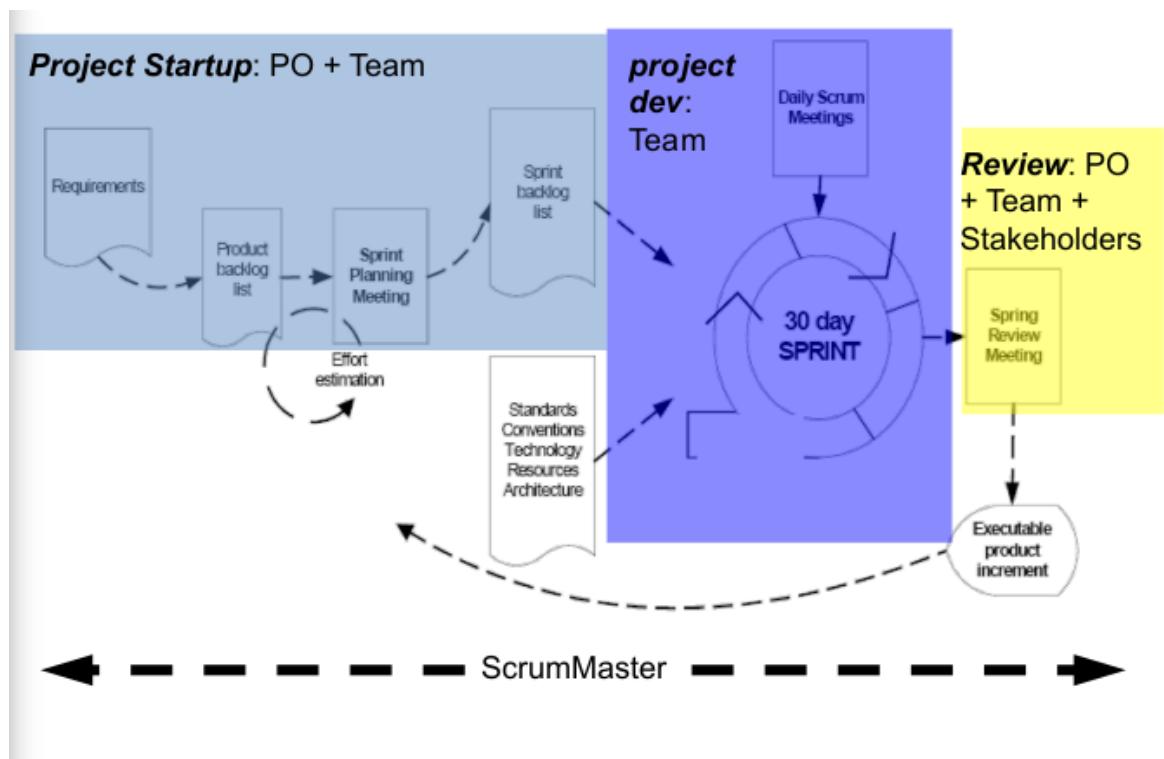


tra le discipline troviamo analisi e progettazione Si hanno le seguenti caratteristiche principali per l'UP:

- è iterativo e incrementale
- si enfasi sul modello invece che sul linguaggio naturale
- è centrato sull'architettura

### 2.2.2 Processo Scrum

Iniziamo con un'immagine che rappresenta questo metodo agile:



È un processo iterativo basato sul controllo dello stato di avanzamento. Si hanno i seguenti principi fondamentali:

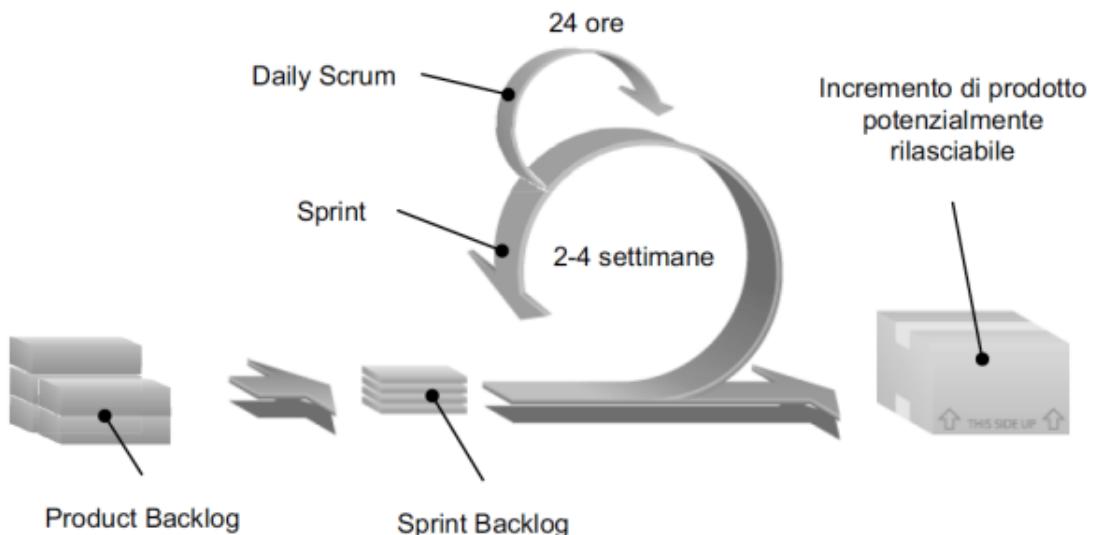
- **visibilità:** gli aspetti significativi del processo di sviluppo devono essere visibili a tutti e tutti gli aspetti devono essere chiari
- **ispezione:** poter ispezionare frequentemente il lavoro fatto per verificare che si sta procedendo verso gli obiettivi posti

- **adattamento**, se si sta deviando dagli obiettivi, occorre un adattamento per minimizzare altre deviazioni. Deve essere svolto nel minor tempo possibile in caso di necessità

Si hanno i seguenti ruoli:

- **product owner** che si occupa della definizione delle caratteristiche del progetto da sviluppare (1 sola persona, e.g. committente, o altri...); gestisce il Product Backlog. Lavora costantemente col team
- **team**: dedicato allo sviluppo e rilascio del prodotto attraverso incrementi successivi (da 3 a 7/8 membri)
- **scrum master**: responsabile che lo SCRUM venga applicato correttamente. Non è un project manager, fa da intermezzo tra team e product owner, collabora col team etc

Ecco un'immagine che rappresenta lo sviluppo con Scrum:



SI hanno quindi:

- releases brevi con sottoinsiemi consegnati velocemente
- in ogni istante si hanno test eseguibili, non si ha logica duplicata e si ha un numero minimo di features
- si ha *refactoring* con miglioramento continuo

- si un testo continuo e automatico, *testing first*
- si hanno due sviluppatori che lavorano insieme, *pair programming*
- il codice è proprietà del team e non del singolo dev, *collective ownership*
- si hanno check-in frequenti e integrazioni continue, *continuous integration*
- il cliente lavora col team

I **requisiti** sono una descrizione dei servizi del sistema e dei suoi vincoli operativi. Si hanno due tipi di requisiti:

- **requisiti utenti:** affermazioni in linguaggio naturale, corredate da tavole e diagrammi, riguardanti i servizi che il sistema offre ed i vincoli operazionali. Sono scritti per i clienti e sono da loro comprensibili anche se non hanno conoscenze tecniche dettagliate.
- **requisiti di sistema:** un documento strutturato che definisce in modo dettagliato le funzioni del sistema, i servizi ed i vincoli operazionali. Definisce cosa deve essere implementato, quindi può essere parte del contratto tra acquirente e sviluppatore. È la base del progetto della soluzione e può essere illustrato utilizzando i **modelli di sistema**.

i requisiti possono avere 3 problemi:

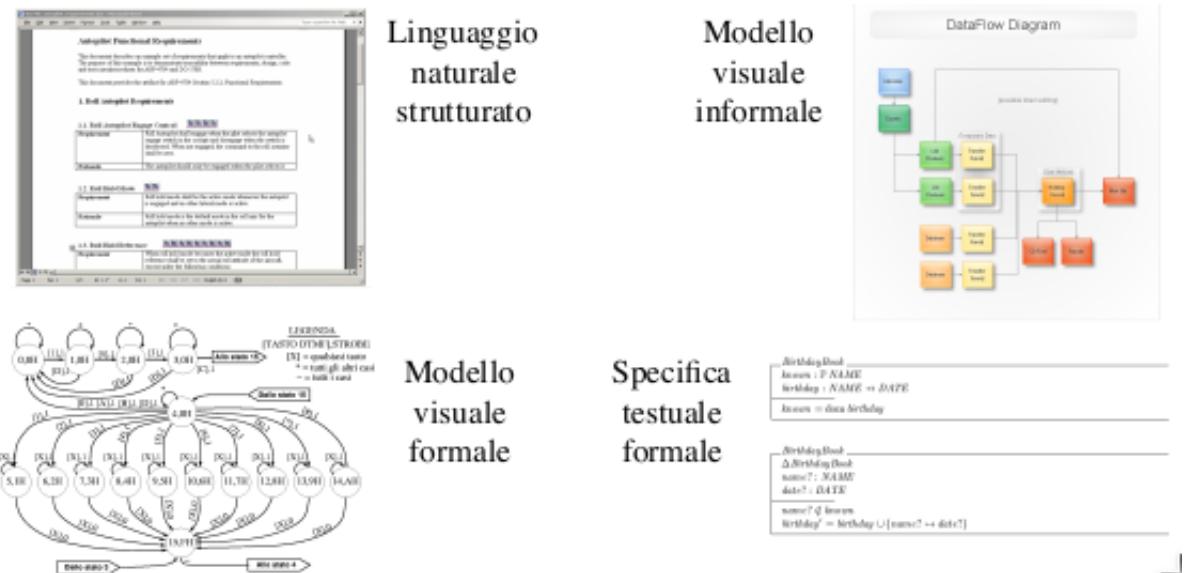
- **ambiguità:** il sistema fornisce visualizzazioni appropriate per leggere i documenti
- **incompletezza:** i requisiti devono descrivere tutti i servizi forniti dal sistema (anche se in realtà tutto ciò è impossibile, per questo esistono i cambiamenti)
- **inconsistenza:** le descrizioni non devono contenere conflitti o contraddizioni (anche questa cosa è difficilissima da ottenere)

SI hanno anche i **requisiti funzionali** servizi che il sistema deve (o non deve) fornire. Si hanno anche i *requisiti non funzionali*, come vincoli sui servizi temporali, standard, sull'usabilità etc... I requisiti non funzionali possono essere più critici dei requisiti funzionali: **se non sono soddisfatti il sistema è spesso inutilizzabile**. Si hanno alcuni requisiti non funzionali:

- **prodotto:** specificano che il prodotto deve comportarsi in un modo particolare esempi: velocità di esecuzione, affidabilità.  
*L'interfaccia utente sarà implementata con HTML semplice senza frames e applets*
- **organizzativi:** conseguenza di politiche e procedure dell'organizzazione del cliente e dello sviluppatore. esempi: linguaggio di programmazione, metodo di sviluppo.  
*Il processo di sviluppo e la relativa documentazione sarà conforme alle norme XYZCo-SP-STAN-95*
- **esterni:** provengono da fattori esterni al sistema e al processo di sviluppo. esempi: interoperabilità, leggi e norme.  
*Il sistema non deve rilevare agli operatori nessuna informazione personale sui clienti oltre al nome e al numero di riferimento*

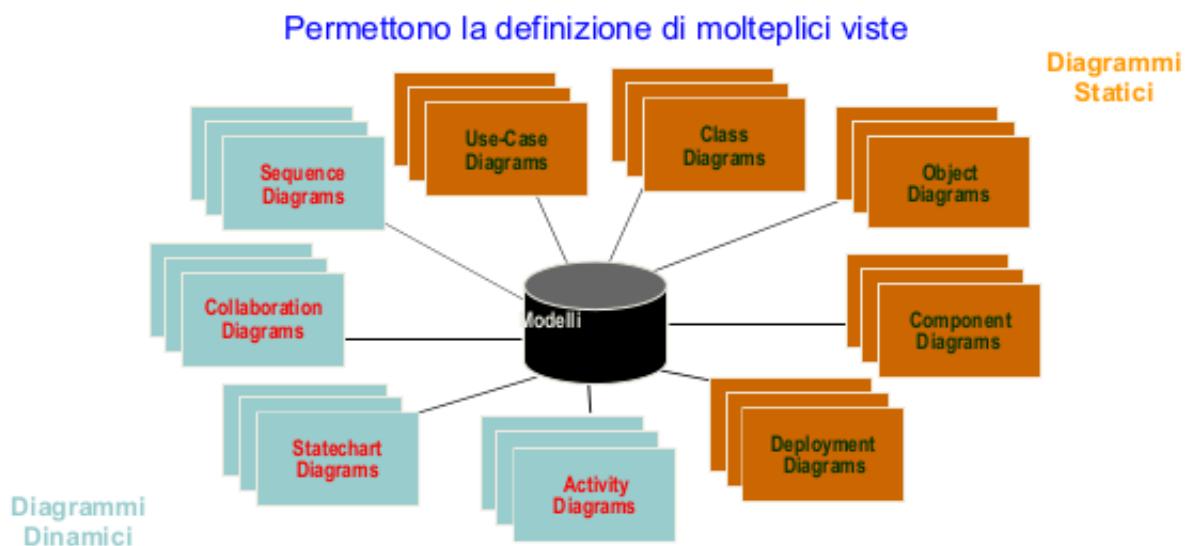


Tutto ciò perché il linguaggio naturale presenta mancanza di chiarezza, ambiguità, confusione etc... Esistono alternative:



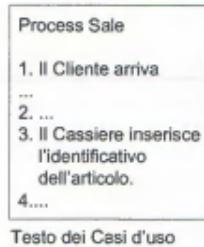
I requisiti hanno un formato standard, usano il linguaggio in modo consistente, evitano il gergo tecnico e evidenziano delle porzioni di testo per identificare le parti più importanti dei requisiti. Un esempio di standard è quello IEEE830.

I casi d'uso possono essere visualizzati con l'UML:



# Capitolo 3

## Casi d'uso



*I casi d'uso sono storie scritte, testuali, di qualche attore che usa un sistema per raggiungere degli obiettivi.* I casi d'uso possono a loro volta influenzare molti altri elaborati dell'analisi, progettazione, implementazione, gestione del progetto e test. Passiamo a qualche definizione:

- un **attore** è qualcosa o qualcuno dotato di comportamento, come una persona (caratterizzata da un ruolo), un sistema informatico o un'organizzazione
- uno **scenario o istanza di caso d'uso** è una sequenza specifica di azioni e interazioni tra il sistema e alcuni attori. Uno scenario descrive una particolare storia nell'uso del sistema, o un percorso attraverso il caso d'uso

*Informalmente* un caso d'uso quindi può essere definito come una collezione di scenari correlati, di successo e fallimento, che descrivono un attore che usa un sistema per raggiungere un obiettivo.

**UP** definisce il modello di caso nell'ambito della disciplina dei Requisiti. Inoltre i **casi d'uso sono documenti di testo, non diagrammi**, e la modellazione dei casi d'uso è innanzitutto **un atto di scrittura di testi, non di disegno di diagrammi**. Il Modello dei Casi d'Uso può includere, opzionalmente, un diagramma UML dei casi d'uso che mostra i nomi

dei casi d'uso e degli attori, nonché le relazioni tra di essi. Esso costituisce un buon diagramma di contesto di un sistema e del suo ambiente, oltre a costituire un indice di facile consultazione dei nomi dei casi d'uso. Si ricorda che **i casi d'uso non sono orientati ad oggetti**.

I casi d'uso sono un buon metodo per mantenere la semplicità e consentire agli esperti di dominio o ai fornitori di requisiti di scrivere essi stessi i casi d'uso, o perlomeno partecipare alla loro scrittura. Un altro valore dei casi d'uso è che mettono in risalto gli obiettivi e il punto di vista dell'utente.

I casi d'uso sono requisiti, soprattutto requisiti funzionali o comportamentali, che indicano che cosa farà il sistema. In UP e in molti metodi moderni, i casi d'uso sono il meccanismo centrale che viene consigliato per la scoperta e la definizione dei requisiti.

Torniamo a parlare di attori. Un attore è qualcosa o qualcuno dotato di comportamento. Anche il sistema in discussione, *SuD*, da *system under discussion*, stesso è un attore, quando ricorre ai servizi di altri sistemi. Si hanno tre tipi di attori correlati al SuD:

1. **attore primario** che raggiunge degli obiettivi utente utilizzando i servizi del SuD. Identificare gli attori primari è utile per trovare gli obiettivi degli utenti, che guidano i casi d'uso
2. **attore di supporto** che offre un servizio (per esempio, informazioni) al SuD. Spesso è un sistema informatico, ma potrebbe essere un'organizzazione o una persona. Identificare gli attori di supporto è utile per chiarire le interfacce esterne e i loro protocolli
3. **attore fuori scena** che ha un interesse nel comportamento del caso d'uso, ma non è un attore primario o di supporto. Identificare gli attori fuori scena è utile per garantire che tutti gli interessi necessari vengano individuati e soddisfatti. Gli interessi degli attori fuori scena sono spesso sottili o facili da tralasciare, a meno che gli attori stessi non vengano esplicitamente nominati

Si hanno inoltre tre diversi livelli di **formalità** per i casi d'uso:

1. **formato breve** ovvero un riepilogo conciso di un solo paragrafo, normalmente relativo al solo scenario principale di successo. Si usa durante l'analisi iniziale dei requisiti, per capire rapidamente l'argomento e la portata. È possibile scriverlo in pochi minuti
2. **formato informale** ovvero più paragrafi, scritti in modo informale, relativi a vari scenari. Si usa come il formato breve ma qui si ha un livello di dettaglio maggiore

3. **formato dettagliato** dove tutti i passi e le variazioni sono scritti nel dettaglio; ci sono anche delle sezioni di supporto, come le pre-condizioni e le garanzie di successo. Va usato opo che molti casi d'uso sono stati identificati e scritti in formato breve, alcuni casi d'uso (circa il 10%) che hanno maggior valore e che sono più significativi dal punto di vista dell'architettura vengono scritti in formato dettagliato. Si ha quindi:

<b>Sezione del caso d'Uso</b>	<b>Commento</b>
<b>Nome del Caso d'Uso</b>	Inizia con un verbo
<b>Portata</b>	Il sistema che si sta progettando
<b>Livello</b>	"Obiettivo utente" o "sottofunzione"
<b>Attore Primario</b>	Nome dell'attore primario
<b>Parti Interessate e Interessi</b>	A chi interessa questo caso d'uso e che cosa desidera
<b>Pre-condizioni</b>	Che cosa deve essere vero all'inizio del caso d'uso
<b>Garanzia di successo</b>	Che cosa deve essere vero se il caso d'uso viene completato con successo
<b>Scenario Principale di Successo</b>	Uno scenario comune di attraversamento del caso d'uso, di successo e incondizionato
<b>Estensioni</b>	Scenari alternativi, di successo e di fallimento
<b>Requisiti speciali</b>	Requisiti non funzionali correlati
<b>Elenco delle variabili tecnologiche e dei dati</b>	Varianti nei metodi di I/O e nel formato dei dati
<b>Frequenza di ripetizione</b>	Frequenza prevista di esecuzione del caso d'uso
<b>Varie</b>	Altri aspetti, ad esempio i problemi aperti

Parliamo ora di scenari. Si hanno due tipologie:

1. **scenario principale (di successo)** che descrive un percorso di successo tipico che soddisfa gli interessi delle parti interessate. Si noti che spesso non comprende alcuna condizione o diramazione. Questo rende il caso d'uso più comprensibile e più estensibile. È formato da una sequenza di passi:
  - (a) un'interazione tra attori
  - (b) una validazione (solitamente effettuata dal sistema)
  - (c) un cambiamento di stato da parte del sistema

Il primo passo di un caso d'uso non rientra sempre in questa classificazione, ma può indicare l'evento (trigger) che scatena l'esecuzione dello scenario. I nomi degli attori vengono di solito scritti con l'iniziale maiuscola, per facilitarne l'identificazione. Un esempio:

**Scenario principale di successo:**

1. Il Cliente arriva alla cassa POS con gli articoli da acquistare.
2. Il Cassiere inizia una nuova vendita.
3. Il Cassiere inserisce il codice identificativo dell'articolo.
4. Il Sistema ...  
Il Cassiere ripete i passi 3-4 fino a che non indica che ha terminato.
5. ...

2. **scenari alternativi (o flussi alternativi o estensioni)** che descrivono tutte le diramazioni e gli altri scenari, sia di successo che di fallimento. La loro stesura è spesso lunga e complessa. Nella scrittura completa dei casi d'uso, la combinazione del flusso di base e degli altri scenari descritti dalle estensioni dovrebbe soddisfare "quasi" tutti gli interessi delle parti interessate. Tuttavia, alcuni interessi possono essere descritti come requisiti non funzionali, e pertanto riportati nelle Specifiche Supplementari piuttosto che nei casi d'uso. Gli scenari relativi alle estensioni sono diramazioni dello scenario principale di successo, e vengono indicati con riferimento ai suoi passi. Si contrassegnano con una lettera posta dopo il numero che identifica il passo (per il passo 3 si avrà 3a, 3b etc...) inoltre prima riporta la condizione, poi la reazione. Per esempio:

**3a. Codice identificativo dell'articolo non valido (non trovato nel sistema):**

1. Il Sistema segnala l'errore e rifiuta l'inserimento.

**3b. Ci sono più articoli della stessa categoria e non è importante tenere traccia dell'identità univoca dell'articolo (per esempio, 5 confezioni di hamburger vegetariani):**

1. Il Cassiere può inserire il codice identificativo della categoria dell'articolo e la quantità.

Un'estensione è costituita da condizione e gestione. La gestione di un'estensione può essere riassunta in un unico passo, oppure può comprendere una sequenza di passi. Al termine della gestione di un'estensione, per default lo scenario si fonde di nuovo con lo scenario principale di successo, a meno che l'estensione non indichi diversamente (per esempio, l'arresto del sistema). In alcuni casi, i punti di estensione sono abbastanza complessi. Questo può essere un motivo per esprimere l'estensione come un caso d'uso separato.

Un caso d'uso può diramarsi in un altro caso d'uso e in effetti, i casi d'uso vengono spesso scritti come ipertesti, e l'esecuzione di un altro caso d'uso viene rappresentata come un collegamento ipertestuale. In questo caso, facendo

### *Capitolo 3. Casi d'uso*

---

clic sul nome del caso d'uso sottolineato, ne verrà visualizzato il testo. Se un requisito non funzionale, un attributo di qualità o un vincolo si riferiscono in modo specifico a un caso d'uso, allora è bene registrarlo insieme al caso d'uso. Potrebbe trattarsi di qualità come prestazioni, affidabilità e usabilità, oppure di vincoli di progettazione, che siano stati imposti o considerati probabili. Si hanno quindi i **requisiti speciali**.

## *Capitolo 3. Casi d'uso*

---

Vediamo ora un esempio completo di caso d'uso:

### Elabora Vendita

Portata: Applicazione POS NextGen

Livello: Obiettivo utente

Attore primario: Cassiere

Parti Interessate e Interessati

- Cassiere: vuole un inserimento dei dati in modo preciso e rapido. Non vuole errore nei pagamenti, perché gli ammanchi di cassa vengono detratti dal suo stipendio
- Addetto alle vendite: Vuole che le commissioni sulle vendite siano aggiornate
- Cliente: vuole effettuare acquisti e fruire di un servizio rapido, nel modo più semplice possibile. Vuole una visualizzazione chiara degli articoli inseriti e dei loro prezzi. Vuole una prova d'acquisto per una eventuale restituzione o sostituzione
- Azienda: Vuole registrare accuratamente le transazioni effettuate e soddisfare gli interessi dei clienti. Vuole che vengano registrati i pagamenti da riscuotere tramite il Servizio di Autorizzazione di Pagamento. Vuole una certa tolleranza ai guasti per consentire di effettuare vendite anche se alcuni componenti del server (per esempio l'autorizzazione remota di pagamento con credito) non sono disponibili. Vuole un aggiornamento automatico e rapido della contabilità e dell'inventario.
- Direttore: Vuole essere in grado di eseguire rapidamente operazioni di sovrascrittura e risolvere in modo semplice i problemi del Cassiere.
- Enti governativi e Fiscali: Vogliono riscuotere le imposte su ciascuna vendita. Possono essere più enti: nazionale, regionale e provinciale.
- Servizio di Autorizzazione e di Pagamento: Vuole ricevere le richieste elettroniche di autorizzazione nel formato e nel protocollo corretto. Vuole una contabilità dettagliata dei suoi debiti verso il negozio.

Pre-condizioni: Il Cassiere è identificato e autenticato

Garanzia di successo (o Post-condizioni): La vendita viene salvata. Le imposte sono calcolate correttamente. La contabilità e l'inventario sono aggiornati. Le commissioni sono registrate. Viene generata una ricevuta. Le approvazioni alle autorizzazioni di pagamento sono registrate.

Scenario principale di successo

1. Il cliente arriva alla cassa POS con gli articoli e/o i servizi da acquistare
2. Il cassiere inizia una nuova vendita
3. Il cassiere inserisce il codice identificativo dell'articolo
4. Il sistema registra la riga di vendita per l'articolo e mostra la descrizione dell'articolo, il suo prezzo, il totale parziale. Il prezzo è calcolato in base a un insieme di regole di prezzo.

Il Cassiere ripete i passi 3-4 fino a che non indica che ha terminato

5. Il sistema mostra il totale con le imposte calcolate.
6. Il Cassiere riferisce il totale al Cliente, e richiede il pagamento
7. Il Cliente paga e il Sistema gestisce il pagamento
8. Il Sistema registra la vendita completata e invia informazioni sulla vendita e sul pagamento ai sistemi esterni di Contabilità (per la contabilità e le commissioni) e di Inventario (per l'aggiornamento dell'inventario)
9. Il Sistema genera la ricevuta
10. Il Cliente va via con la ricevuta e gli articoli acquistati

## *Capitolo 3. Casi d'uso*

---

### Estensioni

- \*a In qualsiasi momento, il Direttore chiede di eseguire una operazione di sovrascrittura:
  1. Il sistema passa alla modalità autorizzata "Direttore"
  2. Il Direttore o il Cassiere esegue un'operazione nella modalità "Direttore"; per esempio riprendere una vendita sospesa su un altro registratore di cassa, annullare una vendita, cambiare la chiusura di cassa, e così via.
  3. Il sistema torna alla modalità autorizzata "Cassiere"
- \*b In qualsiasi momento, il sistema fallisce: per consentire il ripristino e una gestione corretta della contabilità, bisogna garantire che tutto lo stato transazionale significativo possa essere ripristinato, a partire da qualsiasi passo dello scenario
  1. Il Cassiere riavvia il Sistema, si autentica, e richiede il ripristino dello stato precedente
  2. Il Sistema ricostruisce lo stato precedente
    - 2a Il Sistema rileva delle anomalie che impediscono il ripristino:
      1. Il Sistema segnala un errore al Cassiere, registra l'errore e passa a uno stato pulito
      2. Il Cassiere inizia una nuova vendita
  - 1a Il Cliente o il Direttore chiedono di riprendere una vendita sospesa
    1. Il Cassiere esegue l'operazione di ripresa e inserisce il codice identificativo della vendita da riprendere
    2. Il Sistema visualizza lo stato della vendita ripresa, con il totale parziale
      - 2a Vendita non trovata:
        1. Il Sistema segnala l'errore al Cassiere
        2. Il Cassiere probabilmente inizia una nuova vendita e reinserisce tutti gli articoli
      3. Il Cassiere continua con vendita (probabilmente inserendo altri articoli o gestendo il pagamento)
  - 2-4a Il cliente dice al Cassiere di godere di un'esenzione dalle imposte (per esempio perché è anziano)
    1. Il Cassiere verifica, quindi inserisce il codice per lo stato di esenzione dalle imposte
    2. Il Sistema registra lo stato (che utilizzerà durante il calcolo delle imposte)
  - 3a Codice identificativo dell'articolo non valido (non trovato nel sistema):
    1. Il Sistema segnala l'errore e rifiuta l'inserimento
    2. Il Cassiere risponde all'errore:
  - 2a C'è un codice identificativo dell'articolo leggibile (per esempio un codice UPC numerico):
    1. Il Cassiere inserisce il codice dell'articolo manualmente
    2. Il sistema visualizza descrizione e prezzo
  - 2a Codice identificativo dell'articolo non valido: Il Sistema segnala l'errore. Il cassiere prova in un altro modo.
  - 2b Non c'è un codice identificativo dell'articolo, ma sul cartellino è presente un prezzo:
    1. Il Cassiere chiede al direttore di eseguire un'operazione di sovrascrittura
    2. Il Direttore esegue la sovrascrittura
    3. Il Cassiere richiede l'inserimento manuale del prezzo, inserisce il prezzo e richiede la tassazione standard per questo importo (poiché non vi sono informazioni sul prodotto, il calcolatore delle imposte non saprebbe altrimenti come effettuare la tassazione)
    - 2c Il Cassiere esegue Aiuto Ricerca Prodotto per ottenere il vero codice identificativo dell'articolo e il suo prezzo.
    - 2d Altrimenti, il Cassiere chiede a un dipendente il codice identificativo effettivo dell'articolo o il suo prezzo, e inserisce il codice o il prezzo manualmente (vedi sopra)
  - 3b ...

## *Capitolo 3. Casi d'uso*

---

### Requisiti speciali:

- Interfaccia utente di tipo touch screen su un monitor piatto grande. Il testo deve essere visibile da una distanza di un metro
- Risposta all'autorizzazione di credito entro 30 secondi il 90% delle volte
- In qualche modo si desidera un ripristino robusto quando non riesce l'accesso ai servizi remoti, come per esempio il sistema di inventario
- Internazionalizzazione della lingua sul testo visualizzato
- Regole di business inseribili nei passi da 3 a 7
- ...

### Elenco delle varianti tecnologiche e dei dati:

- \*a Richiesta di sovrascrittura da parte del Direttore effettuata passando una scheda apposita attraverso un lettore di schede, oppure inserendo un codice di autorizzazione con la tastiera
- 3a Codice identificativo dell'articolo inserito tramite lettore laser di codici a basse (se il codice a barre è presente) oppure tramite tastiera
- 3b Il codice identificativo dell'articolo può essere basato su uno tra gli schemi di codifica UPC, EAN, JAN o SKU.
- 7a Le informazioni sulla carta di credito sono inserite tramite lettore di schede o tramite tastiera
- 7b Firma per il pagamento con carta di credito ottenuta su ricevuta cartacea. Ma si prevede che, entro due anni, molti vorranno la cattura digitale della firma.

Frequenza di Ripetizione: Potrebbe essere quasi ininterrotta

### Problemi aperti:

- Come variano le leggi fiscali?
- Esaminare la questione del ripristino dei servizi remoti
- Quale personalizzazione è necessaria per le diverse aziende?
- Il cassiere deve estrarre e portare via il cassetto della cassa quando effettua il logout?
- Il cliente può usare direttamente il lettore di schede o lo deve fare il Cassiere?

Si hanno delle linee guida per la stesura dei casi d'uso. Si cerca uno stile essenziale, ignorando l'interfaccia utente e concentrandosi sullo scopo dell'attore, ignorando come un sistema fa qualcosa ma concentrandosi su cosa deve fare. Per facilitare la lettura dei requisiti, è perciò opportuno scrivere i casi d'uso in modo conciso, e allo stesso tempo in modo completo (con una chiara indicazione di soggetto verbo e di eventuali frasi subordinate).

I **casi d'uso a scatola nera** sono il tipo più comune e consigliato; non descrivono il funzionamento interno del sistema, i suoi componenti o aspetti relativi al suo progetto. Piuttosto, il sistema è descritto come dotato di responsabilità. Questa è una metafora comune e unificante nel pensare orientato agli oggetti; gli elementi software hanno responsabilità e collaborano con altri elementi dotati di responsabilità. Usando i casi d'uso a scatola nera per definire le responsabilità del sistema, è possibile specificare che cosa deve fare il sistema (comportamento o requisiti funzionali) senza decidere come lo farà (progettazione). In effetti, la definizione dell'"analisi" rispetto alla "progettazione" è talvolta riassunta come "che cosa" rispetto a "come". Questo è un tema importante nello sviluppo del software: durante l'analisi dei requisiti bisogna specificare il comportamento esterno del sistema, considerato a scatola nera, evitando di prendere decisioni sul "come". Successivamente, durante la progettazione, andrà creata una soluzione che soddisfa le specifiche.

## Capitolo 3. Casi d'uso

---

Per esempio:

Stile a scatola nera	Non a scatola nera
Il Sistema registra la vendita.	Il Sistema memorizza la vendita in una base di dati. ...o (ancora peggio): Il Sistema esegue un'istruzione SQL INSERT per la vendita...

È anche importante trovare i casi d'uso e si ha una procedura di base per farlo:

1. scegliere i confini del sistema, ovvero definire il sistema al meglio. Per chiarire la definizione dei confini del sistema in corso di progettazione, è utile sapere che gli attori primari e gli attori di supporto sono considerati esterni al sistema. Una volta identificati gli attori esterni, i confini diventano più chiari.
2. identificare gli attori primari. Alcune volte sono gli obiettivi a svelare gli attori, o viceversa. Si possono usare una serie di domande per facilitare l'operazione:

Chi avvia e arresta il sistema?	Chi si occupa dell'amministrazione del sistema?
Chi si occupa della gestione degli utenti e della sicurezza?	Il "tempo" è un attore, nel senso che il sistema esegue operazioni in risposta ad alcuni eventi temporali? Chi valuta le attività e le prestazioni del sistema?
Esiste un processo di monitoraggio che riavvia il sistema se si verifica un errore?	Chi valuta i log? Vi si accede in remoto?
Come vengono gestiti gli aggiornamenti software? Sono aggiornamenti automatici o a richiesta?	Chi viene avvisato quando si verificano errori o guasti?
Oltre agli attori primari <i>umani</i> , ci sono sistemi software o robotizzati esterni che utilizzano i servizi del sistema?	

3. identificare gli obiettivi di ciascun attore primario. Si crea quindi un elenco attori-obiettivi, tipo questo:

Attore	Obiettivo	Attore	Obiettivo
Cassiere	elaborare le vendite elaborare i noleggi gestire le restituzioni cash in cash out ...	Amministratore del Sistema	aggiungere utenti modificare utenti eliminare utenti gestire sicurezza gestire tabelle di sistema ...
Direttore ...	avviare il sistema arrestare il sistema ...	Sales Activity System	analizzare dati sulla vendita ...

inoltre risulta più produttivo, nella realtà, chiedere all'attore stesso quale sia il suo obiettivo.

Un altro approccio per trovare attori, obiettivi e casi d'uso è basato sull'identificazione di eventi esterni. Spesso un gruppo di eventi appartiene allo stesso caso d'uso. Per esempio:

Evento esterno	Dall'attore	Obiettivo/Caso d'uso
inserire la riga di vendita per un articolo	Cassiere	elaborare una vendita
inserire il pagamento	Cassiere o Cliente	elaborare una vendita
...		

- definire i casi d'uso che soddisfano gli obiettivi degli utenti; il loro nome va scelto in base all'obiettivo. Di solito, i casi d'uso a livello di obiettivo utente saranno in corrispondenza biunivoca con gli obiettivi degli utenti. **Il nome dei casi d'uso deve iniziare con un verbo e, in generale va definito un caso d'uso per ciascun obiettivo utente.** È opportuno assegnare al caso d'uso un nome simile all'obiettivo dell'utente.

*un'eccezione comune alla scelta di un caso d'uso per obiettivo è quella che riunisce gli obiettivi separati CRUD (acronimo di create, retrieve, update, delete, ovvero creare, leggere, aggiornare, eliminare) in un unico caso d'uso CRUD, chiamato Gestisci <X>. Per esempio, gli obiettivi "modifica utente", "elimina utente" e così via sono tutti soddisfatti dal caso d'uso Gestisci Utenti.*

Naturalmente, nello sviluppo iterativo ed evolutivo, non tutti gli obiettivi o i casi d'uso saranno identificati completamente o correttamente fin dall'inizio. Anche la scoperta dei requisiti è evolutiva. Bisogna poi verificare la validità dei casi d'uso. Si hanno quindi tre test:

- **test del capo:** *il capo vi chiede: "Cosa avete fatto tutto il giorno?" e voi rispondete: "Il login!". Il vostro capo sarà felice?.* Se non lo è il caso d'uso non supera il test del capo, il che significa che non è fortemente mirato a ottenere risultati il cui valore sia misurabile. Potrebbe essere un caso d'uso a un livello più basso, ma non al livello a cui è desiderabile concentrarsi durante l'analisi dei requisiti. Ciò non significa che bisogna sempre ignorare i casi d'uso che non superano il test del capo
- **test EBP (Elementary Business Process),** che deriva dall'ingegneria dei processi di business: *Un processo di business elementare è un'attività svolta da una persona in un determinato tempo e luogo, in*

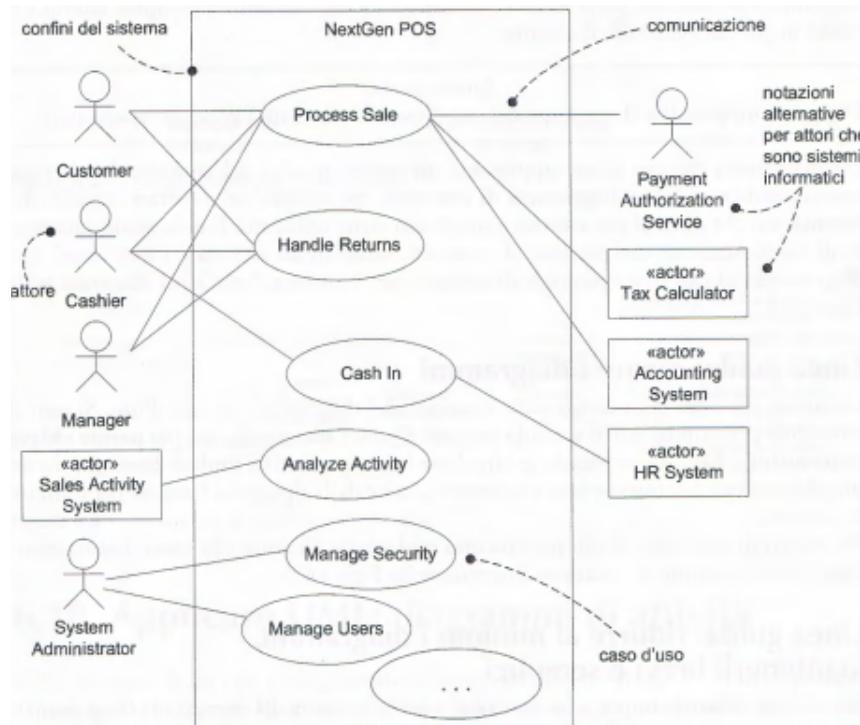
*risposta a un evento di business, che aggiunge un valore di business misurabile e lascia i dati in uno stato coerente; per esempio, "Approva un credito" o "Stabilisci il prezzo per un ordine". Come nella definizione di UP, enfatizza l'aggiunta di un valore di business osservabile o misurabile, e giunge a una situazione in cui il sistema e i dati sono in uno stato stabile e coerente. Il test EBP è simile al test del capo, soprattutto in termini di qualifica in termini di valore aziendale misurabile.*

- **test della dimensione**, che si basa sul fatto che un caso d'uso è formato da diversi passi (e nel complesso si hanno da 3 a 10 pagine di testo). Un errore comune nella modellazione dei casi d'uso è definire un caso d'uso a sé formato da un singolo passo.

Ci potrebbero essere delle eccezioni a questi test.

### 3.1 Diagrammi dei casi d'uso

UML fornisce una notazione dei diagrammi dei casi d'uso che consente di illustrare i nomi e gli attori dei casi d'uso, nonché le relazioni tra gli stessi, per esempio:

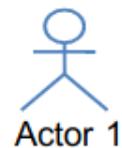


I diagrammi dei casi d'uso e le relazioni tra casi d'uso sono secondari nel lavoro che riguarda i casi d'uso. I casi d'uso sono documenti di testo. Lavorare sui casi d'uso significa scrivere testo. Un diagramma dei casi d'uso rappresenta un ottimo quadro del contesto del sistema; esso costituisce un buon diagramma di contesto, che consiste nel mostrare i confini del sistema, ciò che giace al suo esterno e come esso viene utilizzato. È utile come strumento di comunicazione che riassume il comportamento di un sistema e i suoi attori.

*Per ribadire, il lavoro importante dei casi d'uso è la scrittura del testo, non i diagrammi o le relazioni tra casi d'uso. Se un'organizzazione spreca molte ore (o peggio ancora, giorni) a lavorare su un diagramma dei casi d'uso e a discutere le relazioni tra casi d'uso anziché concentrarsi sulla scrittura del testo, vuol dire che il tempo è stato impiegato male.*

A livello di notazione si ha:

- **gli attori** sono rappresentati da un omino, col nome scritto sotto:

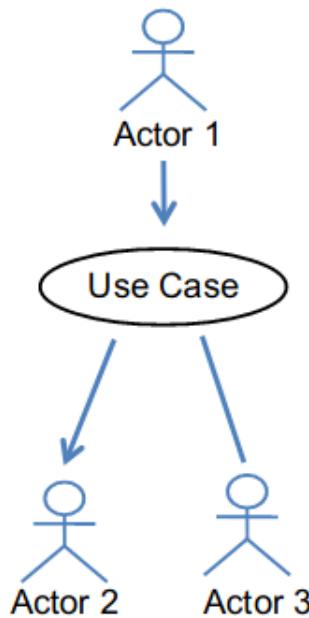


- **i casi d'uso** sono rappresentati con un ovale contenente la dicitura del caso d'uso:

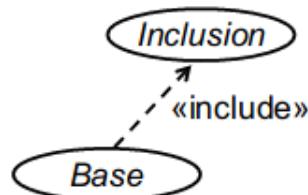


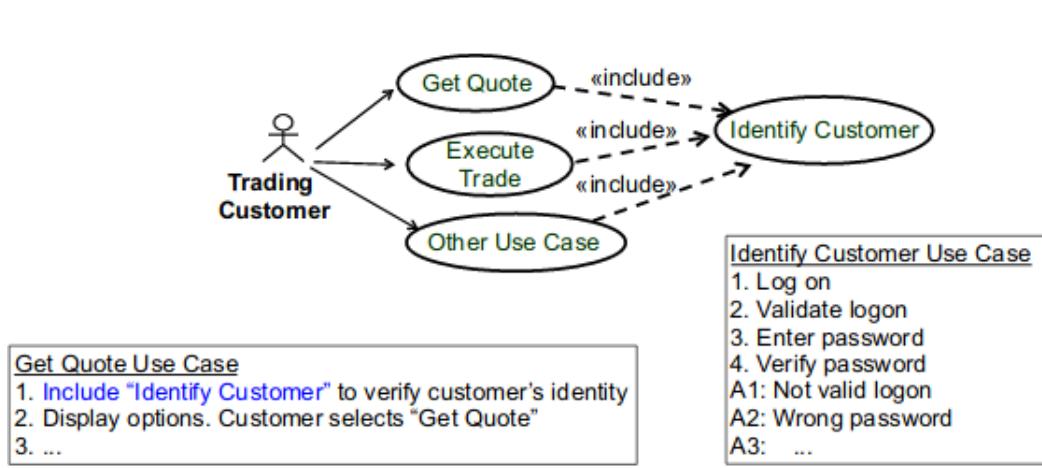
- **le associazioni** sono canali di comunicazione tra attore e caso d'uso.  
Si hanno due rappresentazioni:

1. **una linea continua direzionale**, per specificare chi da inizio all'interazione
2. **una linea continua non direzionale**, per indicare che entrambe le parti possono dare inizio ad un'interazione



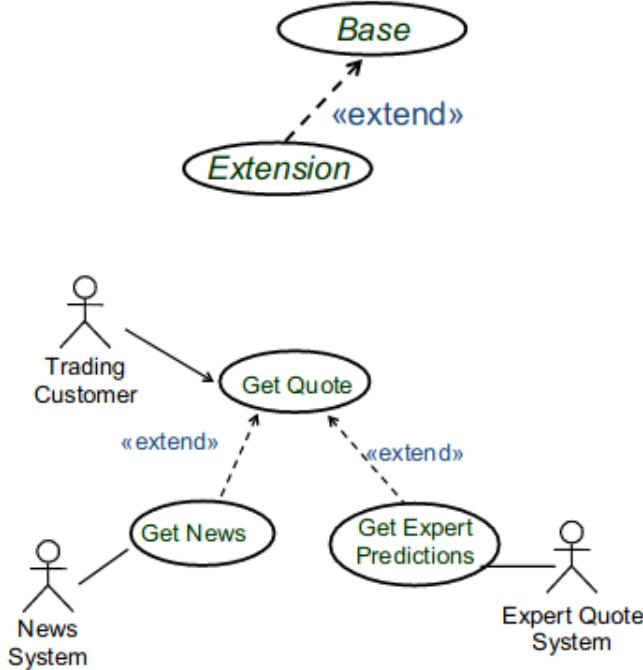
- **l'include (inclusione)** si rappresenta con una linea tratteggiata direzionata con l'indicazione «*include*» e indica la relazione tra un caso d'uso base ed un caso d'uso incluso nel caso d'uso base:





il caso d'uso è eseguito completamente quando viene raggiunto il punto di inclusione

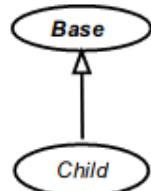
- l'**extend** (estensione), si rappresenta come l'include (specificando «*extend*») e connette un caso d'uso esteso ad un caso d'uso base. Aggiunge varianti ad un caso d'uso base e viene inserito solo se la condizione di estensione è vera. Si hanno estensioni inserite solo in corrispondenza degli extension point:



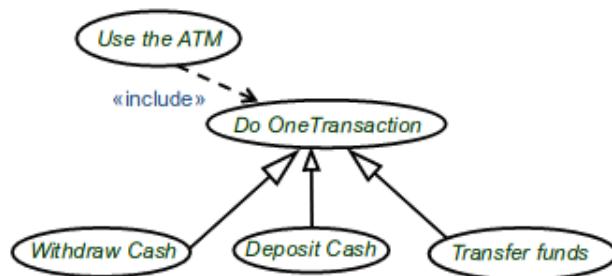
<p><u>Get Quote Use Case</u></p> <p>Basic Flow:</p> <ol style="list-style-type: none"> <li>1. Include "Identify Customer" to verify customer's identity.</li> <li>2. Display options.</li> <li>3. Customer selects "Get Quote."</li> <li>4. Customer gets quote.</li> <li>5. Customer gets other quotes.</li> <li>6. Customer logs off.</li> </ol> <p>A1. Quote System unavailable ... Extension Points: The "Optional Services" extension point occurs at Step 3 in the Basic Flow and Step A1.7 in Quote System Unavailable alternative flow.</p>	<p><u>Get News Use Case</u></p> <p>This use case extends the Get Quote Use Case, at extension point "Optional Services."</p> <p>Basic Flow:</p> <ol style="list-style-type: none"> <li>1. If Customer selects "Get News," the system asks customer for time period and number of news items.</li> <li>2. Customer enters time period and number of items. The system sends stock trading symbol to News System, receives reply, and displays the news to the customer.</li> <li>3. The Get Quote Use Case continues.</li> </ol> <p>A1: News System Unavailable A2: No News About This Stock ... ...</p>
---	---

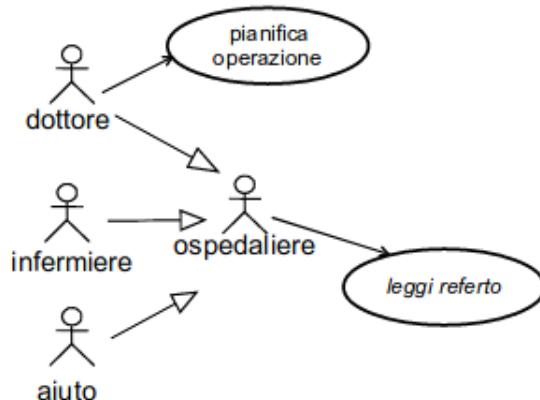
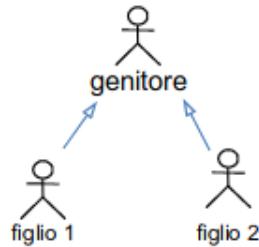
*Il caso d'uso esteso è eseguito quando il punto di estensione è raggiunto e la condizione di estensione è vera*

- **la generalization (generalizzazione)** si ha quando un caso d'uso base è una generalizzazione di un caso d'uso child. Viene rappresentata con una linea continua direzionata con la freccia non riempita:



*Il caso d'uso generalizzato viene eseguito se la condizione di generalizzazione è vera inoltre la generalizzazione è puramente concettuale, non ci sono regole da seguire, come ad esempio l'utilizzo di punti di estensione.*  
Si ha:





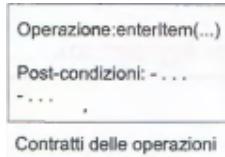
Anche se gli elenchi dettagliati di funzioni non sono consigliabili, le funzionalità del sistema possono essere riassunte in modo utile in un *elenco di caratteristiche* chiaro, ad alto livello, chiamato caratteristiche di sistema, come parte del documento di Visione. Contrariamente alle caratteristiche di basso livello, costituite da molte pagine, un elenco di caratteristiche di sistema dovrebbe comprendere solo alcune decine di voci. Esso fornisce un riepilogo conciso delle funzionalità del sistema, indipendente dalla vista dei casi d'uso, come nel seguente esempio. Talvolta i casi d'uso non sono adatti; alcune applicazioni richiedono un punto di vista guidato dalle caratteristiche. Per esempio, i server applicativi, i sistemi di gestione di basi di dati e altri sistemi di middleware o di back-end vanno piuttosto descritti e si evolvono in termini di caratteristiche ("supporto per i Web Services nella prossima versione"). I casi d'uso non sono idonei a queste applicazioni o al modo in cui esse evolvono in termini di forze del mercato.

I casi d'uso sono fondamentali in UP e in molti altri metodi iterativi. UP incoraggia lo sviluppo guidato dai casi d'uso. Si ha quindi:

- i requisiti funzionali sono registrati principalmente nei casi d'uso (Modello dei Casi d'Uso); le altre tecniche per i requisiti (come l'elenco delle funzioni) sono secondarie, o addirittura non utilizzate
- i casi d'uso rivestono un ruolo importante nella pianificazione iterativa. Il lavoro di un'iterazione è definito, in parte, scegliendo alcuni scenari di casi d'uso o interi casi d'uso. I casi d'uso sono inoltre un input fondamentale per stimare costi e tempi di un progetto
- la progettazione è guidata da realizzazioni di casi d'uso; il team progetta oggetti e sottosistemi che collaborano che consentono di eseguire o realizzare i casi d'uso
- i casi d'uso spesso influenzano l'organizzazione dei manuali per l'utente
- i test funzionali o di sistema corrispondono agli scenari dei casi d'uso
- è possibile creare nella UI "creazioni guidate" (wizard) o scelte rapide (shortcut) per facilitare le attività comuni degli scenari dei casi d'uso più comuni e importanti

# Capitolo 4

## Contratti delle Operazioni



In UP, i **casi d'uso** e le **caratteristiche di sistema** sono i modi principali per descrivere il comportamento del sistema. Normalmente sono sufficienti, ma talvolta è utile una descrizione più dettagliata o precisa del comportamento del sistema. I contratti delle operazioni usano pre-condizione e post-condizione per descrivere nel dettaglio i cambiamenti agli oggetti in un modello di dominio, come risultato di un'operazione di sistema.

I contratti delle operazioni possono essere considerati parte del Modello dei Casi d'Uso, poiché forniscono maggiori dettagli dell'analisi sull'effetto delle operazioni di sistema implicate dai casi d'uso.

Definiamo le sezioni di un contratto:

- **operazione:** nome e parametri dell'operazione
- **riferimenti:** casi d'uso in cui può verificarsi questa operazione
- **pre-condizioni:** ipotesi significative sullo stato del sistema o degli oggetti nel Modello di Dominio prima dell'esecuzione dell'operazione. Sono ipotesi non banali.
- **post-condizioni:** è la sezione più importante ovvero lo stato degli oggetti nel Modello di Dominio dopo il completamento dell'operazione. Le post-condizioni descrivono i cambiamenti nello stato degli oggetti nel modello di dominio. I cambiamenti di stato nel modello di dominio comprendono le istanze create, le associazioni formate o rotte e gli

attributi modificati. Le post-condizioni non sono azioni da eseguire nel corso dell'operazione; si tratta piuttosto di osservazioni (rilevazioni) sugli oggetti del modello di dominio che risultano avere al termine dell'operazione. Si dividono in:

- creazione o cancellazione di un'istanza
- cambiamento del valore di un attributo
- associazioni (collegamenti in UML) formate o spezzate

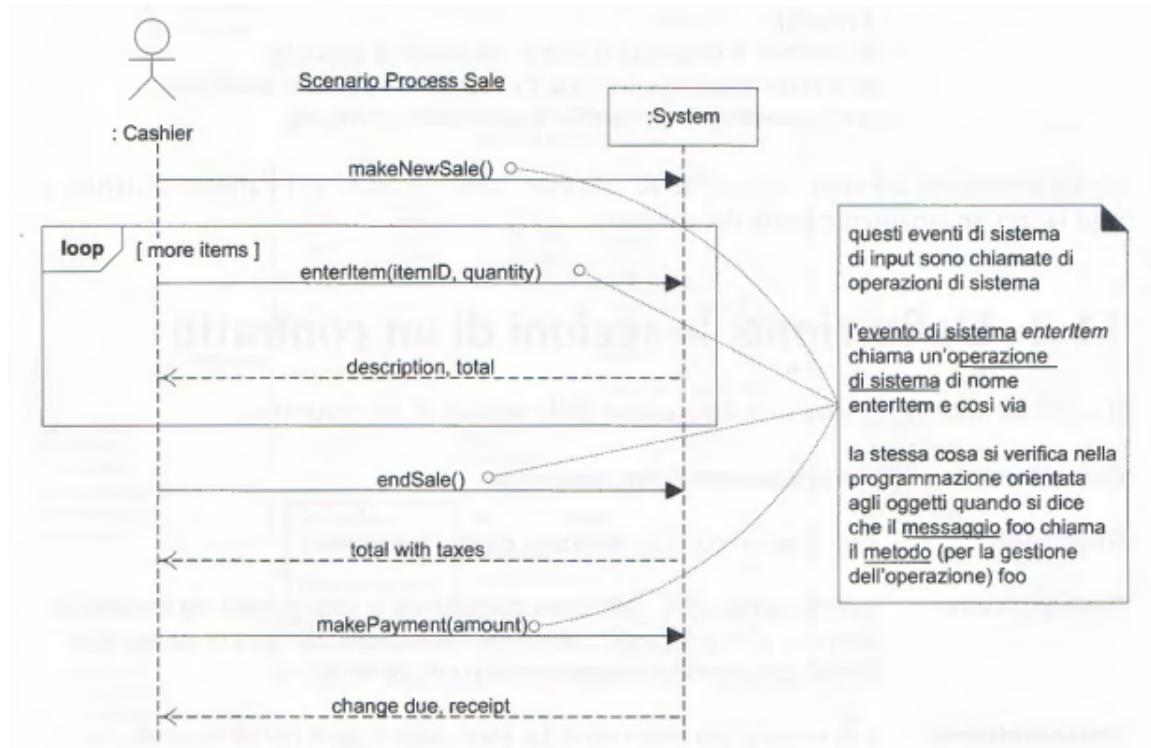
**Le post-condizioni non sono sempre necessarie.** *Le post-condizioni vanno espresse con un verbo al passato, per sottolineare che si tratta di osservazioni dei cambiamenti di stato provocati da un'operazione, e non di un'azione che deve accadere*

I contratti delle operazioni possono essere definiti per le **operazioni di sistema**, ovvero operazioni che il sistema, considerato come un componente a scatola nera, offre nella sua interfaccia pubblica. Le operazioni di sistema possono essere identificate mentre si abbozzano gli SSD, *diagrammi di sequenza di sistema*.

## Capitolo 4. Contratti delle Operazioni

---

Un evento di sistema di input implica che il sistema contenga un'operazione di sistema per gestire quell'evento, così come un messaggio OO (che è un tipo di evento o segnale) viene gestito da un metodo OO (che è un tipo di operazione):



L'intero insieme delle operazioni di sistema, tra tutti i casi d'uso, definisce l'interfaccia di sistema pubblica, considerando il sistema come un singolo componente o una singola classe. In UML, il sistema nel suo insieme può essere rappresentato come un unico oggetto di una classe denominata, per esempio, System.

Durante la creazione dei contratti è normale che emerga la necessità di registrare nuove classi concettuali, attributi o associazioni nel modello di dominio. Non ci si limiti alla formulazione fatta in precedenza del modello di dominio; la si deve ampliare man mano che si fanno nuove scoperte. Nei metodi iterativi ed evolutivi (che riflettono la realtà dei progetti software), tutti gli elaborati dell'analisi e della progettazione sono considerati parziali e imperfetti, ed evolvono in risposta a nuove scoperte.

In UP, i casi d'uso sono il repository principale dei requisiti del progetto. Essi possono fornire tutti i dettagli, o la maggior parte di essi, necessari per sapere che cosa fare durante la progettazione; in questo caso i contratti non sono

d'aiuto. Tuttavia ci sono situazioni in cui non è opportuno descrivere nei casi d'uso tutti i dettagli e la complessità dei cambiamenti di stato richiesti, poiché risulterebbero troppo dettagliati.

Si ha la seguente linea guida per creare contratti:

- identificare le operazioni di sistema dagli SSD
- creare un contratto per le operazioni di sistema complesse o i cui effetti sono probabilmente sottili, o che non sono chiare dai casi d'uso
- descrivere le post-condizioni con le seguenti categorie:
  - creazione o cancellazione di istanza
  - modifica di attributo
  - associazione formata o spezzata

*Il problema più comune è dimenticarsi di includere la formazione di associazioni. In particolare quando sono create nuove istanze, è molto probabile che debbano essere stabilire delle associazioni con diversi oggetti.*

## 4.1 Operazioni e UML

UML definisce formalmente la nozione di **operazione**: *un'operazione è una specifica di una trasformazione o di una interrogazione che un oggetto può essere chiamato ad eseguire.*

In UML sono operazioni, per esempio, gli elementi di un'interfaccia. Sono quindi **astrazioni** e non implementazioni. In UML, invece, un **metodo** è: *l'implementazione di un'operazione. Esso specifica l'algoritmo o la procedura associata a un'operazione.*

Nel metamodello di UML, un'operazione ha una **firma** (o signature, con nome e parametri) e, cosa più importante in questo contesto, è associata a un insieme di oggetti UML di tipo Constraint (Vincolo) classificati come pre-condizioni e post-condizioni che specificano la semantica dell'operazione. UML definisce la semantica delle operazioni attraverso i vincoli, che possono essere specificati nello stile delle pre-condizioni e post-condizioni. Si noti che, come sottolineato in questo capitolo, la specifica di un'operazione UML non può mostrare un algoritmo o una soluzione, ma solo i cambiamenti di stato o gli effetti dell'operazione. Oltre a utilizzare i contratti per specificare le operazioni pubbliche dell'intero sistema (ovvero, le operazioni di sistema), i

contratti possono essere applicati alle operazioni a qualsiasi livello di granularità: le operazioni pubbliche (o l’interfaccia) di un sottosistema, di un componente, di una classe astratta, e così via. Per esempio, è possibile definire operazioni per una singola classe software, per esempio Stack. Le operazioni a grana grossa discusse in questo capitolo appartengono a una classe System che rappresenta l’intero sistema considerato come componente a scatola nera, ma in UML le operazioni possono appartenere a qualsiasi classe o interfaccia, tutte con le relative pre-condizioni e post-condizioni.

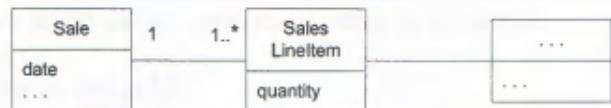
# Capitolo 5

## Modellazione di Dominio

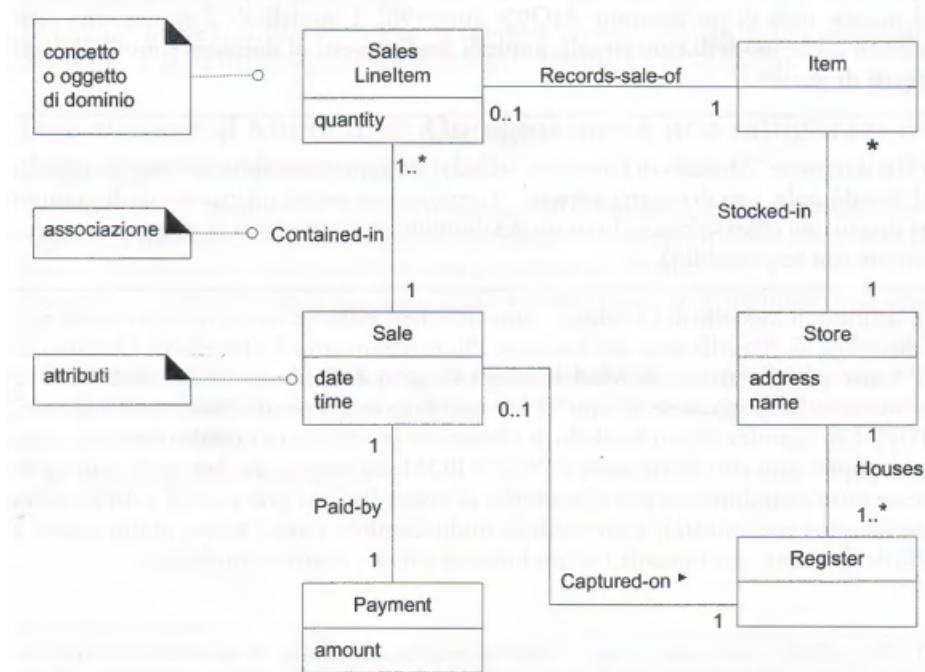
Un **modello di dominio** è il modello più importante e classico dell'analisi OO; esso illustra i concetti significativi di un dominio. Può fungere da sorgente di ispirazione per la progettazione di alcuni oggetti software.

Come tutte le cose nella modellazione agile e nello spirito di UP, il **modello di dominio è opzionale**. Il modello di dominio a sua volta può influenzare i contratti delle operazioni, il glossario e il Modello di Progetto, in particolare gli oggetti software nello strato del dominio del Modello di Progetto.

Vediamo un piccolo esempio:



L'immagine mostra un modello di dominio parziale, disegnato con la notazione di un diagramma delle classi di UML. L'applicazione della notazione dei diagrammi delle classi di UML per un modello di dominio produce un modello secondo un punto di vista concettuale:



Quindi un **modello di dominio** è una rappresentazione visuale di classi concettuali o di oggetti del mondo reale di un dominio. In UP, il termine "Modello di Dominio" indica una rappresentazione di classi concettuali del mondo reale, non di oggetti software. Il termine "non" indica un insieme di diagrammi che descrivono classi software, lo strato del dominio di un'architettura software o oggetti software con responsabilità. Si ha che il Modello di Dominio di UP è una specializzazione del *Modello degli Oggetti di Business*. Applicando la notazione UML, un modello di dominio è illustrato con un insieme di diagrammi delle classi in cui non sono definite operazioni (firme di metodi). Esso fornisce un punto di vista concettuale e può mostrare:

- oggetti di dominio o classi concettuali
- associazioni tra classi concettuali
- attributi di classi concettuali

Ovviamente quanto espresso con la notazione UML poteva essere espresso come testo semplice nel Glossario di UP. Un Modello di Dominio UP è una visualizzazione di oggetti del mondo reale in un dominio di interesse, **non** di oggetti software.

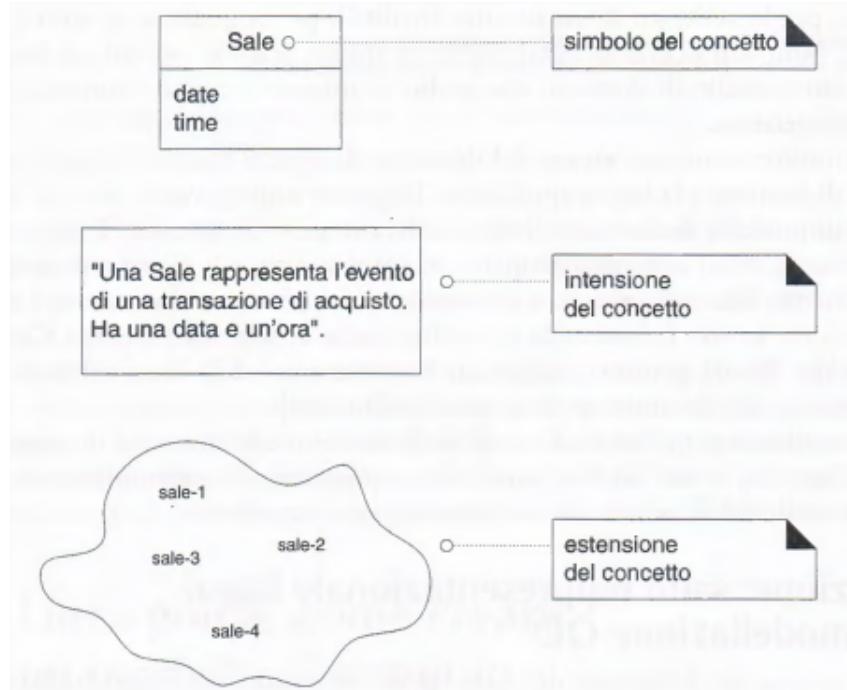
Si indica quindi con **strato di dominio** il secondo significato di modello di dominio, orientato agli oggetti software, come avviene comunemente.

Il modello di dominio illustra **classi concettuali**. Una **classe concettuale**,

informalmente, è un'idea, una cosa, o un oggetto, formalmente, può essere considerata in termini del suo simbolo, della sua intensione e della sua estensione:

- **simbolo:** parole o immagini che rappresentano una classe concettuale
- **intensione:** la definizione di una classe concettuale
- **estensione:** l'insieme di esempi a cui la classe concettuale si applica

Ovvero:



**Un modello di dominio non è un modello di dati.** Non si deve dunque escludere una classe semplicemente perché i requisiti non indicano alcuna necessità ovvia di ricordare informazioni sulla stessa (un criterio comune nella modellazione dei dati per la progettazione di basi di dati relazionali, ma irrilevante nella modellazione di dominio) oppure perché la classe concettuale non ha attributi.

Vediamo le linee guida per creare un modello di dominio. Si hanno i seguenti passo:

1. trovare le classi concettuali. Si hanno tre strategie utili per trovarle:
  - (a) riusare o modificare dei modelli esistenti. Questo è il primo approccio, il migliore e generalmente il più facile dal quale iniziare, se possibile

(b) utilizzare un elenco di categorie:

Categoria di classe concettuale	Esempi
<b>transazioni commerciali</b> <i>Linea guida:</i> sono aspetti critici (riguardano denaro), dunque si inizi con le transazioni.	<i>Sale, Payment, Reservation</i>
<b>elementi/righe di transazioni</b> <i>Linea guida:</i> le transazioni spesso sono composte da righe per gli articoli correlati, quindi queste vanno considerate subito dopo le transazioni.	<i>SalesLineItem</i>
<b>prodotto o servizio correlato a una transazione o a una riga di transazione per articolo</b> <i>Linea guida:</i> le transazioni sono per qualcosa (un prodotto o un servizio). Vanno considerate subito dopo.	<i>Item Flight, Seat, Meal</i>
<b>dove viene registrata la transazione?</b> <i>Linea guida:</i> importante.	<i>Register, Ledger FlightManifest</i>
<b>ruoli di persone o organizzazioni correlati alle transazioni; attori nei casi d'uso</b> <i>Linea guida:</i> normalmente dobbiamo sapere quali sono le parti coinvolte in una transazione.	<i>Cashier, Customer, Store MonopolyPlayer Passenger, Airline</i>
<b>luogo della transazione; luogo del servizio</b>	<i>Store Airport, Plane, Seat</i>
<b>eventi significativi, spesso con un'ora o un luogo che è necessario ricordare</b>	<i>Sale, Payment MonopolyGame Flight</i>
<b>oggetti fisici</b> <i>Linea guida:</i> questo è particolarmente importante quando si crea software per il controllo di dispositivi, oppure simulazioni.	<i>Item, Register Board, Piece, Die Airplane</i>
<b>descrizioni di oggetti</b> <i>Linea guida:</i> vedere il Paragrafo 9.13 per una discussione.	<i>ProductDescription FlightDescription</i>
<b>cataloghi</b> <i>Linea guida:</i> le descrizioni sono spesso contenute in un catalogo.	<i>ProductCatalog FlightCatalog</i>
<b>contenitori di oggetti (fisici o informazioni)</b>	<i>Store, Bin Board Airplane</i>
<b>oggetti in un contenitore</b>	<i>Item Square (in una Board) Passenger</i>
<b>altri sistemi che collaborano</b>	<i>CreditAuthorizationSystem AirTrafficControl</i>
<b>registrazioni di questioni finanziarie, di lavoro, contrattuali e legali</b>	<i>Receipt, Ledger MaintenanceLog</i>
<b>strumenti finanziari</b>	<i>Cash, Check, LineOfCredit TicketCredit</i>
<b>piani, manuali, documenti cui si fa regolarmente riferimento per eseguire il lavoro</b>	<i>DailyPriceChangeList RepairSchedule</i>

(c) identificare nomi e locuzioni nominali. Si usa quindi l'**analisi linguistica**. È basata sull'identificazione di nomi e di locuzioni nominali (ovvero, di frasi oggetti formate da un nome principale insieme ai suoi aggettivi e determinanti) nelle descrizioni testuali di un dominio; questi vengono poi considerati come classi concettuali candidate o attributi. **Occorre prestare particolare attenzione con questo metodo;** una corrispondenza meccanica da nome a classe non è possibile, poiché le parole nel linguaggio naturale sono ambigue.

2. disegnarle come classi in un diagramma delle classi UML
3. aggiungere associazioni e attributi

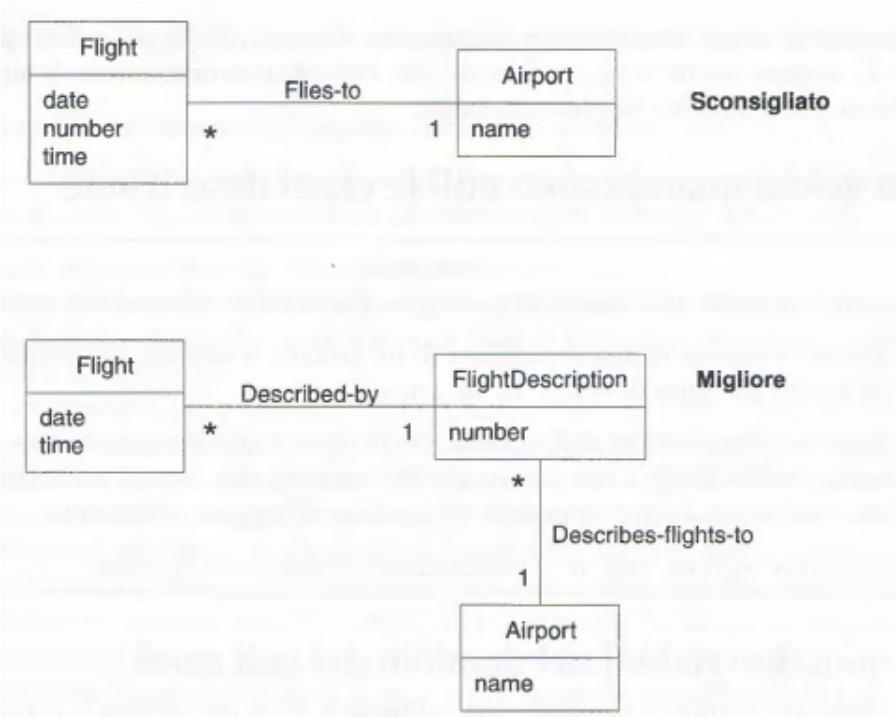
*Il modello di dominio è una visualizzazione di concetti e terminologia di dominio significativi.*

Uno degli errori più comuni durante la creazione di un modello di dominio è quello di rappresentare qualcosa come un attributo mentre dovrebbe essere rappresentato come una classe concettuale. Si ha quindi una regola per evitare questo: *se nel mondo reale non pensiamo a una determinata classe concettuale X come a un numero o a un testo, allora probabilmente X è una classe concettuale, non un attributo.*

Analizziamo ora le **classi descrizione**. Una classe descrizione contiene informazioni che descrivono qualcos'altro. Può infatti accadere che ci sia necessità che alcuni oggetti siano descrizioni (talvolta chiamate specifiche) di qualcosa'altro. La necessità di classi descrizione è comune nei domini relativi a vendite, prodotti e servizi. È comune anche nella produzione, che richiede una descrizione di un prodotto manufatto, che è distinta dal prodotto stesso. Aggiungere una classe descrizione è necessario nei seguenti casi:

- quando è necessaria una descrizione di un articolo o servizio, indipendentemente dall'attuale esistenza di istanze di tali articoli o servizi
- quando l'eliminazione delle istanze degli oggetti che descrivono darebbe luogo a una perdita di informazioni che invece è necessario conservare, ma che sono state erroneamente associate all'oggetto eliminato
- quando si vogliono ridurre le informazioni ridondanti o ripetute

vediamo un esempio:



## 5.1 Associazioni

**Un'associazione** è una relazione tra classi (più precisamente, tra istanze di queste classi) che indica una connessione significativa e interessante.

In UML è definita come *la relazione semantica tra due o più classificatori che comporta connessioni tra le rispettive istanze*.

Le associazioni che è utile mostrare sono solitamente quelle che implicano la conoscenza di una relazione che deve essere memorizzata per una certa durata di tempo, che a seconda del contesto potrebbe essere di millisecondi o di anni.

Si hanno due tipi di associazioni da includere nel modello di dominio:

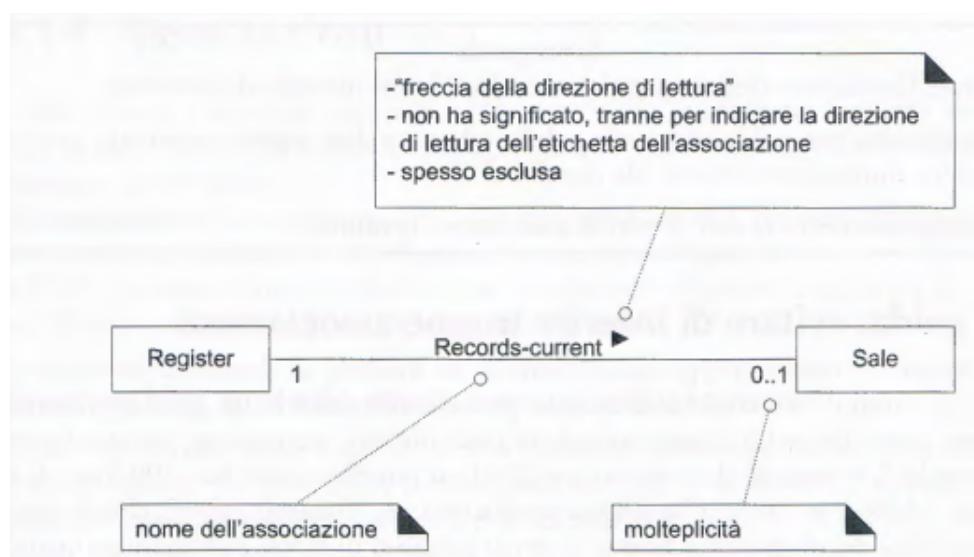
1. associazioni per cui la conoscenza della relazione deve essere conservata per una qualche durata
2. associazioni derivate dall'elenco di associazioni comuni

**È bene evitare di inserire troppe associazioni in un modello di dominio.** Si ricorda che in un grafo con  $n$  nodi ci possono essere  $\frac{n(n-1)}{2}$  associazioni tra nodi distinti, questo implica che si può rischiare di incappare in

problematiche di "disturbo visivo", che diminuisce la leggibilità del diagramma stesso.

### 5.1.1 Notazione per le Associazioni

Un'associazione è rappresentata da una linea che collega le classi partecipanti; il nome di un'associazione ha l'iniziale maiuscola, per differenziarla dalle classi, che hanno l'iniziale maiuscola. Le estremità di un'associazione possono contenere un'espressione di molteplicità, che indica le relazioni numeriche tra le istanze delle classi. Un'associazione è per natura bidirezionale, nel senso che è possibile una navigazione logica dalle istanze di una delle due classi a quelle dell'altra, e viceversa. Questa navigazione è puramente astratta, **non è un'affermazione su connessioni tra entità software**.

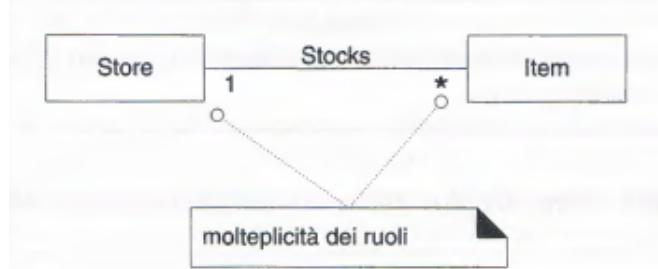


Una "freccia della direzione di lettura" opzionale indica la direzione di lettura del nome dell'associazione; non indica una direzione di visibilità o di navigabilità. Se la freccia non è presente, la convenzione è di leggere l'associazione da sinistra verso destra oppure dall'alto verso il basso, anche se ciò non costituisce una regola in UML.

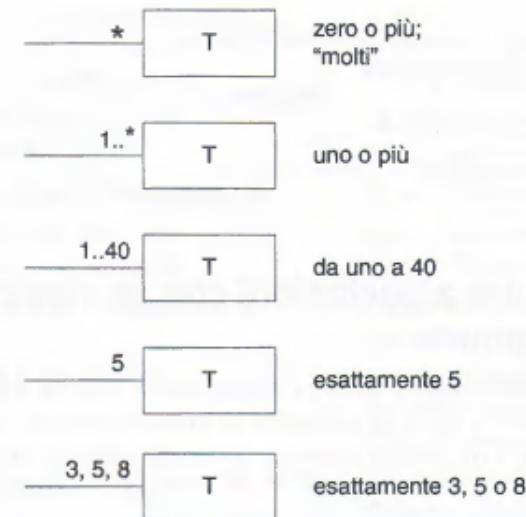
Normalmente si assegna il nome a un'associazione in base al formato *NomeClasse-LocuzioneVerbale-NomeClasse*, in cui la locuzione verbale (ovvero, una frase formata da un verbo principale, più eventuali complemento, oggetto e avverbio) crea una sequenza leggibile e significativa.

Ciascuna estremità di un'associazione è detta ruolo e può avere, optionalmente:

- espressione di molteplicità. La molteplicità definisce quante istanze di una classe A possono essere associate a un'istanza di una classe B, in un particolare momento, piuttosto che in un arco di tempo:

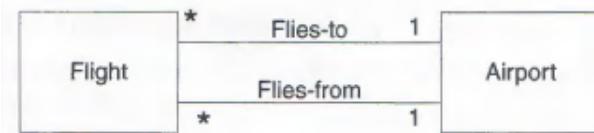


Si hanno diversi valori per una molteplicità:



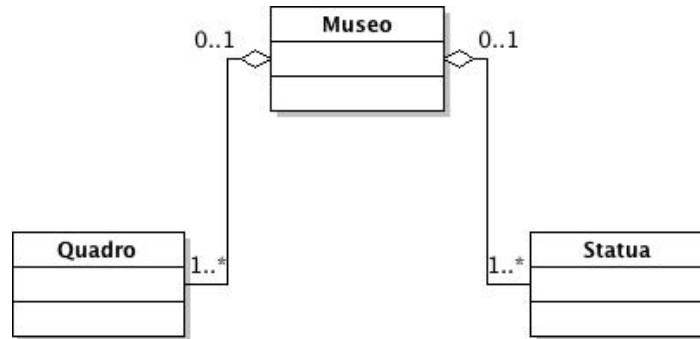
- nome
- navigabilità

Si possono avere due classi collegate da più associazioni:



### 5.1.2 Aggregazione e composizione

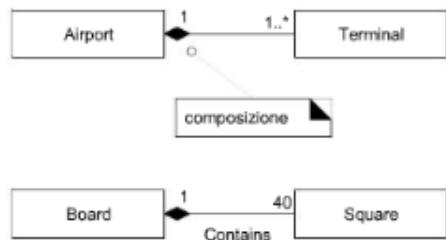
L'**aggregazione** è un tipo di associazione che suggerisce la relazione *intero-parte*. Se ne sconsiglia l'uso ma presenta la seguente notazione:



La **composizione**, nota anche come *aggregazione composta*, è un forte tipo di aggregazione di tipo *intero-parte* e implica che:

- ciascuna istanza della parte appartiene ad una sola istanza del composto alla volta
- ciascuna parte appartiene sempre ad un composto

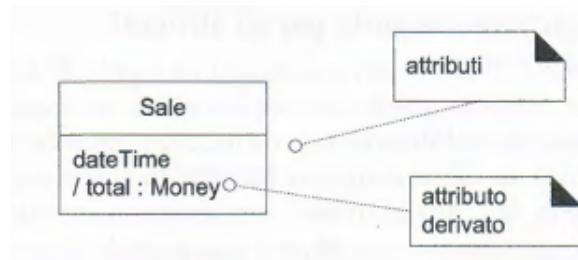
Si ha la seguente notazione:



## 5.2 Attributi

Un **attributo** è un valore logico (un dato) di un oggetto. Vanno inclusi gli attributi che i requisiti (per esempio, i casi d' uso) suggeriscono o di cui implicano una necessità di ricordare informazioni.

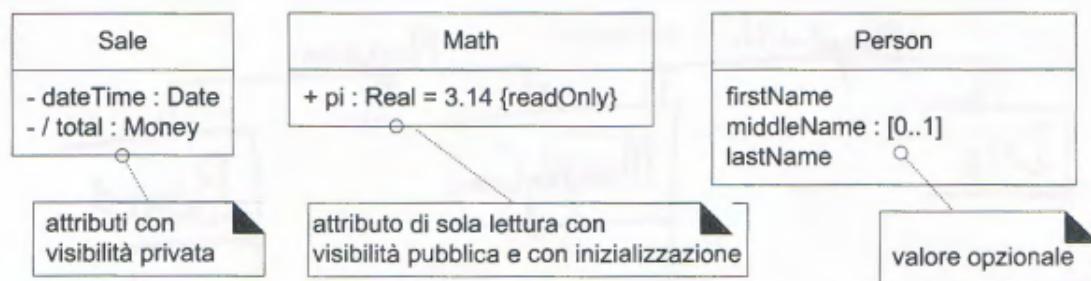
In UML gli attributi sono mostrati nella seconda sezione del rettangolo per una classe, optionalmente è possibile anche mostrare il loro tipo e altre informazioni:



Si ha però una sintassi completa:

*visibility name : type multiplicity = default property-string*

per esempio:

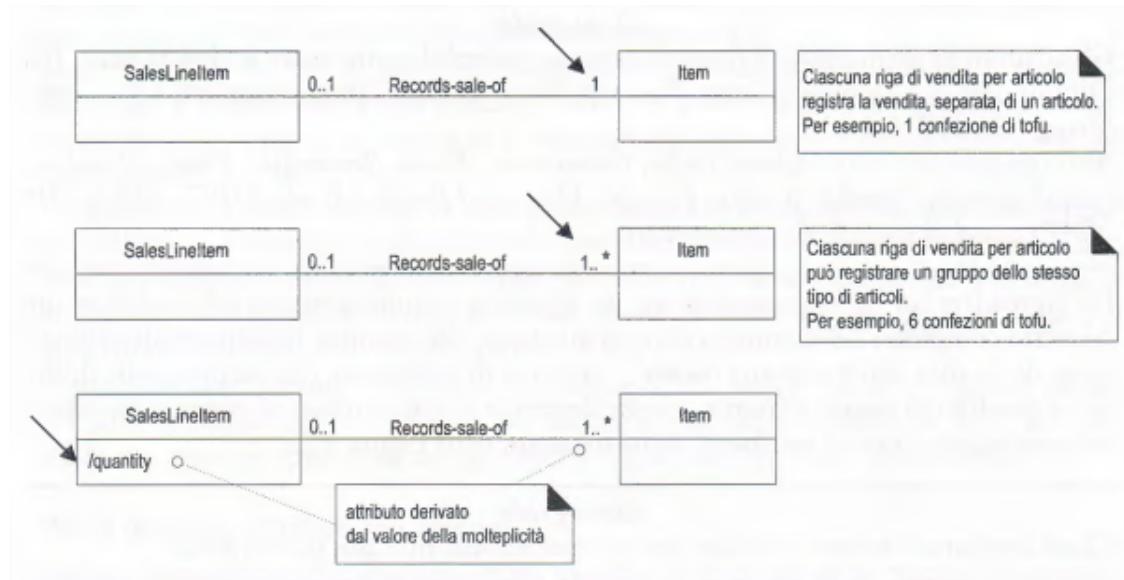


Convenzionalmente, la maggior parte dei modellatori assume che gli attributi hanno una visibilità privata(indicata col "-"), a meno che sia mostrato diversamente, e pertanto normalmente non viene disegnato un simbolo esplicito per la visibilità. Tra le stringhe di proprietà per gli attributi, la più comune è probabilmente `readOnly`, che indica un attributo di sola lettura (una costante). È possibile usare una molteplicità per indicare la presenza opzionale per un valore, o il numero di oggetti che possono appartenere a un attributo collezione.

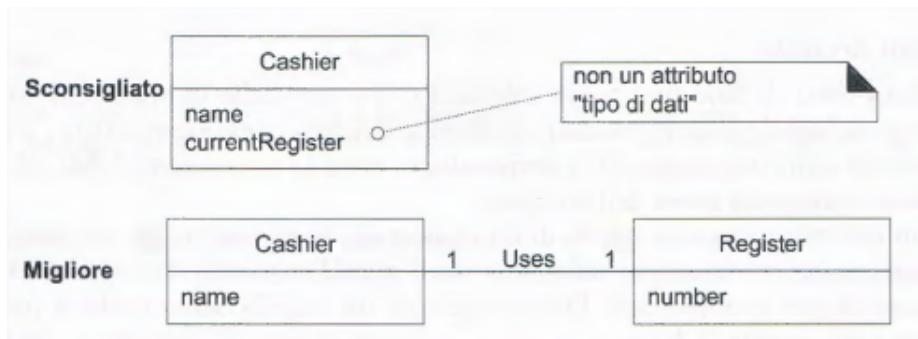
Alcuni modellatori accettano che queste specifiche siano lasciate solo nel modello di dominio; questo però è un metodo dispersivo e che può portare a errori, poiché ci sono persone che tendono a non guardare il modello di dominio in modo dettagliato o come guida per i requisiti (e magari non mantengono nemmeno un modello di dominio). Al contrario, si suggerisce di riportare tutti questi requisiti relativi agli attributi nel Glossario di UP, usato come dizionario dei dati. Magari è stata impiegata un'ora ad abbozzare un modello di dominio con un esperto del dominio; dopodiché è utile impiegare

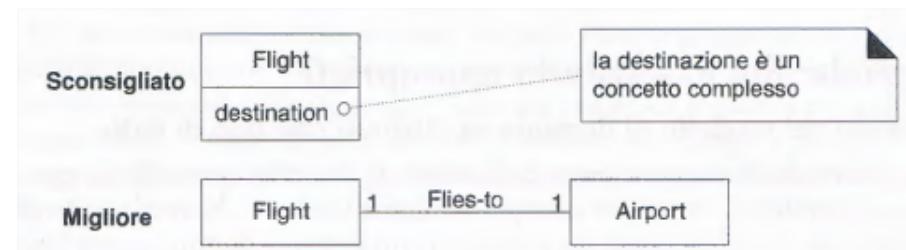
ancora 15 minuti per esaminarlo e trasferire nel Glossario i requisiti implicati per gli attributi. Un'altra alternativa è quella di utilizzare uno strumento che integri i modelli UML con un dizionario dei dati; quindi tutti gli attributi appariranno automaticamente come voci del dizionario.

Si possono avere attributi derivati:



In modo informale, la maggior parte degli attributi dovrebbe essere di un tipo di dato considerato "primitivo", come per esempio numeri e booleani. Normalmente, il tipo di un attributo non dovrebbe essere un concetto complesso del dominio complesso, classi concettuali vanno correlate con associazioni non con attributi:





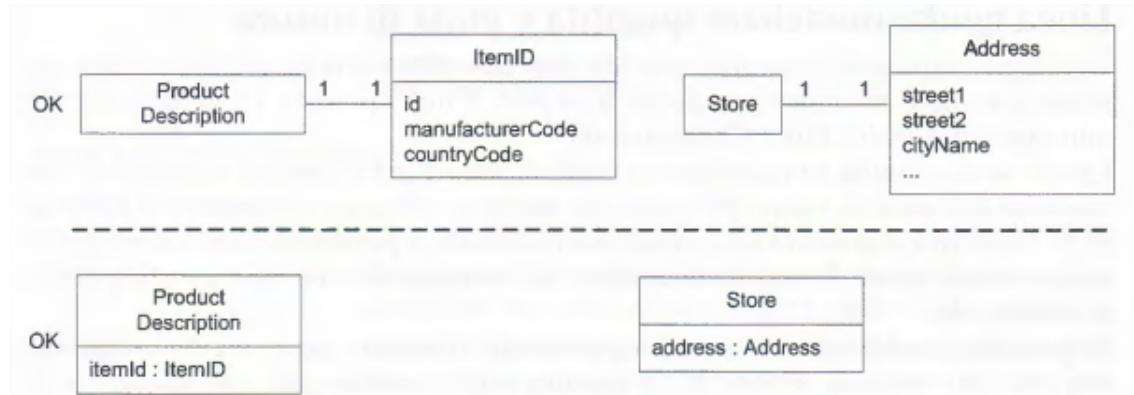
nello specifico un **tipo di dato** è rappresentato da tipi "primitivi" quali numeri, booleani, caratteri, stringhe ed enumerazioni (come Size = small, large). Più precisamente, questo termine in UML indica un insieme di valori la cui identità univoca non è significativa (nel contesto del nostro modello o sistema). In altre parole, i test di uguaglianza non sono basati sull'identità, bensì sul valore. **In altre parole, i test di uguaglianza non sono basati sull'identità, bensì sul valore.**

È possibile definire nuovi tipi di dato:

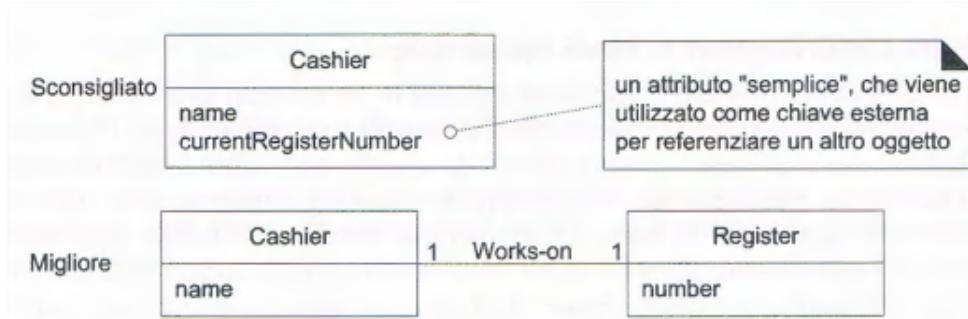
Rappresentare nel modello di dominio come una nuova classe tipo di dato ciò che inizialmente può essere considerato un numero o una stringa se:

- È composto da sezioni separate.
  - per esempio, numero di telefono, nome di persona
- Ci sono operazioni a esso associate, come il parsing o la convalida.
  - numero di previdenza sociale, codice fiscale
- Ha altri attributi.
  - un prezzo promozionale potrebbe avere una data di inizio (effettiva) e una data di fine
- È una quantità con un'unità di misura.
  - l'importo di un pagamento ha un'unità monetaria
- È un'astrazione di uno o più tipi con alcune di queste qualità.
  - il codice identificativo di articolo nel dominio delle vendite è una generalizzazione di tipi come Universal Product Code (UPC) e European Article Number (EAN)

e si ha la seguente rappresentazione:

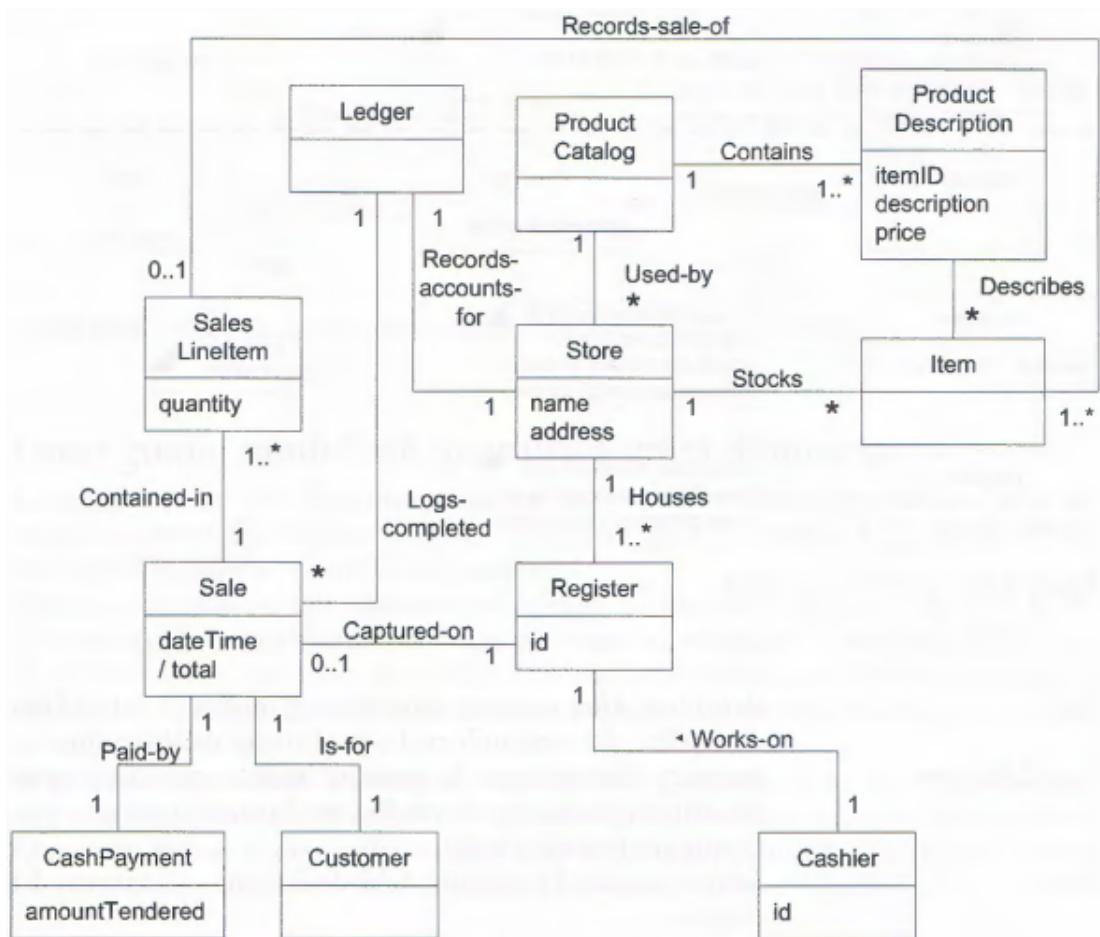


li attributi non dovrebbero essere usati per correlare classi concettuali nel modello di dominio. La violazione più comune di questo principio consiste nell'aggiungere un attributo chiave esterna, come viene fatto normalmente nella progettazione delle basi di dati relazionali per correlare due tipi:



La maggior parte delle quantità numeriche non dovrebbe essere rappresentata come dei semplici numeri. Solitamente è buona pratica associare le unità di misura.

Nel complesso un modello di dominio risulterà:



Anche se paradossalmente sono state dedicate molte pagine alla spiegazione della modellazione di dominio, nelle mani degli esperti lo sviluppo di un modello (parziale, evolutivo) in ciascuna iterazione può durare anche solo 30 minuti, o ancora meno se si utilizzano dei pattern di analisi predefiniti. Nello sviluppo iterativo, si fa evolvere un modello di dominio in modo incrementale su diverse iterazioni. In ciascuna di esse, il modello di dominio è limitato agli scenari precedenti e corrente presi in considerazione, anziché estendersi a un modello "big bang" di tipo a cascata che tenti di catturare dall'inizio tutte le possibili classi concettuali e relazioni.

# Capitolo 6

## Progettazione a Oggetti

Esistono 3 modi per progettare ad oggetti:

1. **codifica**, ovvero progettare mentre si codifica (in Java, C#, ... ), possibilmente insieme a strumenti potenti come il refactoring. Dal modello mentale al codice.
2. **disegno e poi codifica**, ovvero disegnare alcuni diagrammi UML su una lavagna o con uno strumento CASE per UML, poi passare alla codifica usando un IDE "testuale"
3. **solo disegno**,

Alcuni degli scopi della modellazione agile sono ridurre il costo aggiuntivo del disegno e modellare per comprendere e comunicare, anziché per documentare. Le pratiche comprendono l'utilizzo di molte lavagne bianche (dieci in una stanza, non due) o di speciali fogli di plastica bianca aderenti tramite carica statica (che hanno la stessa funzione di una lavagna) per ricoprire ampie aree delle pareti, l'utilizzo di pennarelli, fotocamere digitali e stampanti per catturare "UML come abbozzo", uno dei tre modi per applicare UML. Si ha inoltre la modellazione insieme agli altri e il creare diversi modelli in parallelo (lavorando per tot tempo ad un diagramma su una lavagna e poi un altro po' su un'altra).

Si può usare gli strumenti CASE per UML. Questi strumenti vanno da quelli molto costosi a quelli gratis e open source, e ogni anno la loro utilità migliora. Solitamente si sceglie uno strumento che si integra in un IDE diffuso, che sia in grado di eseguire reverse-engineering (generando diagrammi dal codice) non solo per i diagrammi di classe ma anche per quelli di interazione.

**La modellazione agile alla parete e l'utilizzo di uno strumento CASE per UML integrato in un IDE testuale possono essere pratiche**

**complementari. Sono da provare entrambe durante le varie fasi delle attività.**

Si hanno anche delle linee guida "temporali": per un'iterazione con un time-boxing di tre settimane, si dedichino alcune ore o al massimo un giorno al disegno di UML, all'inizio dell'iterazione.

e si fa modellazione agile, prima di ogni sessione di modellazione successiva, si esegua il reverse-engineering della base di codice crescente in diagrammi UML, si faccia una stampa (magari con un grande plotter), e si faccia riferimento a questi diagrammi nel corso della sessione di abbozzo.

Si hanno due tipi di modelli per gli oggetti:

1. **modelli dinamici** che, come i diagrammi di interazione di UML (diagrammi di sequenza e diagrammi di comunicazione), aiutano a progettare la logica, il comportamento del codice o il corpo dei metodi. Questi diagrammi tendono a essere i più interessanti, difficili e importanti da creare. È nel corso della modellazione a oggetti dinamica che il meccanismo si mette in moto, ovvero si pensa in modo dettagliato e preciso a quali oggetti devono esistere e come questi collaborano attraverso messaggi e metodi. Per la programmazione dinamica usiamo i **diagrammi di interazione**. Si noti che è soprattutto durante la modellazione dinamica che si applicano la progettazione guidata dalle responsabilità e i principi GRASP. UML contiene altri diagrammi dinamici, come i **diagrammi di macchina a stati** e i **diagrammi di attività**
2. **modelli statici** che, come i diagrammi delle classi di UML, aiutano a progettare la definizione dei package, dei nomi delle classi, degli attributi e delle firme (ma non dei corpi) dei metodi. In UML si hanno altri diagrammi statici, come i **diagrammi dei package** e i **diagrammi di deployment**

Prima si dedica del tempo ai modelli dinamici e poi a quelli statici.

Mentre si disegna un diagramma a oggetti di UML, è necessario saper rispondere ad alcune domande fondamentali:

- quali sono le responsabilità dell'oggetto?
- con chi collabora l'oggetto?
- quali design pattern devono essere applicati?

Inoltre la progettazione ad oggetti richiede la conoscenza di:

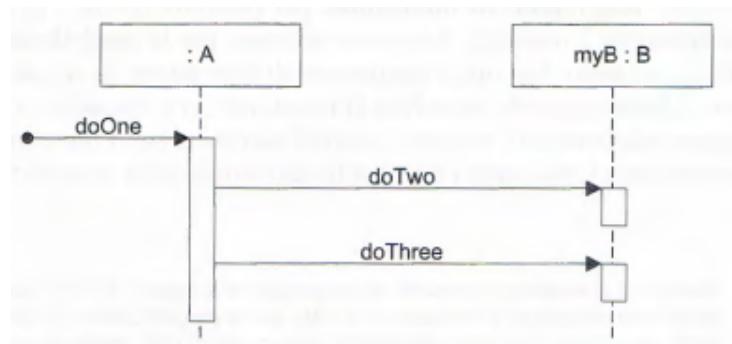
- principi di assegnazione di responsabilità
- design pattern

*Un'altra tecnica di progettazione a oggetti è la CRC, Class, Responsibility, Collaboration.*

## 6.1 Diagrammi di Interazione di UML

UML comprende i **diagrammi di interazione** per illustrare il modo in cui gli oggetti interagiscono attraverso i messaggi. Essi sono utilizzati per la modellazione dinamica degli oggetti. Ce ne sono due tipi:

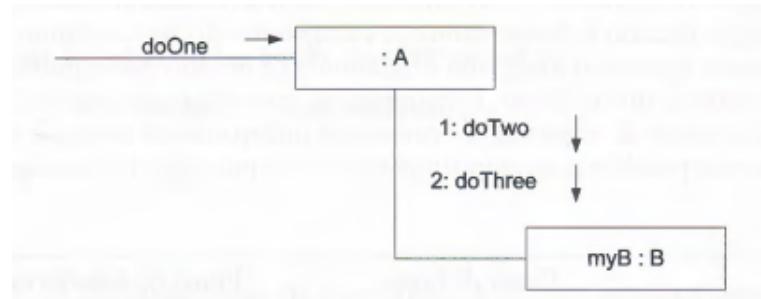
1. **modelli di interazione di sequenza.** Questi diagrammi mostrano le interazioni in un formato "a steccato" in cui ciascun nuovo oggetto viene aggiunto sulla destra:



dove B ha dei metodi `doTwo` e `doThree` mentre la classe A potrebbe essere:

```
public class A {
    private B myB = new B();
    private void doOne() {
        myB.doTwo();
        myB.doThree();
    }
    ...
}
```

2. **modelli di interazione di comunicazione.** Questi diagrammi mostrano le interazioni tra gli oggetti in un formato a grafo o a rete, in cui gli oggetti possono essere posizionati ovunque nel diagramma:



Entrambi possono esprimere interazioni simili e si può avere il **diagramma di interazione generale**, introdotto con UML2; esso fornisce una visione generale di come sono correlati un insieme di diagrammi di interazione, in termini di logica e di processi e flussi.

Normalmente si preferiscono i diagrammi di sequenza a causa del loro maggior potere notazionale. Inoltre i diagrammi di sequenza hanno altri vantaggi rispetto a quelli di comunicazione, inoltre la specifica di UML è maggiormente incentrata su quelli di sequenza. Questo comporta un supporto migliore degli strumenti e si hanno più opzioni di notazione. Inoltre con i diagrammi di sequenza è più facile vedere la sequenza "call-flow", *chiamata-flusso*, poiché è sufficiente leggerli dall'alto verso il basso. Con i diagrammi di comunicazione è necessario leggere i numeri di sequenza, come "1: " e "2: ". Pertanto i diagrammi di sequenza sono ottimi per la documentazione o per leggere facilmente una sequenza call-flow ottenuta con il reverse engineering, generata dal codice sorgente con uno strumento UML.

Senza approfondire troppo pro e contro possiamo basarci su questa tabella riassuntiva:

Tipo	Punti di forza	Punti di debolezza
Diagramma di sequenza	Mostra chiaramente la sequenza o l'ordinamento temporale dei messaggi. Opzioni di notazione numerose e dettagliate.	Costringe a estendersi verso destra quando si aggiungono nuovi oggetti; consuma lo spazio orizzontale.
Diagramma di comunicazione	Economizza lo spazio; flessibilità nell'aggiunta di nuovi oggetti nelle due dimensioni.	Più difficile vedere la sequenza dei messaggi. Meno opzioni di notazione.

Vediamo un piccolo esempio di un diagramma di sequenza. Si hanno:

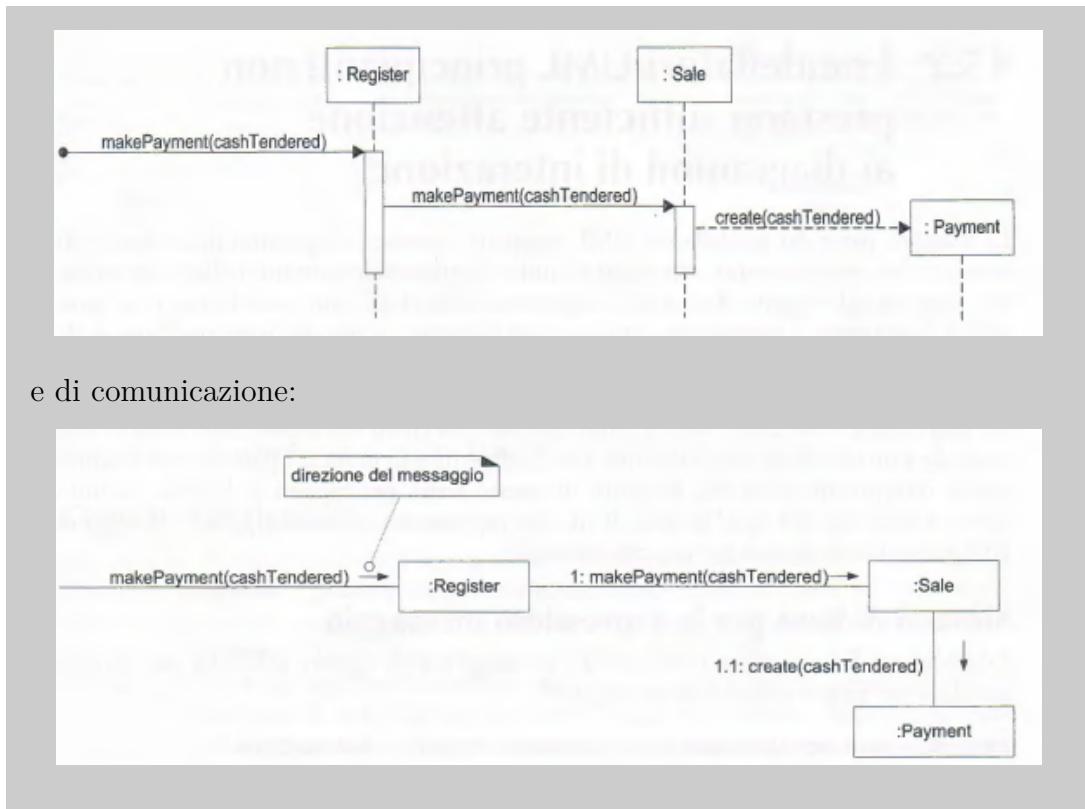
1. il messaggio makePayment viene inviato a un'istanza di Register. Il mittente non è identificato
2. l'istanza di Register invia il messaggio makePayment a un'istanza di Sale
3. l'istanza di Sale crea un'istanza di Payment.

Si potrebbe avere il seguente codice:

```
public class sale {
    private Payment payment;

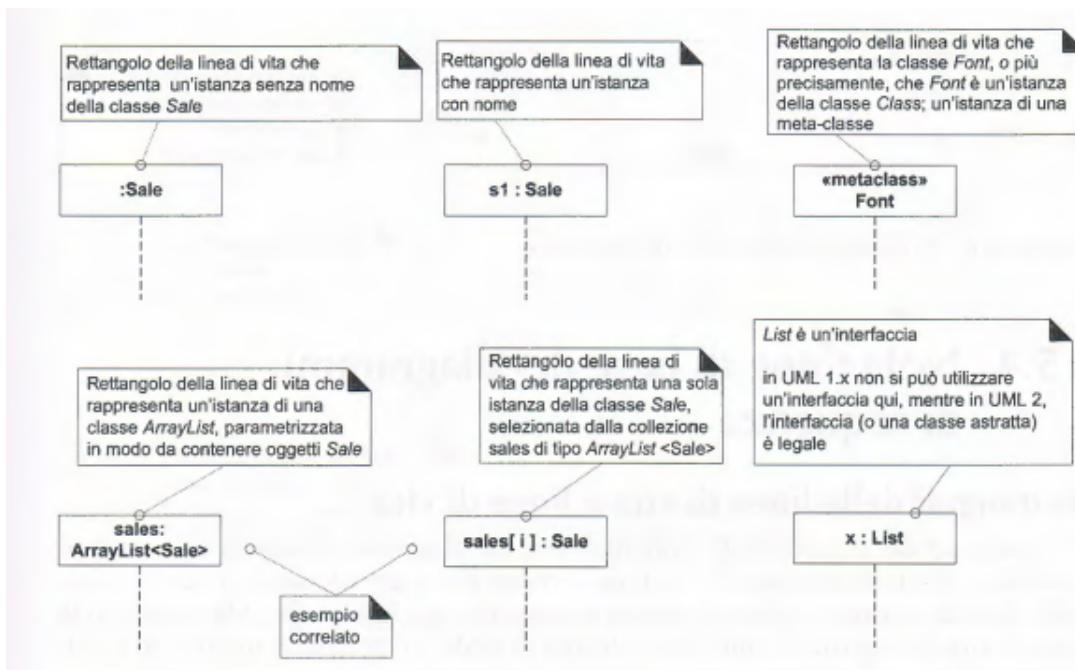
    public void makePayment (Money cashTendered) {
        payment = new Payment( cashTendered );
        ...
    }
    ...
}
```

con il seguente diagramma di sequenza:



### 6.1.1 Notazione in UML

In UML, nei diagrammi di interazione, si hanno le **linee di vita**, *lifeline box* rappresentate mediante dei rettangoli. La loro definizione precisa in UML è sottile, ma informalmente essi rappresentano i partecipanti all'interazione, ovvero delle parti correlate definite nel contesto di un qualche diagramma strutturale, per esempio di un diagramma delle classi. Non è molto preciso dire che il rettangolo della linea di vita equivale a un'istanza di una classe ma, informalmente e in pratica, i partecipanti saranno spesso interpretati come tali:



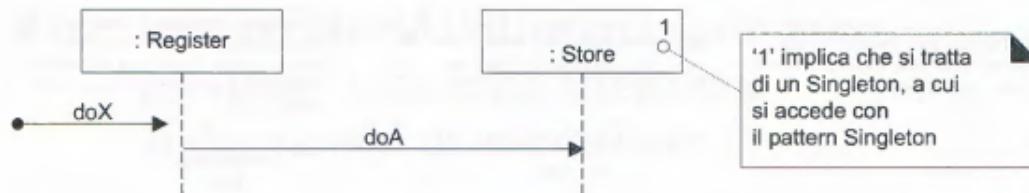
I diagrammi di interazione mostrano dei messaggi tra gli oggetti e si ha una sintassi ben definita:

*return = message(parameter : parameterType) : returnType*

Le parentesi vengono solitamente escluse se non ci sono parametri, anche se sono permesse. Le informazioni sul tipo possono essere escluse se ovvie o poco importanti. Vediamo qualche esempio:

```
initialize(code)
initialize
d = getProductDescription(id)
d = getProductDescription(id:ItemID)
d = getProductDescription(id:ItemID) : ProductDescription
```

Nel mondo dei design pattern OO, ce n'è uno che è particolarmente diffuso, chiamato il **pattern Singleton**. Una delle sue implicazioni è che viene istanziata una sola istanza della classe, mai due. In altre parole, è un'istanza "singleton" e un tale oggetto è contrassegnato da un 1 nell'angolo superiore destro del rettangolo della linea di vita.



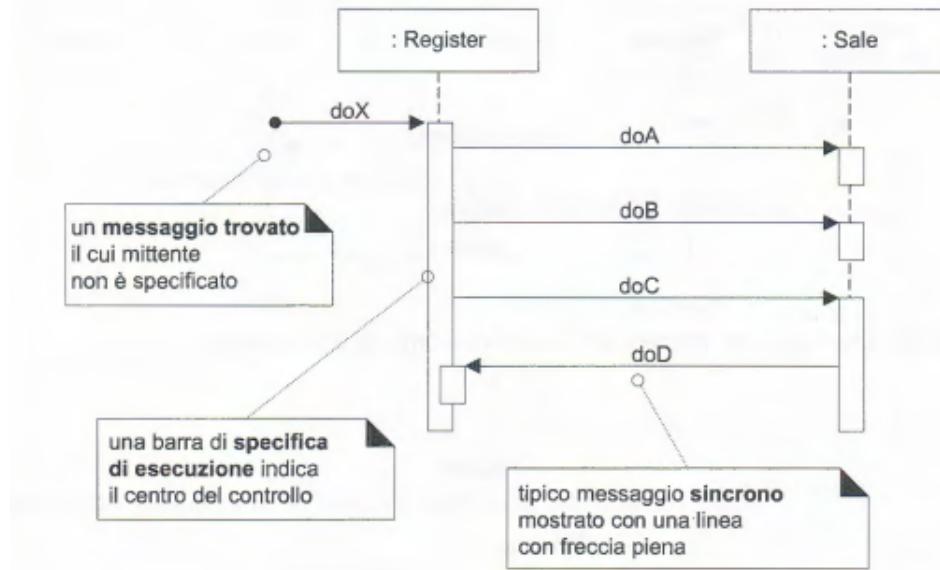
**Figura 15.6** Singleton nei diagrammi di interazione.

### 6.1.2 Notazione dei Diagrammi di Sequenza

Diversamente dai diagrammi di comunicazione, nei diagrammi di sequenza i rettangoli delle linee di vita comprendono una linea verticale che si estende sotto di esse (si tratta delle linee di vita vere e proprie). Anche se quasi tutti gli esempi di UML mostrano le linee di vita tratteggiate (a causa dell'influenza di UML 1), di fatto la specifica di UML 2 afferma che possono essere continue oppure tratteggiate.

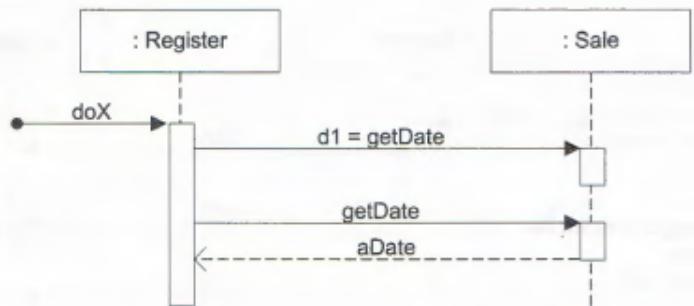
Ogni messaggio (di solito sincrono) tra gli oggetti è rappresentato da un'espressione messaggio mostrata su una linea continua con una freccia piena tra le linee di vita verticali. L'ordinamento del tempo è organizzato dall'alto verso il basso delle linee di vita. Il messaggio iniziale (in alto a sinistra) è chiamato in UML un messaggio trovato, ed è mostrato con un cerchio pieno di apertura; ciò implica che il mittente non sarà specificato, non è noto, o che il messaggio proviene da una sorgente qualsiasi. Tuttavia, per convenzione un team o uno strumento possono ignorare questa indicazione e utilizzare invece una normale linea di messaggio senza il cerchio, intendendo per convenzione che si tratta di un messaggio trovato. I diagrammi di sequenza possono anche mostrare il centro del controllo (informalmente, in una normale chiamata bloccante, che l'operazione è sullo stack delle chiamate) utilizzando una barra di specifica di esecuzione (precedentemente chiamata barra di attivazione o semplicemente attivazione in UML 1).

La barra è opzionale ma è comune disegnarla con un CASE per UML e meno comunque con un abbozzo alla parete:

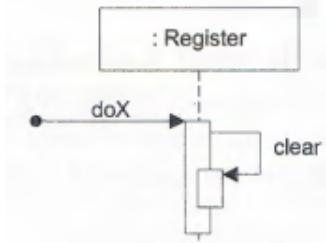


Ci sono due modi per mostrare il risultato di ritorno per un messaggio:

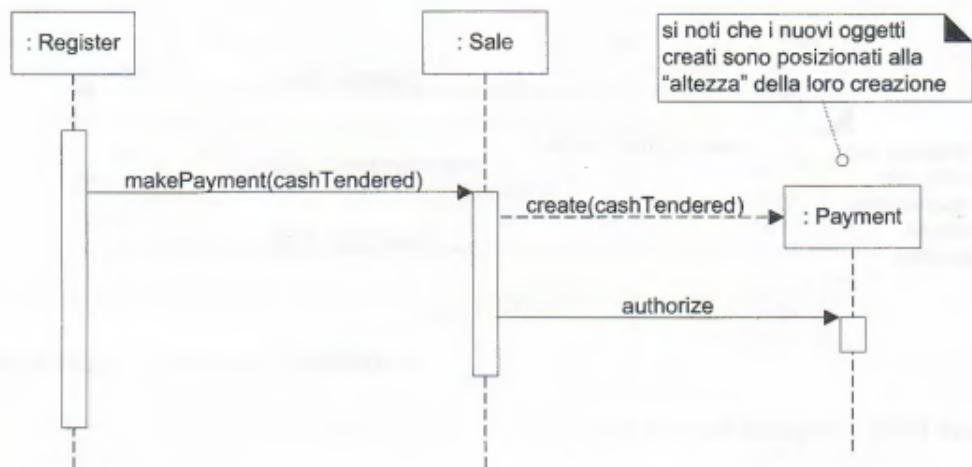
1. utilizzare per il messaggio la sintassi  $returnVar = message(parameter)$ .  
Questo metodo è preferibile
2. utilizzare una linea di messaggio di risposta (o ritorno) alla fine di una barra di specifica di esecuzione



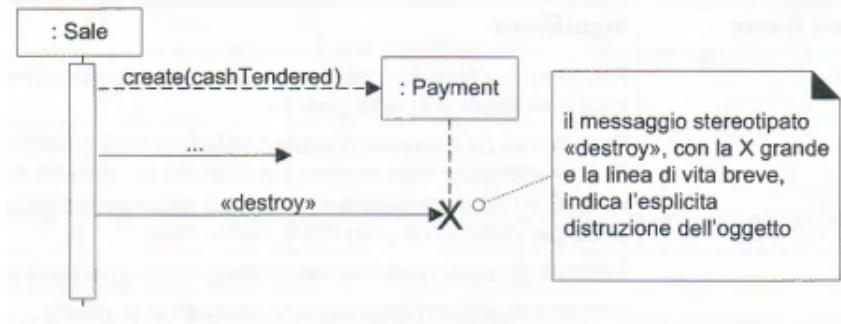
È possibile mostrare un messaggio inviato da un oggetto a se stesso utilizzando una barra di specifica di esecuzione annidata (rappresentando il `this`):



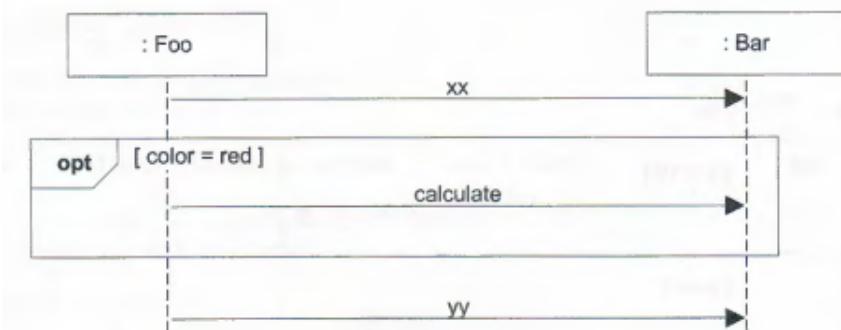
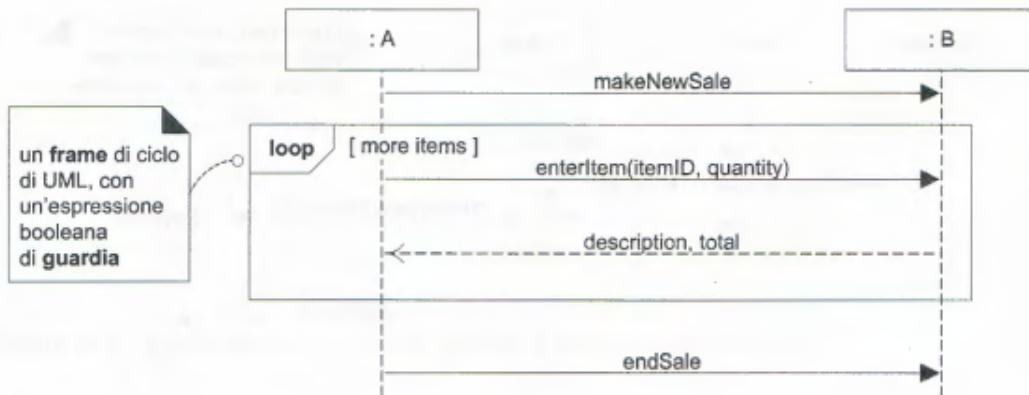
Vediamo la notazione per creare oggetti in UML. La freccia è piena se si tratta di un messaggio regolare sincrono (per esempio, per indicare l'invocazione di un costruttore Java), oppure non piena se si tratta di una chiamata asincrona. Non è obbligatorio chiamare il messaggio `create` (qualunque cosa è permessa), ma si tratta di un idioma di UML. [interpretazione tipica (in linguaggi come Java o C#) di un messaggio `create` su una linea tratteggiata con una freccia piena è "invocare l'operatore `new` e chiamare il costruttore"]:

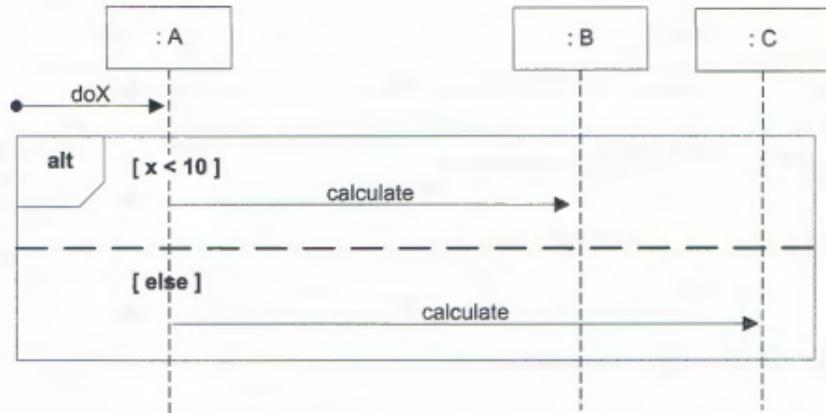


In alcune circostanze è opportuno mostrare una distruzione esplicita di un oggetto; per esempio, quando si utilizza il C++ (che non ha garbage collection automatica), o quando si vuole indicare specificamente che un oggetto non è più utilizzabile (come una connessione chiusa a una base di dati). La notazione delle linee di vita di UML fornisce un modo per esprimere questa distruzione:



Come supporto ai costrutti condizionali e di ciclo (tra le altre cose), UML utilizza i **frame**. frame sono regioni o frammenti dei diagrammi; hanno un operatore o un'etichetta (come loop) e una guardia (clausola condizionale):

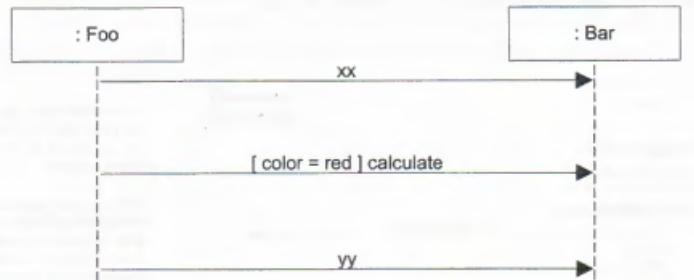




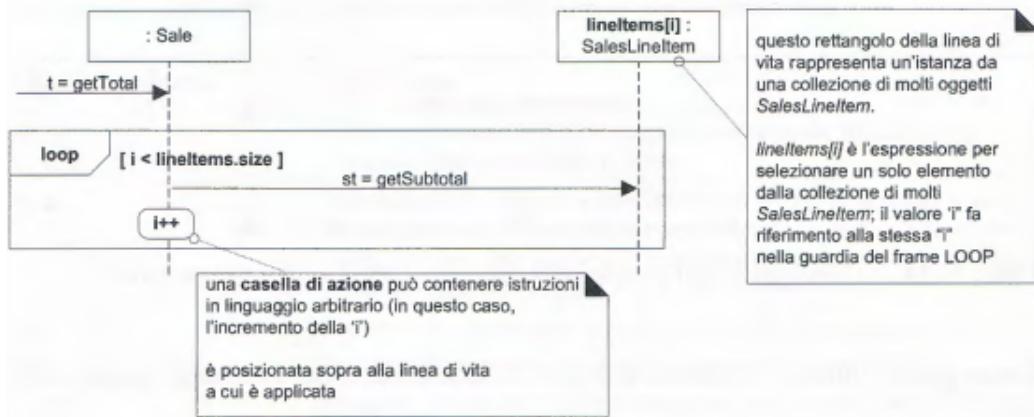
in tabella si hanno gli operatori più comuni per i frame:

Operatore frame	Significato
alt	Frammento alternativo per logica condizionale mutuamente esclusiva espressa nelle guardie.
loop	Frammento da eseguire ripetutamente finché la guardia è vera. Si può anche scrivere <i>loop(n)</i> per indicare un ciclo da ripetere <i>n</i> volte. C'è una discussione in corso sul potenziamento della specifica per definire un ciclo <i>FOR</i> , come <i>loop(i, 1, 10)</i> .
opt	Frammento opzionale che viene eseguito se guardia è vera.
par	Frammenti paralleli che vengono eseguiti in parallelo.
region	Regione critica all'interno della quale può essere eseguito solo un thread.

La vecchia notazione UML 1.x per i messaggi condizionali singoli nei diagrammi di sequenza non è più legale in UML 2, ma è così semplice che probabilmente resterà diffusa ancora per anni, soprattutto durante l'abbozzo:



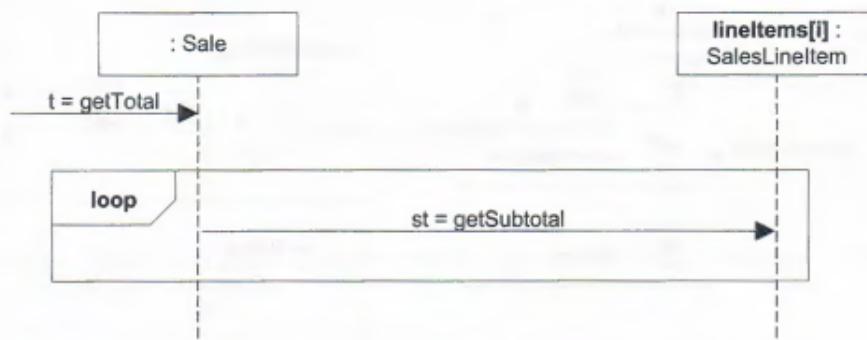
Abbiamo anche una tecnica particolare per mostrare l'iterazione su una collezione:



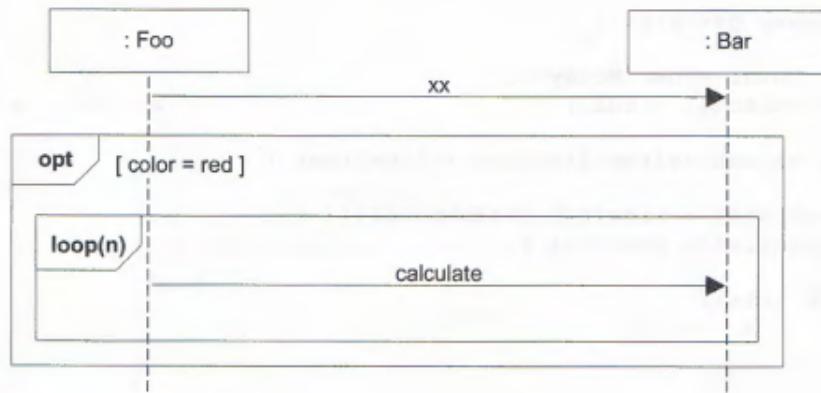
L'espressione selettore è utilizzata per selezionare un solo oggetto da un gruppo. Una linea di vita partecipante deve rappresentare un solo oggetto, non una collezione. Vediamo cosa rappresenterebbe in codice:

```
public class Sale {
    private List<SalesLineitem> lineitems =
        new ArrayList<SalesLineitem>();
    public Money getTotal() {
        Money total = new Money();
        Money subtotal = null;
        for ( SalesLineitem lineitem : lineitems) {
            subtotal = lineitem.getSubtotal();
            total.add( subtotal );
        }
        ...
    }
}
```

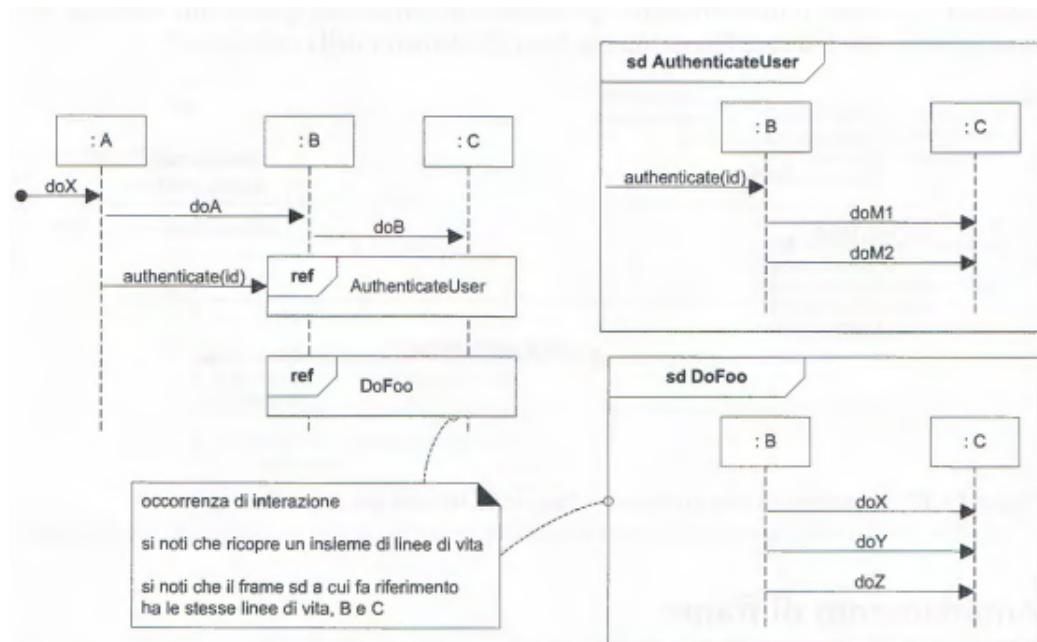
Si ha anche una variante semplificata senza dettagli:



I frame possono essere annidati:



Si possono inoltre correlare i diagrammi di interazione. Una occorrenza di interazione (chiamata anche un uso di interazione) è un riferimento a un’interazione all’interno di un’altra interazione. È utile, per esempio, quando si vuole semplificare un diagramma e decomporne una porzione in un altro diagramma, oppure nel caso in cui ci sia un’occorrenza di interazione riutilizzabile. Gli strumenti UML ne traggono vantaggio, grazie alla loro utilità nel correlare e collegare i diagrammi:



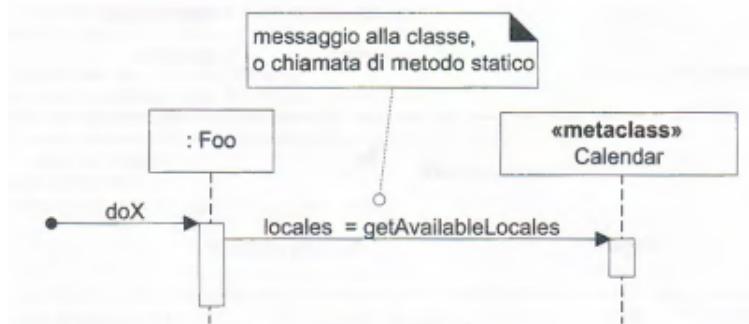
Si hanno due frame correlati:

1. un frame attorno a un intero diagramma di sequenza, etichettato con il tag **sd** e con un nome, per esempio *AuthenticateUser*
2. un frame con tag **ref**, chiamato un riferimento, che referenzia un altro diagramma di sequenza tramite il nome; è la vera e propria occorrenza di interazione

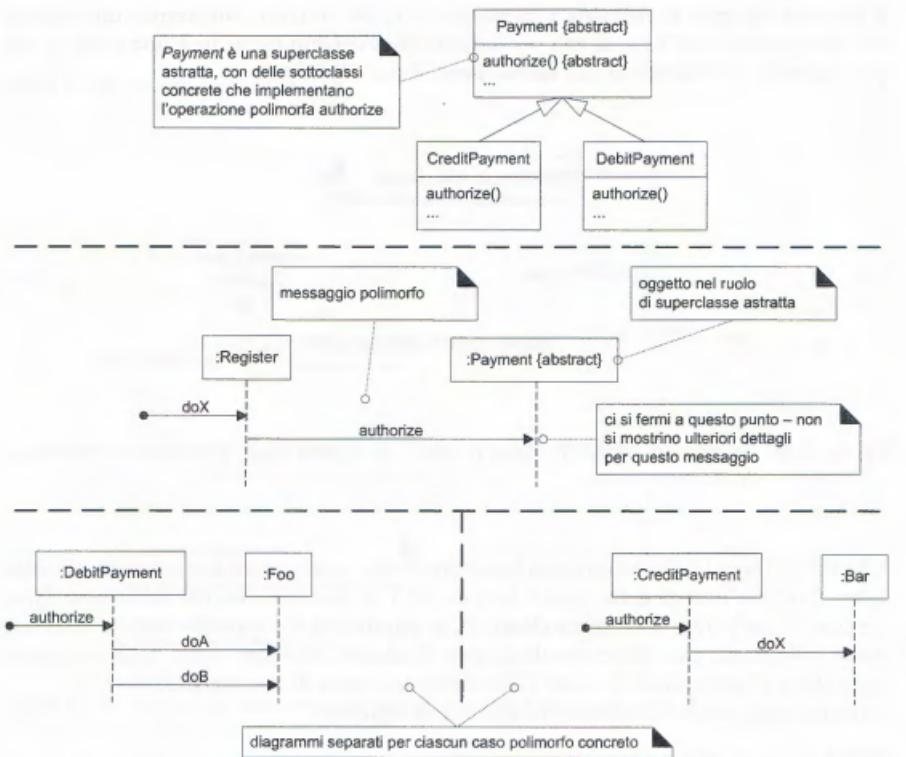
I diagrammi di interazione generale contengono anch'essi un insieme di frame riferimento (occorrenze di interazione). Questi diagrammi organizzano i riferimenti in una struttura più ampia di logica, processi e flussi.

*Qualsiasi diagramma di sequenza può essere racchiuso in un frame sd, per dargli un nome. Si racchiuda un diagramma in un frame e gli si assegna un nome solo quando si vuole fare riferimento allo stesso utilizzando un frame ref.*

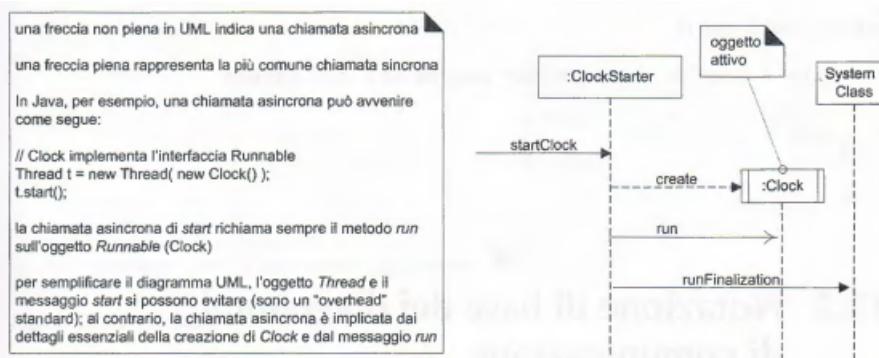
È possibile mostrare le chiamate a un metodo di classe, o statico, utilizzando un'etichetta nel rettangolo di una linea di vita per indicare che l'oggetto ricevente è una classe o, più precisamente, un'istanza di una meta-classe. Infatti le classi Class e Type sono meta-classi, il che significa che le rispettive istanze sono esse stesse delle classi:



Vediamo come rappresentare un altro aspetto fondamentale della progettazione orientata a oggetti, il **polimorfismo**. Un approccio consiste nell'utilizzare più diagrammi di sequenza: uno che mostra il messaggio polimorfo all'oggetto della superclasse astratta o dell'interfaccia, insieme a diagrammi di sequenza separati che descrivono nel dettaglio ciascun caso polimorfo, ciascuno che inizia con un messaggio trovato polimorfo:



Una **chiamata di messaggio asincrono** non attende una risposta; non è bloccante. Vengono utilizzate negli ambienti multi-thread come .NET e Java per creare e iniziare nuovi thread di esecuzione. La notazione UML per le chiamate asincrone è un messaggio con una freccia con la punta non piena; le chiamate sincrone regolari (bloccanti) sono mostrate con una freccia a punta piena:

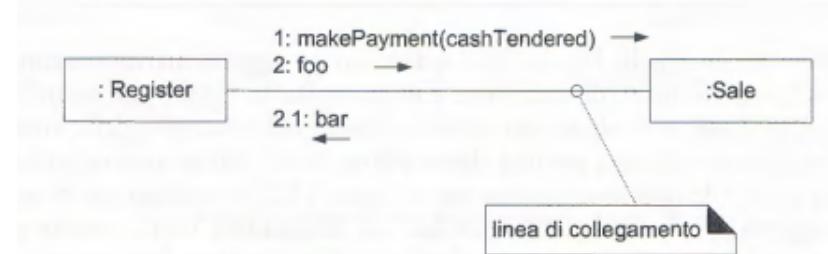


Si ha un **oggetto attivo** quando ciascuna istanza è eseguita nel proprio thread di esecuzione e lo controlla. In UML, può essere mostrato con delle doppie linee verticali sui lati sinistro e destro del rettangolo della linea di

vita. La stessa notazione è utilizzata per una **classe attiva**, le cui istanze sono oggetti attivi.

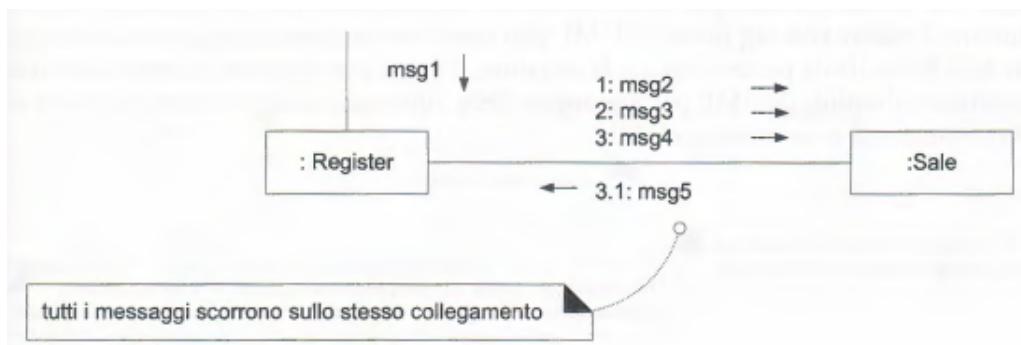
### 6.1.3 Notazione nei Diagrammi di Comunicazione

Iniziamo con i **collegamenti**. Un collegamento è un percorso di connessione tra due oggetti, che indica che è possibile una qualche forma di navigazione e di visibilità tra gli oggetti. Più formalmente, un collegamento è un'istanza di un'associazione:

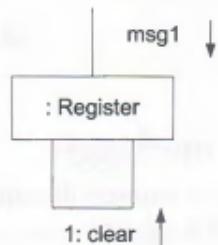


i noti che su uno stesso collegamento possono scorrere più messaggi e messaggi in entrambe le direzioni. Non c'è una linea di collegamento per ciascun messaggio; tutti i messaggi scorrono sulla stessa linea, come in una strada su cui è consentito il traffico in entrambe le direzioni.

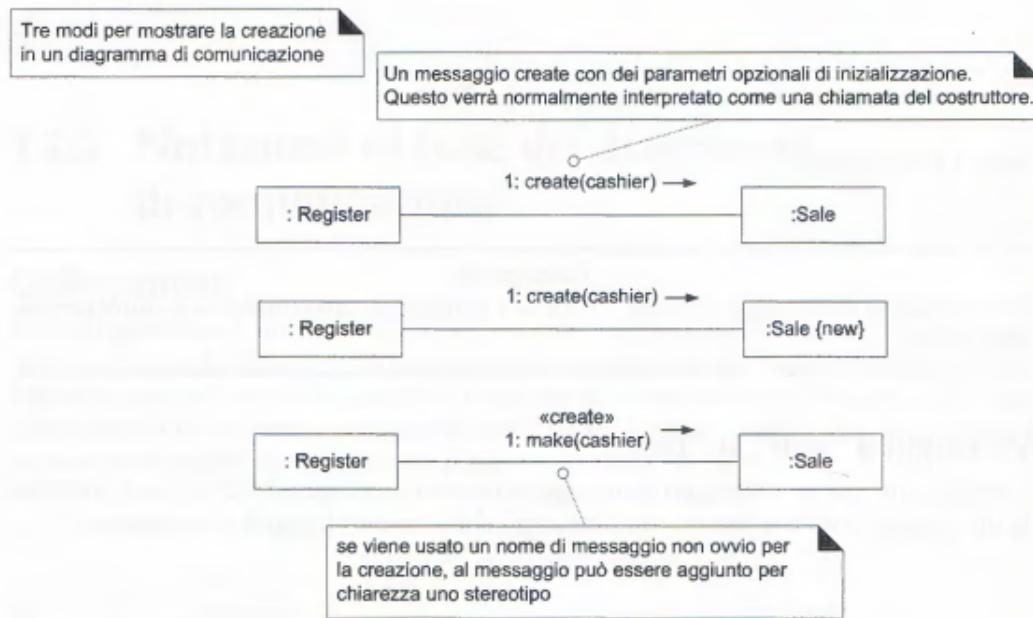
Passiamo ai **messaggi**. Ogni messaggio tra oggetti è rappresentato da un'espressione messaggio e da una piccola freccia che indica la direzione del messaggio. Lungo un collegamento possono scorrere più messaggi. Viene aggiunto un numero di sequenza per mostrare l'ordine sequenziale dei messaggi nel thread corrente, per comodità non si numera il messaggio iniziale:



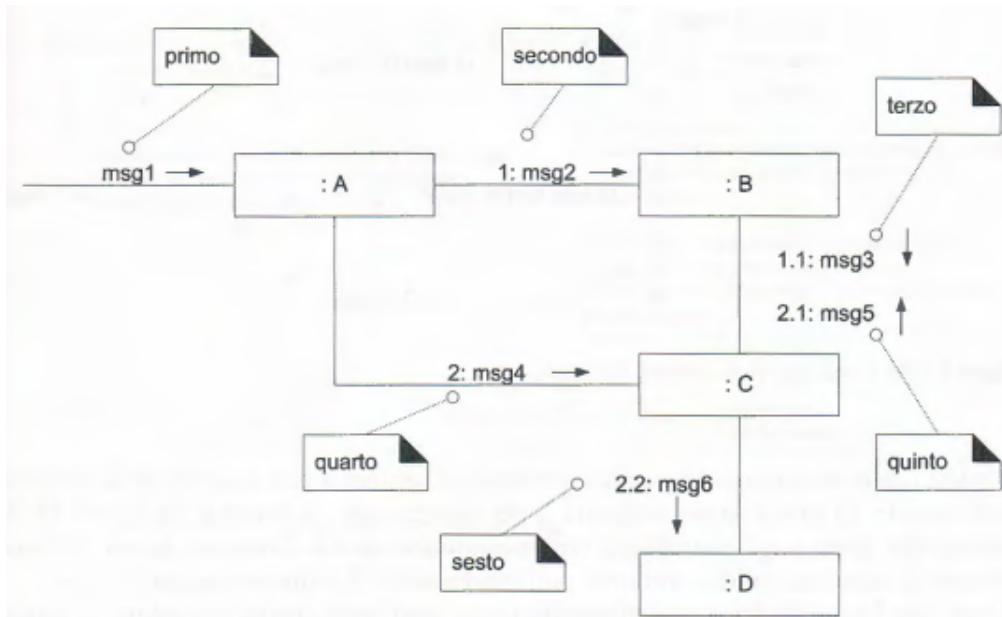
È possibile inviare un messaggio da un oggetto a se stesso. Ciò è illustrato da un collegamento a se stesso, con i messaggi che scorrono lungo il collegamento:



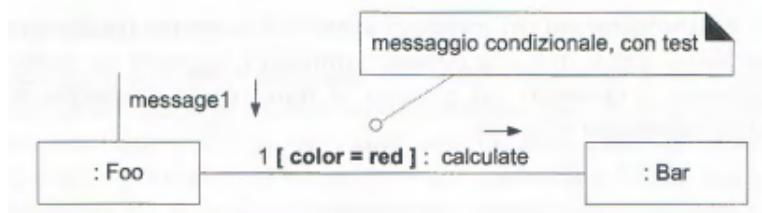
Qualsiasi messaggio può essere utilizzato per creare un'istanza, ma la convenzione in UML prevede di utilizzare per questo scopo un messaggio chiamato *create*, anche se alcuni usano *new*. Se viene usato un altro nome di messaggio (meno ovvio), il messaggio può essere annotato con uno stereotipo UML, come `!create`. Il messaggio *create* può comprendere dei parametri, che indicano il passaggio di valori iniziali. Ciò indica, per esempio, una chiamata di un costruttore con dei parametri in Java. Inoltre, il valore con tag *new* di UML può essere eventualmente aggiunto al rettangolo della linea di vita per evidenziare la creazione. I valori con tag sono un meccanismo di estensione flessibile di UML per aggiungere delle informazioni significative dal punto di vista semantico a un elemento UML:



L'**ordine dei messaggi** è illustrato con i numeri di sequenza, per comodità il primo non è numerato. Inoltre l'ordine e l'annidamento dei messaggi seguenti è mostrato mediante uno schema di numerazione legale in cui ai messaggi annidati è aggiunto un numero alla fine. L'annidamento è denotato dal preporre il numero del messaggio in entrata al numero del messaggio in uscita:

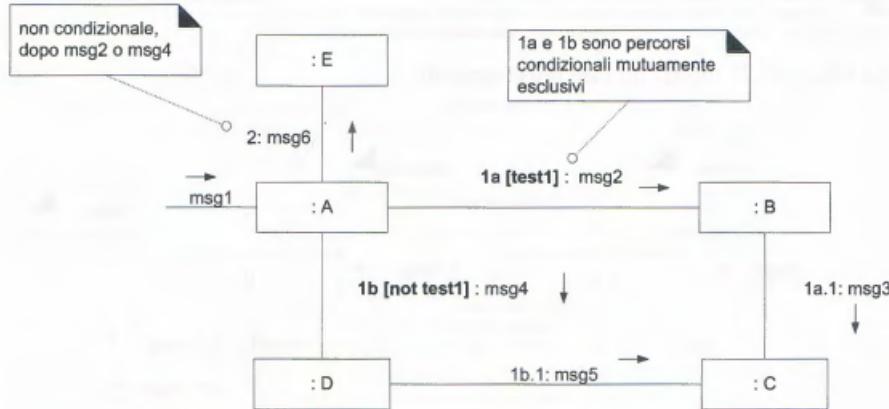


Un **messaggio condizionale** viene indicato facendo seguire al numero di sequenza una clausola condizionale tra parentesi quadre, simile alla clausola per una iterazione. Il messaggio viene inviato solo se la clausola viene valutata a true:

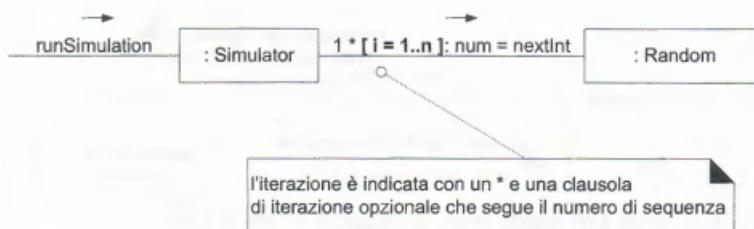


Si possono avere **percorsi condizionali mutualmente esclusivi**. In questo caso occorre modificare le espressioni di sequenza con una lettera di percorso condizionale. La prima lettera utilizzata è, per convenzione, la lettera "a". quindi dopo msg1 potrebbero venire eseguiti 1a o 1b. Entrambi hanno 1 come numero di sequenza, poiché entrambi potrebbero essere il primo

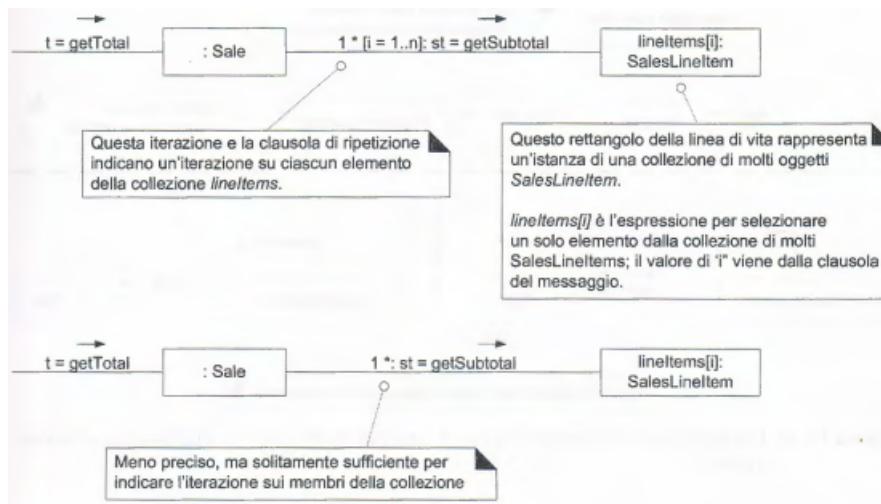
messaggio interno. Si noti che i numeri dei messaggi annidati successivi sono ancora preceduti, in modo coerente, dal numero di sequenza del rispettivo messaggio entrante. Pertanto 1b.1 è un messaggio annidato a 1b.



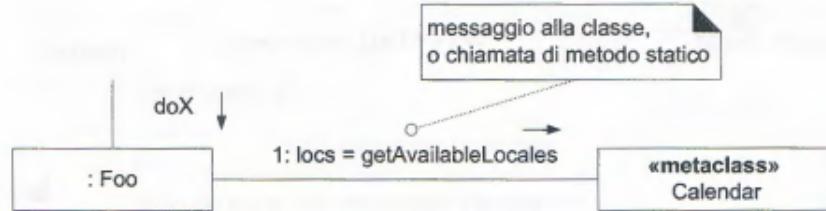
vediamo l'**iterazione**. Se i dettagli della clausola di iterazione non sono importanti per il modellatore, è possibile utilizzare un semplice "\*":



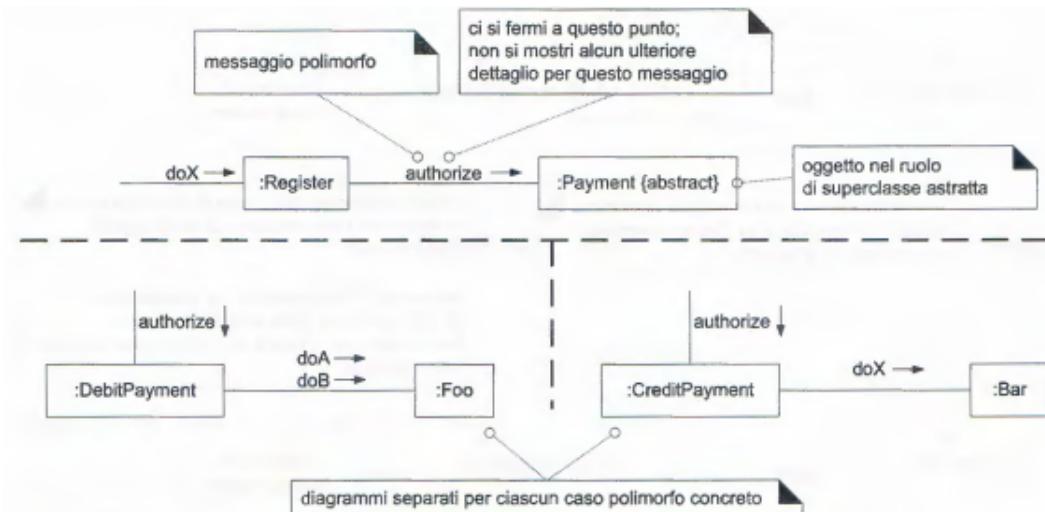
Per **iterare su una collezione** non si ha alcuna convenzione UML ufficiale ma si usa:



Per invocare una classe:



Come nel caso dei diagrammi di sequenza, è possibile utilizzare più diagrammi di comunicazione per mostrare, separatamente, ciascun **caso polimorfo concreto**:



Come nei diagrammi di sequenza, le chiamate sincrone sono mostrate con una freccia con la punta piena e quelle asincrone con una freccia con la punta non piena:

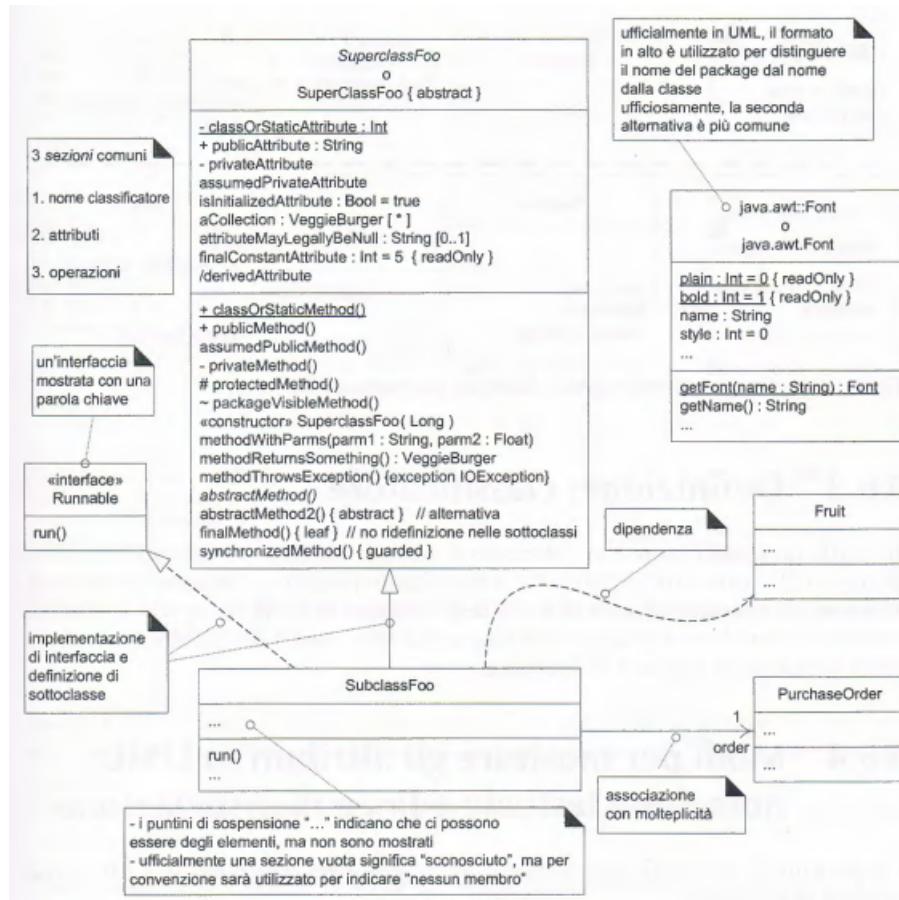


# Capitolo 7

## Diagrammi delle Classi

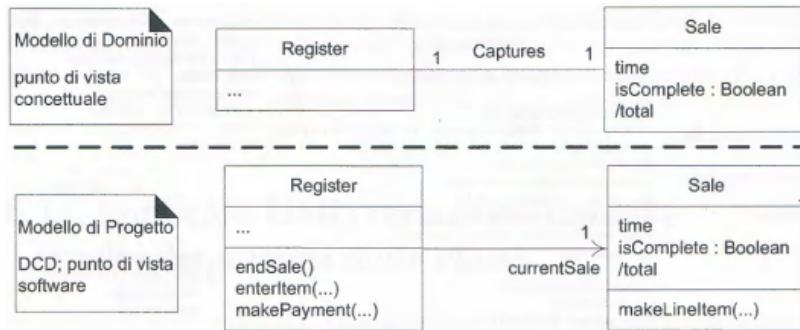
UML comprende i **diagrammi delle classi** per illustrare le classi, le interfacce e le relative associazioni. Essi sono utilizzati per la modellazione statica degli oggetti. Visto che la maggior parte degli elementi sono stati già analizzati partiamo da una figura che racchiude la gran parte delle notazioni, anche quelle opzionali come i modificatori +/- di visibilità (ricordiamo che se non indicata per convenzione si ha che un attributo è privati), i parametri e le sezioni:

## Capitolo 7. Diagrammi delle Classi

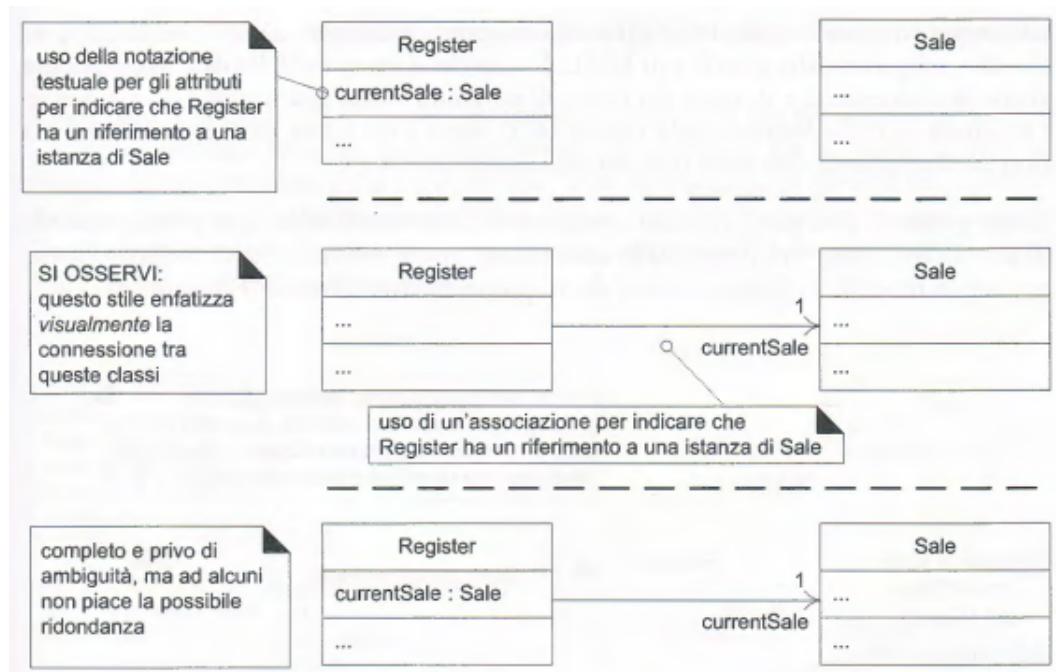


Da un punto di vista concettuale i diagrammi delle classi possono essere utilizzati per visualizzare un modello di dominio. Per la discussione occorre anche un termine univoco che chiarisca quando un diagramma delle classi è utilizzato da un punto di vista software o di progetto. A questo scopo, un termine comune nella modellazione è diagramma delle classi di progetto **DCD**, *Design Class Diagram*. In UP, l'insieme di tutti i DCD fa parte del Modello di Progetto. Altre parti del Modello di Progetto comprendono i diagrammi di interazione e i diagrammi dei package di UML:

## Capitolo 7. Diagrammi delle Classi



In UML, un **classificatore** è un *elemento di modello che descrive caratteristiche comportamentali e strutturali*. I classificatori possono essere **specializzati**. Essi sono una generalizzazione di molti degli elementi di UML, comprese le classi, le interfacce, i casi d'uso e gli attori. Nei diagrammi delle classi, i due classificatori più comuni sono le **classi regolari** e le **interfacce**. Gli attributi di un classificatore sono mostrati in vari modi:



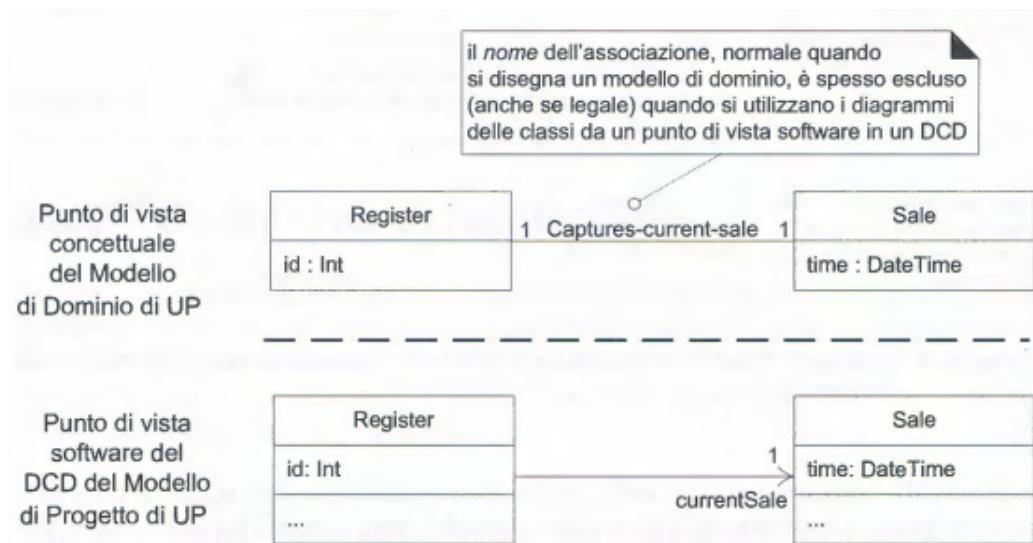
- con un'annotazione testuale, per esempio `currentSale : Sale`. Il formato completo però sarebbe:

*visibility name: type multiplicity = default property-string*

## Capitolo 7. Diagrammi delle Classi

---

- con una linea di associazione, con una **freccia di navigabilità** rivolta dall'oggetto sorgente all'oggetto destinazione, che indica che l'oggetto sorgente ha un attributo di tipo oggetto destinazione. Inoltre viene indicata la molteplicità solo vicino alla destinazione, con la solita notazione. Si ha un **nome di ruolo** solo all'estremità vicina alla destinazione, per indicare il nome dell'attributo e non si ha un nome per l'associazione. D'altra parte, quando si utilizzano i diagrammi delle classi per un modello di dominio, si mostrino i nomi delle associazioni, ma si evitino le frecce di navigazione, poiché un modello di dominio non è da un punto di vista software:



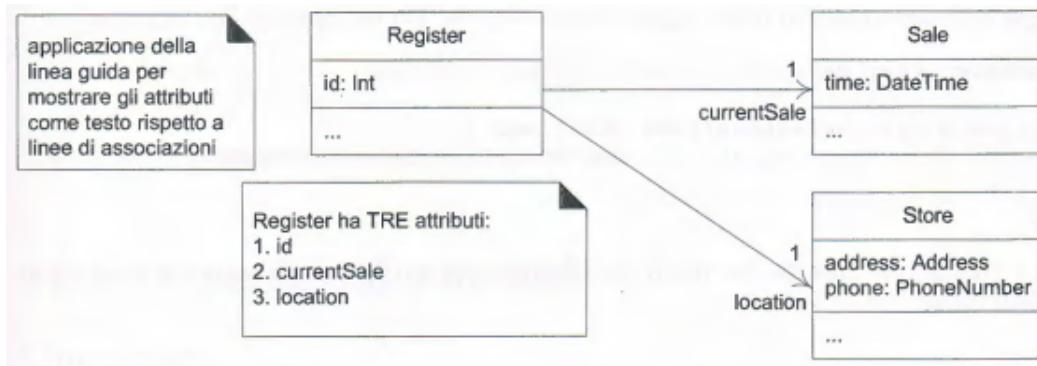
- con entrambe le notazioni sopra indicate

Si utilizza la notazione testuale per gli attributi per gli oggetti tipi di dato, e la notazione delle linee di associazione per gli altri attributi.

## Capitolo 7. Diagrammi delle Classi

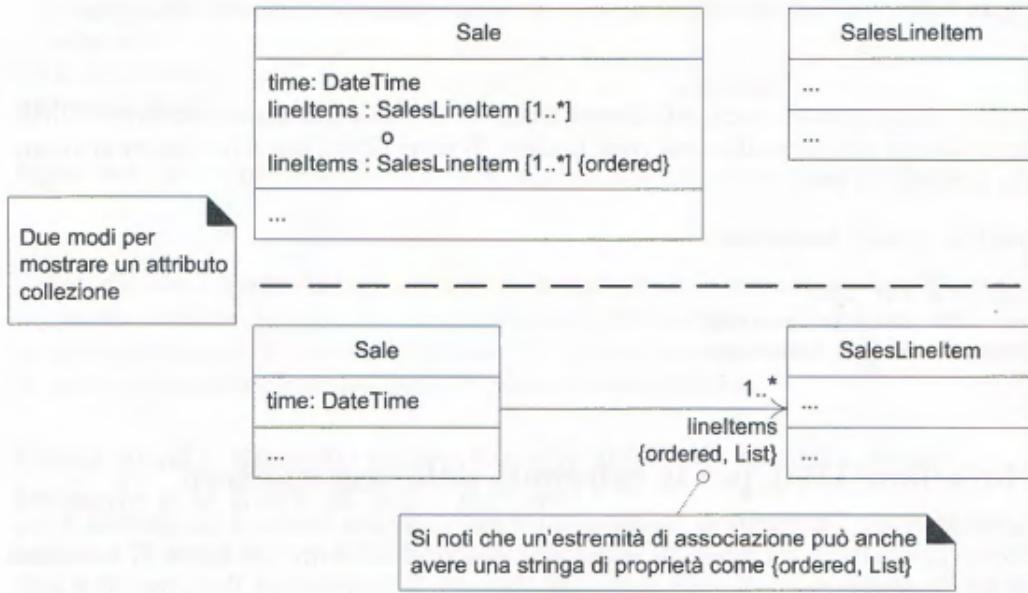
---

Da un punto di vista semantico sono equivalenti, ma mostrare una linea di associazione verso il rettangolo di dà un risalto visuale:



```
public class Register {  
    private int id;  
    private Sale currentSale;  
    private Store location;  
    ...  
}
```

L'estremità di un'associazione può avere una freccia di navigabilità, e può anche comprendere un nome di ruolo opzionale (ufficialmente, un nome di estremità di associazione) per indicare il nome dell'attributo. Naturalmente l'estremità dell'associazione può anche mostrare un valore di molteplicità. È possibile usare una stringa di proprietà come {ordered} o {ordered, List}. La prima {ordered} è una parola chiave definita da UML che implica che gli elementi della collezione sono ordinati. Un'altra parola chiave correlata è {unique}, che implica un insieme di elementi distinti. La parola chiave {List} mostra come UML consente anche le parole chiave definite dall'utente. È stato usato {List} per indicare che l'attributo collezione lineItems sarà implementato con un oggetto che implementa l'interfaccia *List*. Si hanno due modi per rappresentare un attributo collezione, con anche l'indicazione di proprietà opzionali, come {ordered}:



Si ha una sintassi particolare per le operazioni:

*visibility name (parameter-list) {property-string}*

e manca l'elemento per il tipo di ritorno, che è stato introdotto in UML2:

*visibility name (parameter-list) : return-type {property-string}*

**Al contrario degli attributi, in mancanza di specifica sulla visibilità, si assume visibilità pubblica per le classi.**

Nelle proprietà possono essere contenute informazioni riguardanti le eccezioni, una nota indicante se l'operazione è stratta etc...

**Un'operazione non è un metodo.** Un'operazione di UML è una dichiarazione, con un nome, dei parametri, un tipo di ritorno, un elenco di eccezioni e magari un insieme di vincoli di pre-condizioni e post-condizioni.

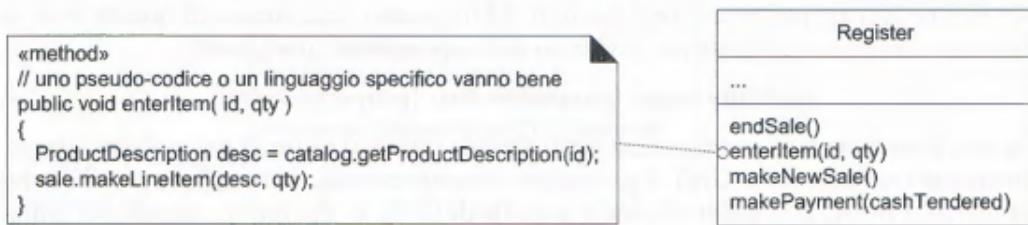
In UML, un **metodo** è l'implementazione di un'operazione; se sono definiti dei vincoli, il metodo deve soddisfarli. Un metodo può essere illustrato in vari modi, tra cui:

- nei diagrammi di interazione, dai dettagli e dalla sequenza dei messaggi
- nei diagrammi delle classi, con un simbolo di nota di UML stereotipato con `#method`

## Capitolo 7. Diagrammi delle Classi

---

Si noti che quando si usa una nota UML per mostrare un metodo, in realtà si stanno mischiando viste statiche e dinamiche in uno stesso diagramma. Il corpo di un metodo (che definisce del comportamento dinamico) aggiunge un elemento dinamico a un diagramma delle classi statico:



Si noti che questo stile va bene per i diagrammi di un libro o di un documento e per l'output generato da uno strumento, ma forse è troppo meticoloso o stilizzato per l'abbozzo o per fornire l'input a uno strumento. Gli strumenti possono fornire una finestra a comparsa per inserire in modo semplice il codice per un metodo.

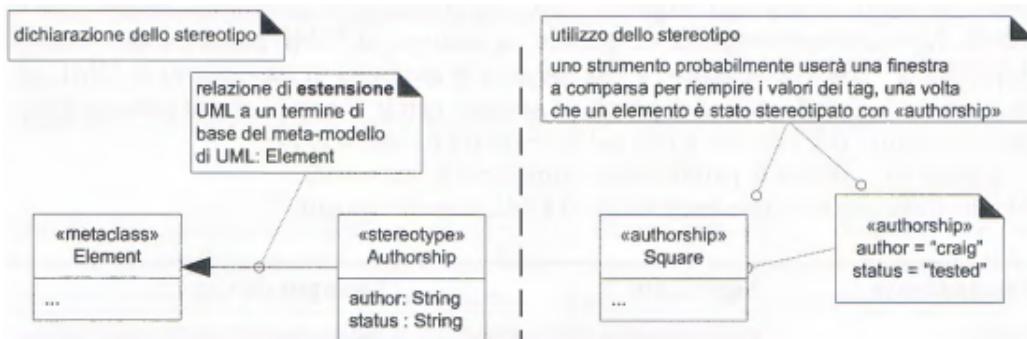
Le operazioni di accesso **get** e **set** sono spesso escluse (o filtrate) dai diagrammi delle classi, a causa dell'elevato rapporto disturbo-valore che generano; per  $n$  attributi, ci possono essere  $2n$  operazioni get e set poco interessanti. La maggior parte degli strumenti per UML consente di filtrare la loro visualizzazione, ed è particolarmente comune ignorarle quando si esegue l'abbozzo alla parete.

Una **parola chiave** in UML è un decoratore testuale per classificare un elemento di un modello. Per esempio, la parola chiave `interface` indica che il rettangolo per un classificatore è un'interfaccia anziché una classe regolare. In generale, quando un elemento di UML può avere una "stringa di proprietà", come per esempio le operazioni e le estremità di associazioni di UML, alcuni dei termini della stringa di proprietà saranno parole chiave (e alcuni possono essere termini definiti dall'utente), scritti nel formato tra parentesi graffe anziché tra le "":

Parola chiave	Significato	Esempio di uso
<code>«actor»</code>	il classificatore è un attore	nei diagrammi delle classi, sopra al nome di un classificatore
<code>«interface»</code>	il classificatore è un'interfaccia	nei diagrammi delle classi, sopra al nome di un classificatore

Parola chiave	Significato	Esempio di uso
{abstract}	l'elemento è astratto; non può essere istanziato	nei diagrammi delle classi, dopo il nome di un classificatore o il nome di un'operazione
{ordered}	un insieme di oggetti ha un ordine predefinito	nei diagrammi delle classi, a un'estremità di associazione

Come le parole chiave, gli **stereotipi** sono mostrati con i simboli delle virgolette basse, come «authorship». Tuttavia **non** sono parole chiave, infatti uno stereotipo rappresenta un raffinamento di un concetto di modellazione esistente, ed è definito all'interno di un profilo UML; informalmente, un profilo è una collezione di stereotipi, tag e vincoli correlati per specializzare l'uso di UML in un dominio o una piattaforma specifica, per esempio un profilo UML per la gestione del progetto o per la modellazione dei dati. Gli stereotipi sono un *meccanismo di estensione* di UML, il quale ne predefinisce alcuni come «destroy»:



In UML, una proprietà è un "valore con un nome che denota una caratteristica di un elemento; una proprietà ha un impatto semantico". Alcune proprietà sono predefinite in UML, come la visibilità (una proprietà di un'operazione). Altre proprietà possono essere definite dall'utente. Le proprietà degli elementi possono essere presentate in diversi modi, e un approccio testuale è quello di utilizzare il formato delle stringhe di proprietà di UML, nel formato {name1 = value1, name2 = value2}; per esempio {abstract, visibility = public}. Alcune proprietà sono mostrate senza valore, come {abstract}; solitamente ciò implica che si tratta di una proprietà booleana, un'abbreviazione di {abstract = true}. Si noti che {abstract} è un esempio sia di un vincolo che di una stringa di proprietà.

Vediamo ora la definizione di **generalizzazione** in UML: *una relazione tassonomica tra un classificatore più generale e un classificatore più specifico. Ogni istanza del classificatore più specifico è anche un'istanza indiretta del classificatore più generale. Pertanto il classificatore più specifico possiede indirettamente le caratteristiche del classificatore più generale.*

**Non sempre la generalizzazione equivale all'ereditarietà dei linguaggi di programmazione orientati a oggetti.**

Le **classi astratte**, oltre che mediante il tag {abstract}, possono essere indicate anche mediante la scrittura del nome della classe in corsivo.

Le **classi finali** e le **operazioni finali** non possono essere ridefinite nelle sottoclassi, sono mostrate con il tag {leaf}.

Analizziamo ora le **dipendenze**. Le linee di dipendenza possono essere usate in qualsiasi diagramma, ma sono particolarmente comuni nei diagrammi delle classi e dei package. UML comprende una **relazione di dipendenza** generale che indica che un elemento *cliente* (di qualsiasi tipo, comprese le classi, i package, i casi d'uso e così via) è a conoscenza di un altro elemento *fornitore* e che un cambiamento nel fornitore potrebbe influire sul cliente.

Una dipendenza è illustrata con una linea tratteggiata, con una freccia dal cliente verso il fornitore.

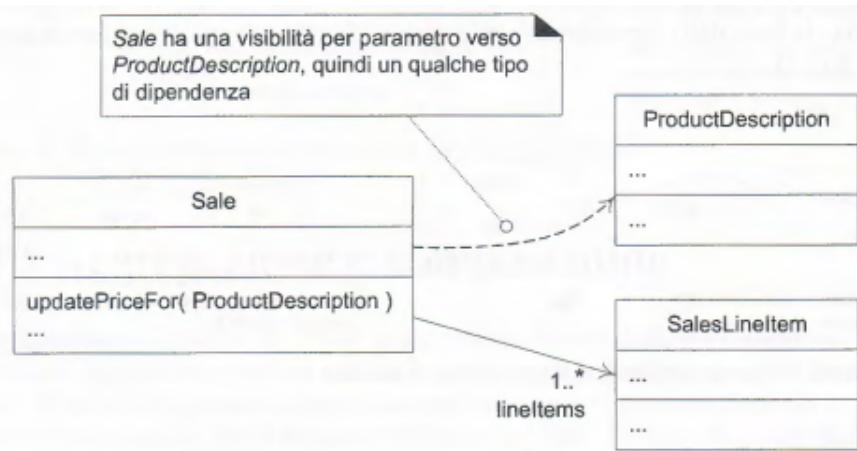
Esistono molte tipologie di dipendenze; ecco alcune tipologie comuni in termini di oggetti e di diagrammi delle classi:

- avere un attributo del tipo del fornitore
- inviare un messaggio ad un fornitore (e la visibilità verso il fornitore sarebbe data da un attributo, variabile parametro, una variabile locale, una variabile globale, o una visibilità di classe, con una chiamata di metodo o di classe)
- ricevere un parametro del tipo del fornitore
- il fornitore è una superclasse o un'interfaccia implementata

Tutti questi elementi potrebbero essere mostrati con una linea di dipendenza in UML, ma per alcune di queste tipologie ci sono già delle linee speciali che suggeriscono la dipendenza. Per esempio, c'è una linea speciale di UML per indicare la superclasse, una che mostra l'implementazione di un'interfaccia e una per gli attributi (la linea "attributo come associazione"). Vediamo un esempio:

## Capitolo 7. Diagrammi delle Classi

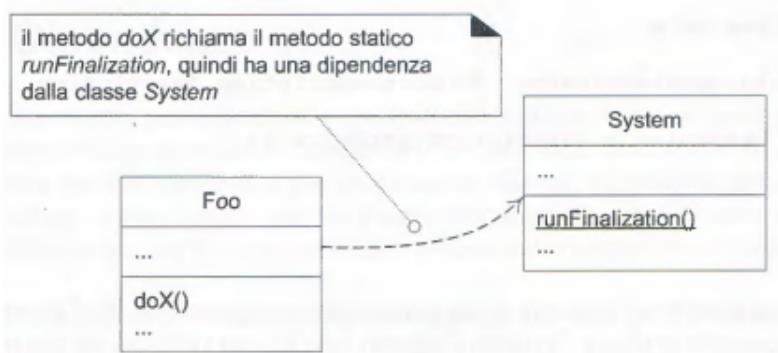
---



che descrive:

```
public class Sale {
    public void updatePriceFor( ProductDescription description ) {
        Money basePrice = description.getPrice();
        ...
    }
    ...
}
```

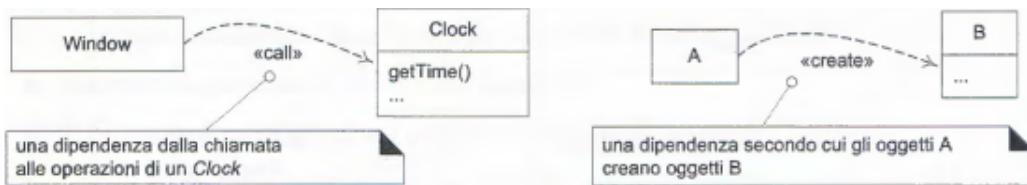
oppure si ha:



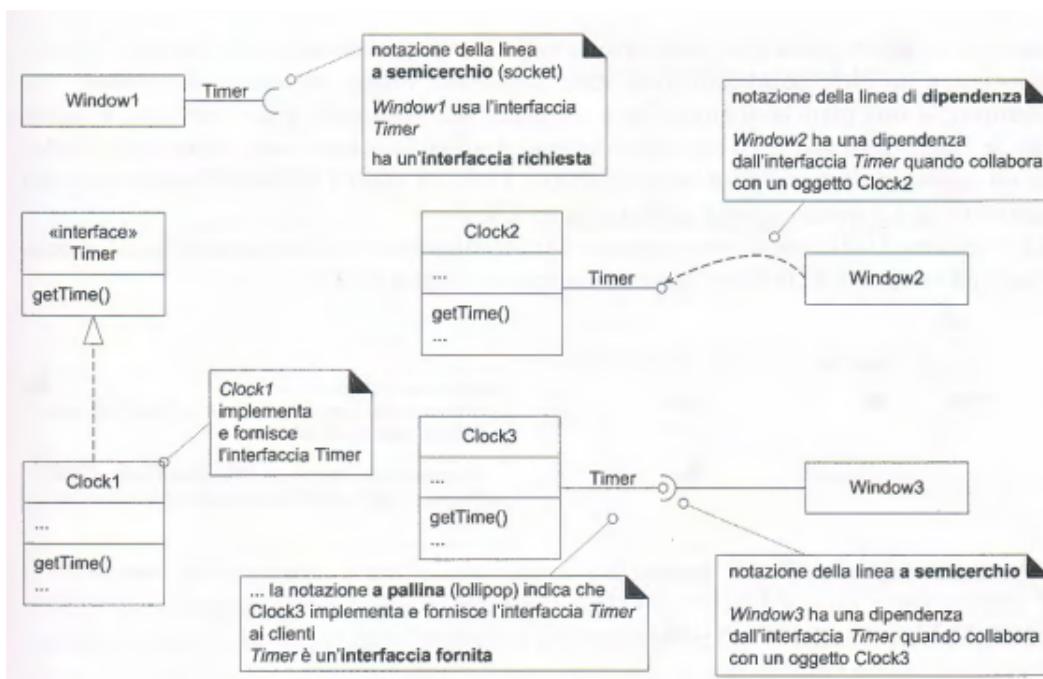
per:

```
public class Foo {
    public void doX() {
        System.runFinalization();
        ...
    }
    ...
}
```

Una linea di dipendenza può essere etichettata:



UML fornisce diversi modi per mostrare l'implementazione di una **interfaccia**, il fornire un'interfaccia ai clienti e la dipendenza da una interfaccia (una interfaccia richiesta). In UML, l'implementazione di un'interfaccia viene chiamata formalmente realizzazione di interfaccia. In UML2 è stata introdotta la **notazione a semicerchio, a socket**:



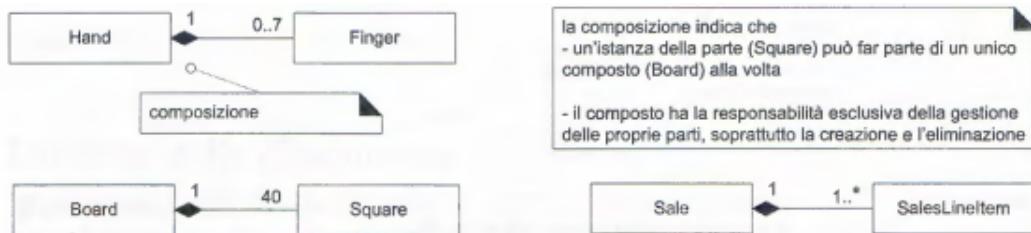
L'**aggregazione** è associazione che suggerisce in modo approssimativo una relazione intero-parte ha la solita semantica ma è definita in UML (anche se gli stessi creatori di UML ne sconsigliano l'uso).

La **composizione**, detta anche *aggregazione composta*, è un tipo forte di aggregazione intero-parte che è utile da mostrare in alcuni modelli. Una composizione comporta che:

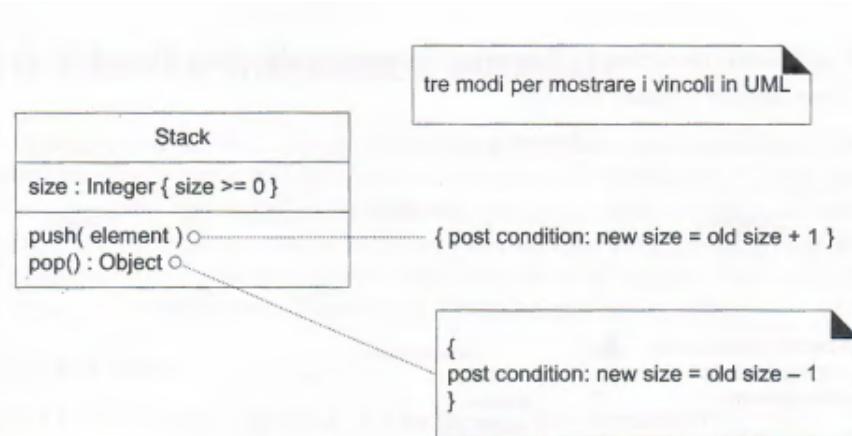
- un'istanza della parte appartiene a una sola istanza del composto alla volta

- la parte deve sempre appartenere a un composto
- il composto è responsabile della creazione e della cancellazione delle sue parti e se il composto viene distrutto, le rispettive parti devono essere distrutte, oppure associate a un altro composto

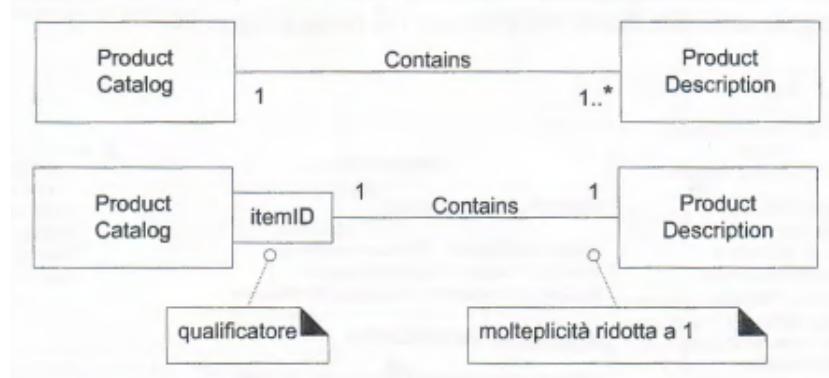
La notazione UML per la composizione è un rombo pieno su una linea di associazione, posto all'estremità della linea vicina al composto:



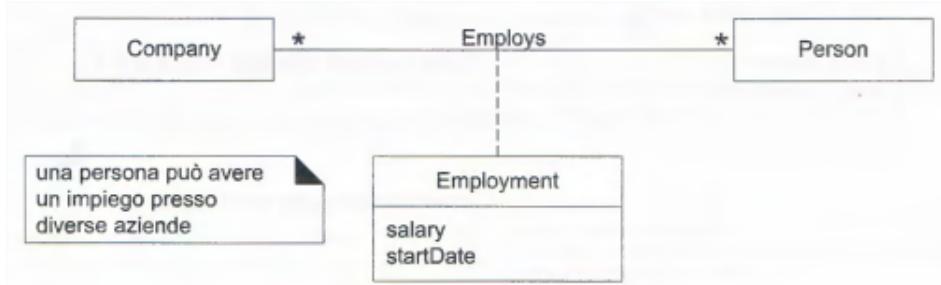
I vincoli possono essere usati nella maggior parte dei diagrammi UML, ma sono particolarmente comuni nei diagrammi delle classi. Un vincolo di UML è una restrizione o una condizione su un elemento UML; viene visualizzato come testo tra parentesi graffe, per esempio: { size >= 0 }:



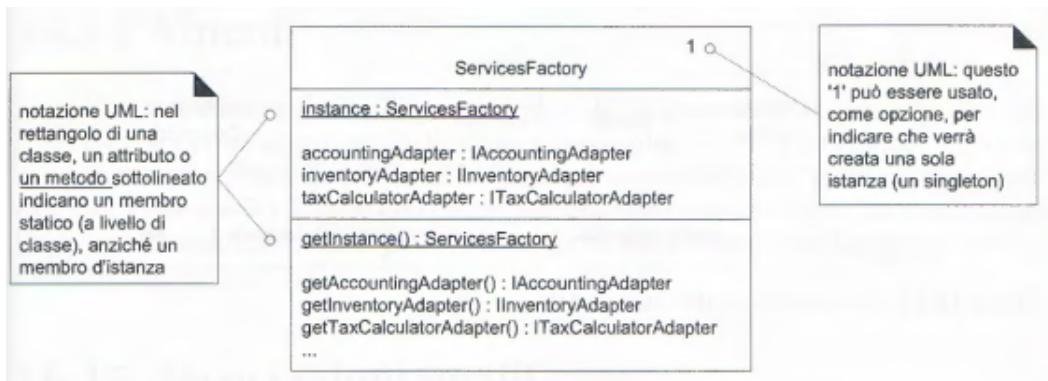
Un'**associazione qualificata** ha un qualificatore usato per selezionare un oggetto (o più oggetti) da un insieme più ampio di oggetti correlati, sulla base della chiave del qualificatore. Informalmente, da un punto di vista software, suggerisce di effettuare una ricerca tra gli elementi tramite una chiave, come gli oggetti in una *HashMap*. La qualifica riduce la molteplicità all'estremità di destinazione dell'associazione, solitamente da molti a uno, poiché solitamente implica la selezione di una sola istanza da un insieme più ampio, come si vede nei due esempi:



Una **classe di associazione** consente di trattare un'associazione come una classe a sé, e di modellarla con attributi, operazioni e altre caratteristiche. In UML, ciò viene illustrato da una linea tratteggiata che collega l'associazione con la classe di associazione:



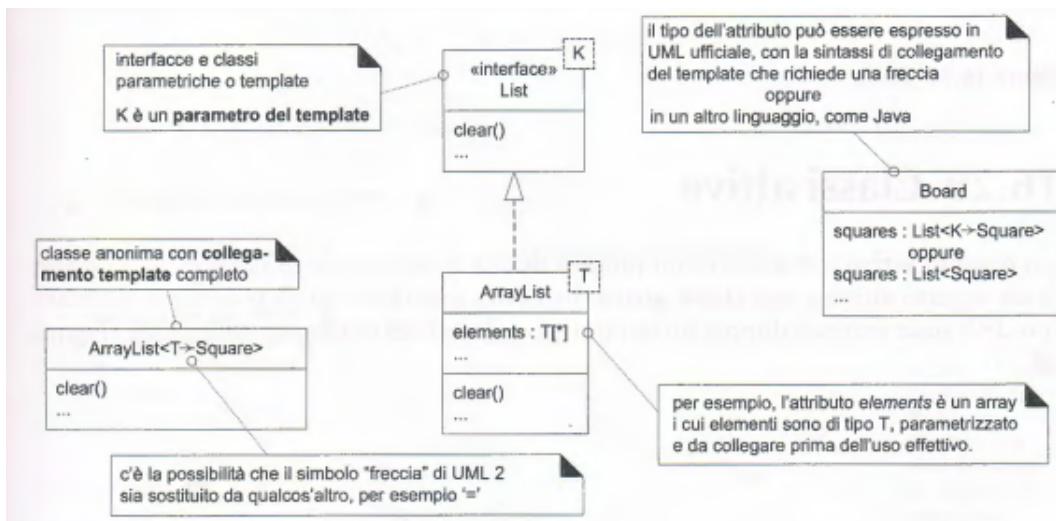
Nel mondo dei design pattern 00, ce n'è uno che è particolarmente diffuso, chiamato il pattern **Singleton**. Un'implicazione di questo pattern è che esiste una sola istanza di una classe, mai due. In altre parole si tratta di un'istanza "singleton". In un diagramma UML, una simile classe può essere contrassegnata da '1' nell'angolo superiore destro nella sezione del nome:



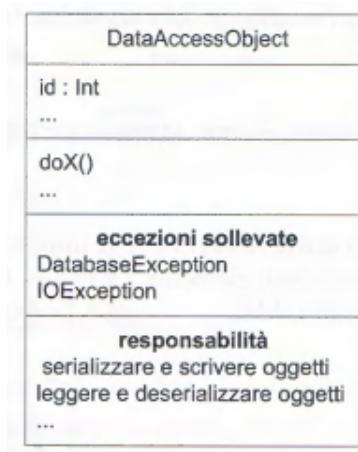
## Capitolo 7. Diagrammi delle Classi

---

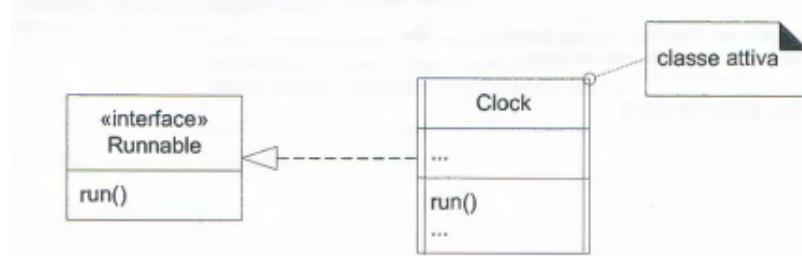
Molti linguaggi (C, C++, ... ) supportano tipi a template, noti anche come **template**, tipi parametrizzati e generici. Questi sono usati comunemente come tipo per gli elementi delle classi collezione, come gli elementi di liste e mappe. Nell'esempio si ha che *List* e *ArrayList* sono parametrizzate con il tipo *Square*, definito dall'utente:



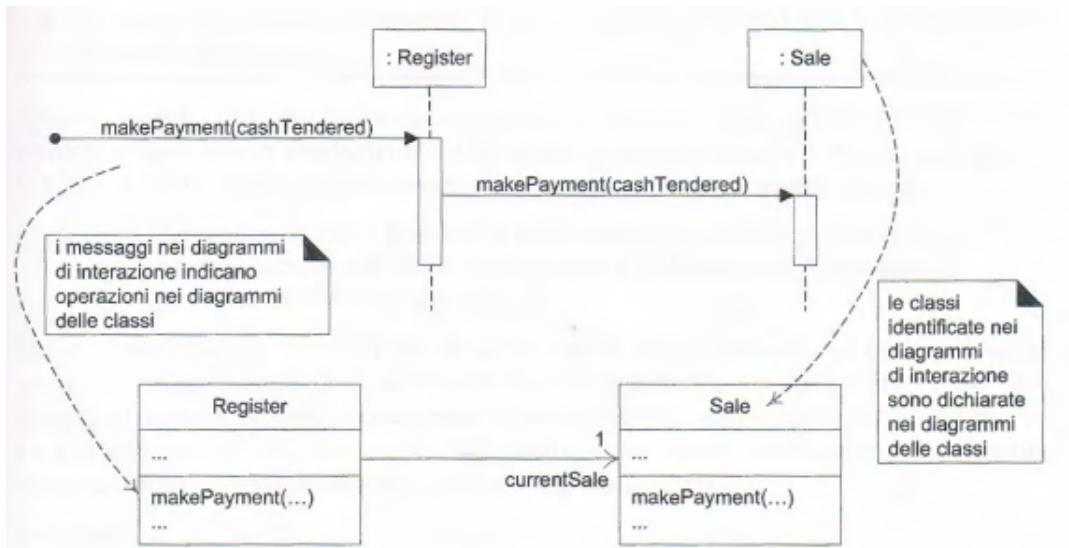
Oltre alle normali sezioni predefinite di una classe (nome, attributi e operazioni), è possibile aggiungere al rettangolo di una classe delle ulteriori sezioni definite dall'utente:



Un **oggetto attivo** è eseguito in un proprio thread di esecuzione e lo controlla. La classe di un oggetto attivo è una classe attiva. In UML, una **classe attiva** può essere mostrata con delle linee verticali doppie sui lati sinistro e destro del rettangolo della class:



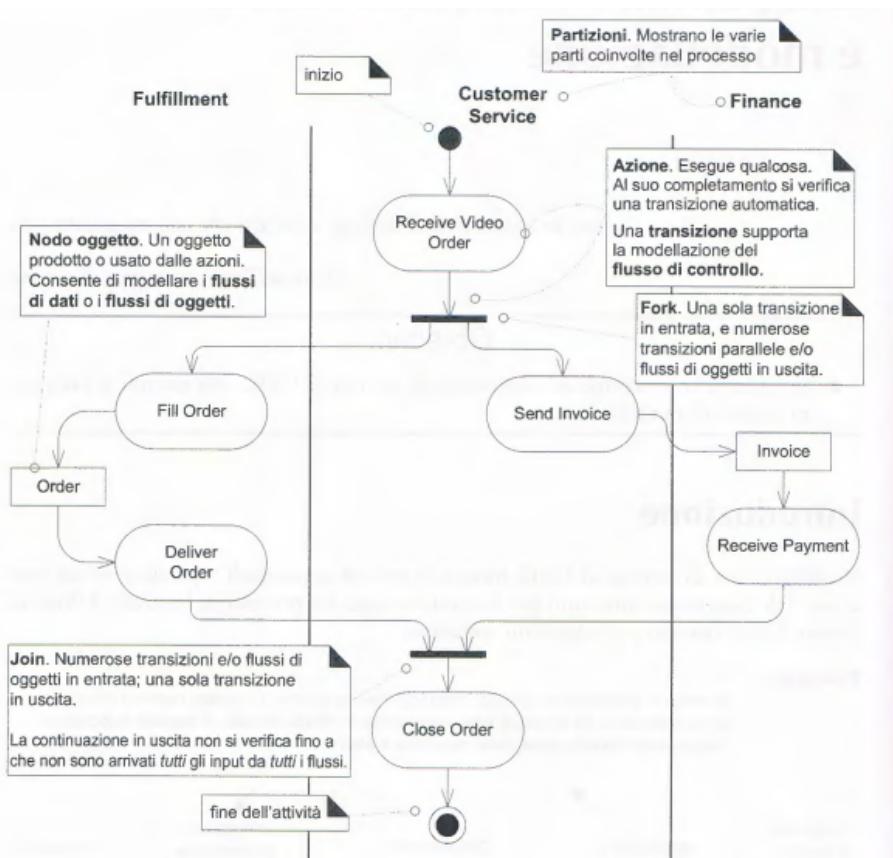
È possibile generare le definizioni dei diagrammi delle classi dai diagrammi di interazione. Ciò suggerisce un ordinamento lineare secondo cui disegnare i diagrammi di interazione prima dei diagrammi delle classi, anche se abbiamo visto che spesso l'elaborazione si ha in parallelo:



# Capitolo 8

## Diagrammi di Attività in UML

Un **diagramma di attività** di UML mostra le attività sequenziali e parallele in un processo. Tali diagrammi sono utili per la modellazione dei processi di business, i flussi di lavoro, i flussi dei dati e gli algoritmi complessi. Vediamo un'immagine con la notazione base, dove si mostrano i concetti base di **azione**, **partizione**, **fork**, **join** e **nodo oggetto**:



Si ha che:

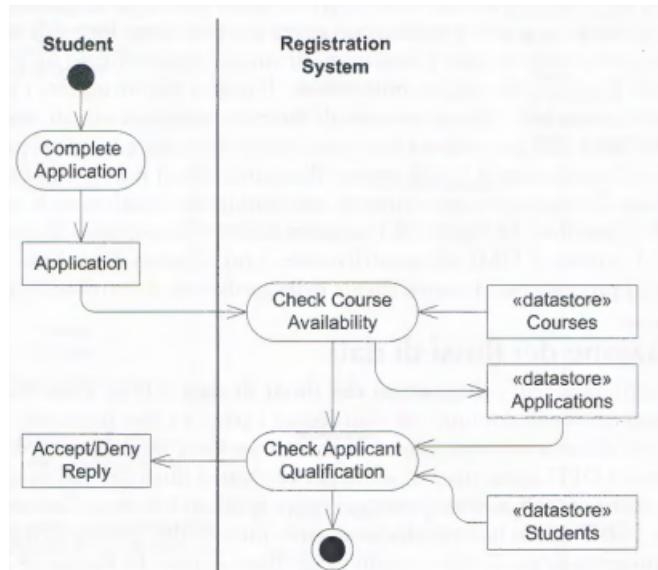
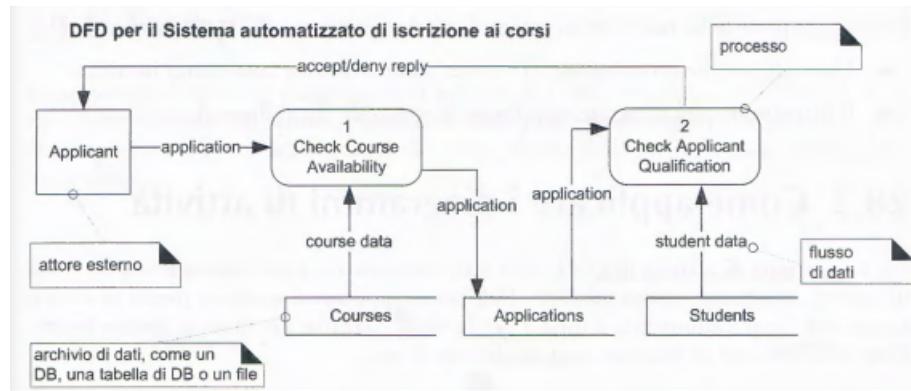
- una volta terminata un'azione, si verifica una transizione automatica in uscita
- il diagramma può mostrare sia il flusso di controllo che il flusso dei dati

*Un diagramma di attività di UML offre una notazione ricca per mostrare una sequenza di attività, comprese attività parallele. Può essere applicato a qualsiasi punto di vista o scopo, ma l'applicazione più diffusa è per la visualizzazione dei flussi di lavoro (workflow) e dei processi di business, nonché dei casi d'uso.*

Vediamo i due casi:

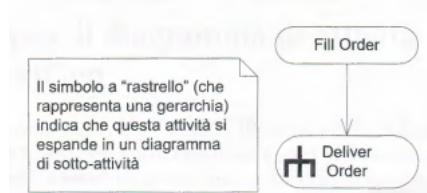
1. **modellazione dei processi di business:** si ottengono diagrammi come quelli nell'immagine di presentazione del capitolo
2. **modellazione dei flussi di dati:** i diagrammi dei flussi di dati **DFD**, *Data-Flow Diagram*, sono divenuti un modo comune per visualizzare i passi e i dati principali coinvolti nei processi di un sistema software. Non è la stessa cosa della modellazione dei processi di business; anzi, i DFD erano usati di solito per mostrare i flussi dei dati in un sistema informatico , anche se teoricamente potevano essere applicati alla modellazione dei processi di business. I DFD erano utili per documentare i flussi di dati principali o per esaminare un nuovo progetto di alto livello in termini dei flussi di dati. Nella notazione *Gane-Sarson* si hanno i passi numerati secondo l'ordine. Le informazioni modellate in un DFD sono utili, sia per la documentazione sia per la scoperta, ma UML non prevede la notazione dei DFD. I diagrammi delle attività di UML possono soddisfare gli stessi obiettivi. *Si noti che oltre ai nodi oggetto, utili per mostrare i flussi di dati , si possono usare anche i nodi datastore UML.*

Vediamo il confronto tra i diagrammi di flusso dei dati di Gane-Sarson, prima, e la loro riproduzione coi diagrammi di attività, poi:

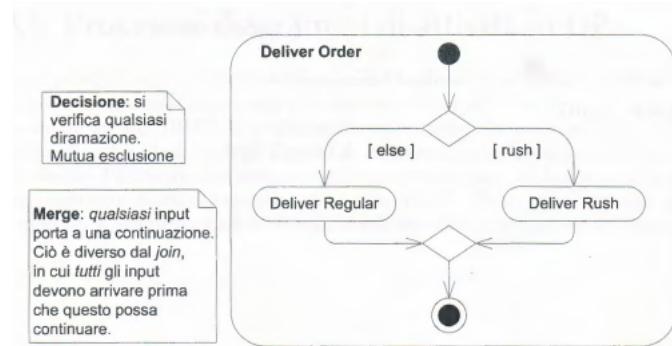


Gli algoritmi paralleli nei problemi di programmazione concorrente coinvolgono più partizioni, oltre alle operazioni **fork** e **join**. Lo spazio fisico complesso è suddiviso in grossi blocchi, e vengono eseguiti molti thread (o processi) paralleli, uno per ciascun sottoblocco. In questi casi possono essere utilizzate le partizioni dei diagrammi di attività di UML per rappresentare i diversi thread o processi del sistema operativo. I nodi oggetto possono essere utilizzati per modellare gli oggetti e i dati condivisi. E naturalmente è possibile utilizzare il fork per modellare la creazione e l'esecuzione parallela di thread o di processi multipli, uno per ciascuna partizione.

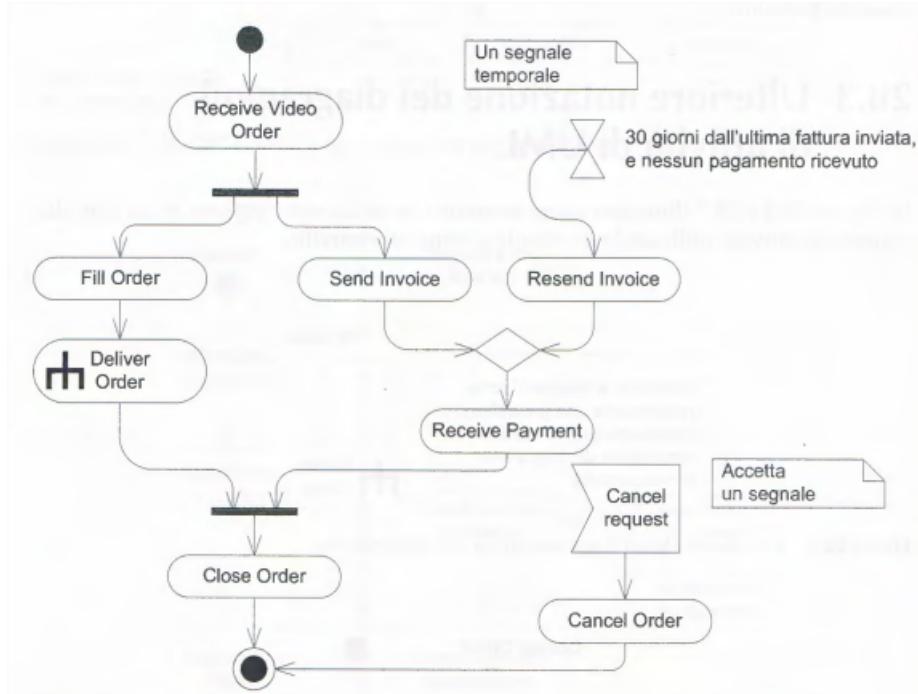
Vediamo come mostrare che un'attività è espansa in un altro diagramma di attività, utilizzando il simbolo a forma di rastrello:



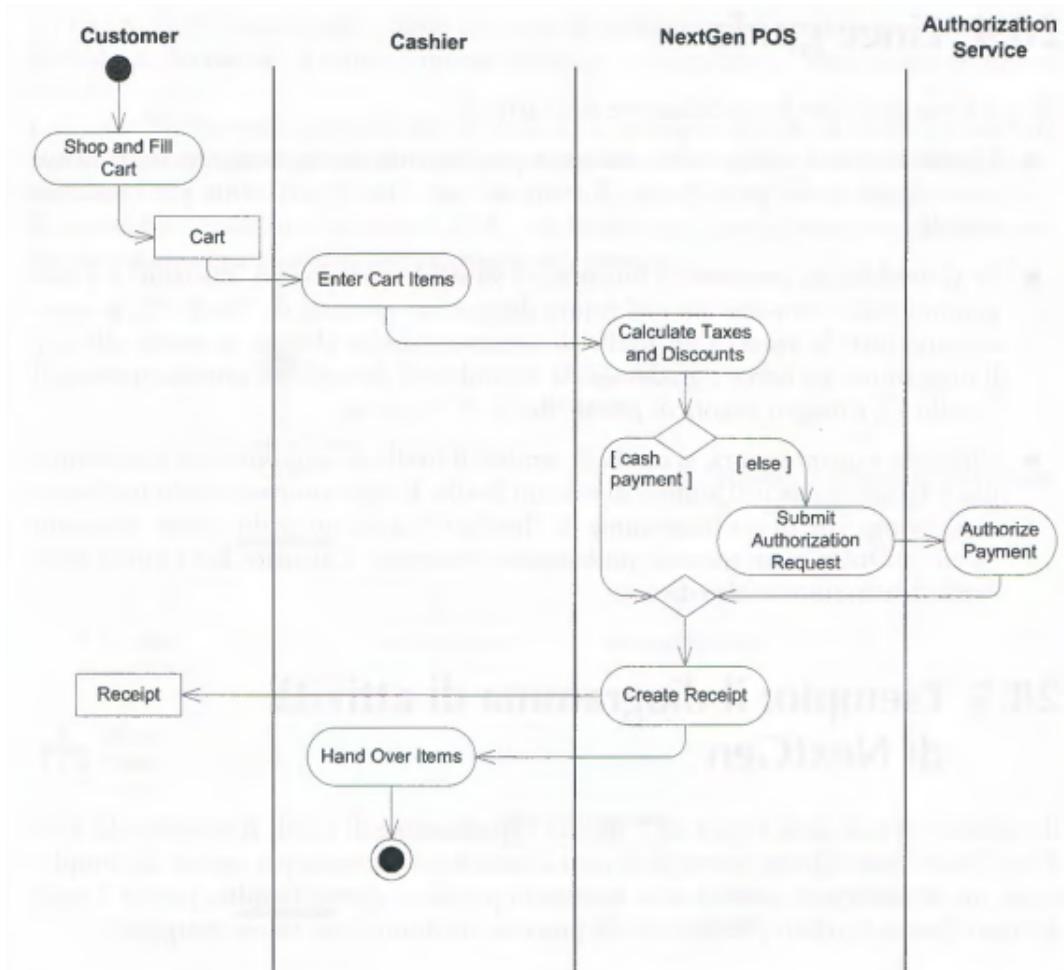
Inoltre si può mostrare le diramazioni condizionali, utilizzando il simbolo di decisione. Il simbolo **merge**, correlato, mostra come i flussi possano riunirsi insieme.



Vediamo anche i **segnali**, che sono utili per esempio quando si devono modellare eventi quali avviare un'azione in un determinato istante di tempo o una richiesta di annullamento:



Vediamo un esempio, invece, completo della rappresentazione di un caso d'uso mediante il diagramma delle attività:



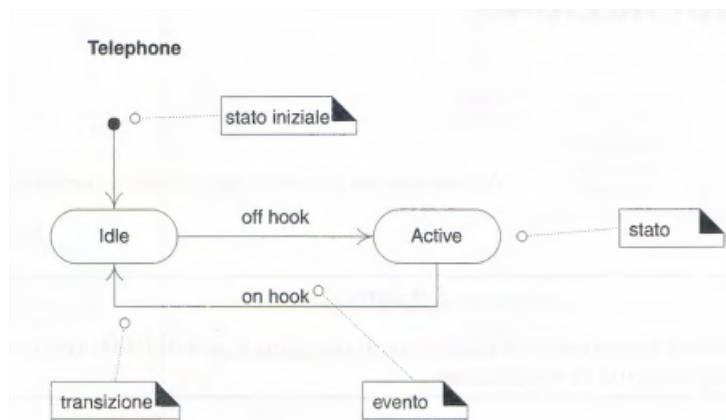
Una delle discipline di UP è la **Modellazione del business**; il suo scopo è quello di *comprendere e comunicare la struttura e la dinamica dell'organizzazione in cui deve essere rilasciato un sistema*. Un elaborato fondamentale della disciplina di Modellazione del business è il **Modello degli Oggetti di Business** (un soprainsieme del Modello di Dominio di UP), che sostanzialmente visualizza come funziona un business, utilizzando i diagrammi delle classi, di sequenza e di attività di UML. Pertanto i diagrammi di attività sono particolarmente applicabili nella disciplina della Modellazione del business di UP.

## Capitolo 9

# Diagrammi di Macchina a Stati in UML

Come i diagrammi di attività, i **diagrammi di macchina a stati** di UML mostrano una vista dinamica. UML include la notazione per illustrare gli eventi e gli stati delle cose: transazioni, casi d'uso, persone e così via.

Un diagramma di macchina a stati di UML, illustra gli eventi e gli stati interessanti per un oggetto, e il comportamento di un oggetto in reazione a un evento. Le transizioni sono mostrate come frecce, etichettate con il rispettivo evento. Gli stati sono mostrati come rettangoli arrotondati. È comune includere uno pseudo-stato iniziale, con una transizione automatica a un altro stato quando viene creata l'istanza:



Un diagramma di macchina a stati mostra il ciclo di vita di un oggetto: quali eventi sperimenta, le sue transizioni e gli stati in cui si trova tra questi eventi. Non è necessario che illustri ogni possibile evento; se si verifica un evento che non è rappresentato nel diagramma, l'evento viene ignorato per quanto riguarda il diagramma di macchina a stati. Pertanto, è possibile creare un

diagramma di macchina a stati che descrive il ciclo di vita di un oggetto a livelli che possono essere semplici o complessi, a seconda delle necessità. Definiamone meglio le parti:

- un **evento** è un evento significativo, degno di nota
- uno **stato** è la condizione di un oggetto in un certo intervallo di tempo, il tempo tra due eventi
- una **transizione** è una relazione tra due stati che indica che quando si verifica un evento, l'oggetto passa dallo stato precedente allo stato successivo

Se un oggetto risponde a un evento sempre nello stesso modo, viene considerato **indipendente dallo stato** (o senza modo) rispetto a quell'evento. Se un oggetto reagisce sempre allo stesso modo per tutti gli eventi di interesse, è un oggetto indipendente dallo stato. Al contrario, gli oggetti **dipendenti dallo stato** reagiscono in modo diverso agli eventi a seconda del loro stato o modo.

**Bisogna considerare le macchine a stati per oggetti dipendenti dallo stato che hanno un comportamento complesso, non per gli oggetti indipendenti dallo stato.**

*In generale, i sistemi informatici aziendali hanno poche classi dipendenti dallo stato complesse. In questi casi, raramente è utile applicare la modellazione delle macchine a stati. Al contrario, il controllo di processo, il controllo dei dispositivi, i gestori di protocollo e i domini delle telecomunicazioni hanno spesso molti oggetti dipendenti dallo stato.*

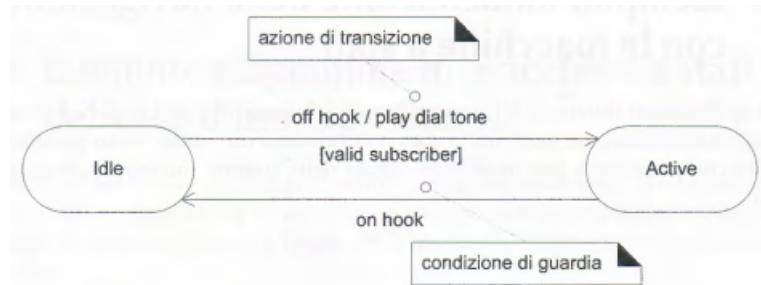
Le macchine a stati, come già sappiamo, sono applicate in due modi:

1. per modellare il comportamento di un oggetto reattivo complesso in risposta agli eventi
2. per modellare le sequenze valide delle operazioni, ovvero specifiche di protocollo o di linguaggio

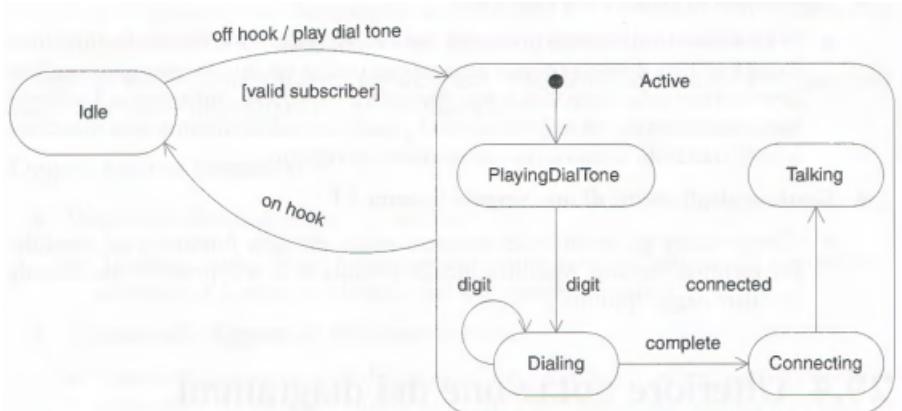
Si hanno diverse situazioni in cui sono comodi:

- per rappresentare oggetti reattivi complessi come i dispositivi fisici controllati da software, come transazioni e oggetti di business correlati o come i mutatori di ruolo, ovvero oggetti che cambiano ruolo
- per rappresentare protocolli e sequenze valide, come protocolli di comunicazione (esempio TCP), come flussi o navigazione di pagine UI

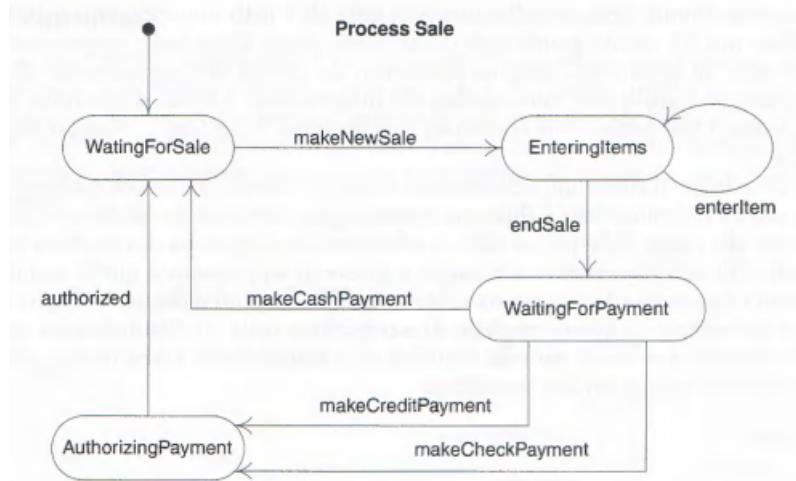
Una transizione può causare l'avvio di un'azione. In un'implementazione software, ciò può rappresentare l'invocazione di un metodo della classe a cui è associato il diagramma di macchina a stati. Una transizione può anche avere una guardia condizionale, ovvero un test booleano. La transizione avviene solo se il test viene superato:



Uno stato consente l'annidamento per contenere dei sottostati; un sottostato eredita le transizioni del proprio superstato (lo stato che lo racchiude). I sottostati possono essere mostrati graficamente annidandoli nel rettangolo arrotondato del superstato:



Nei casi di studio non ci sono oggetti reattivi complessi veramente interessanti, per cui verrà illustrato un diagramma di macchina a stati per mostrare la sequenza valida delle operazioni di caso un d'uso:



In UP non esiste alcun modello di *macchina a stati* anche se può essere implementata in qualsiasi modello UP.

# Capitolo 10

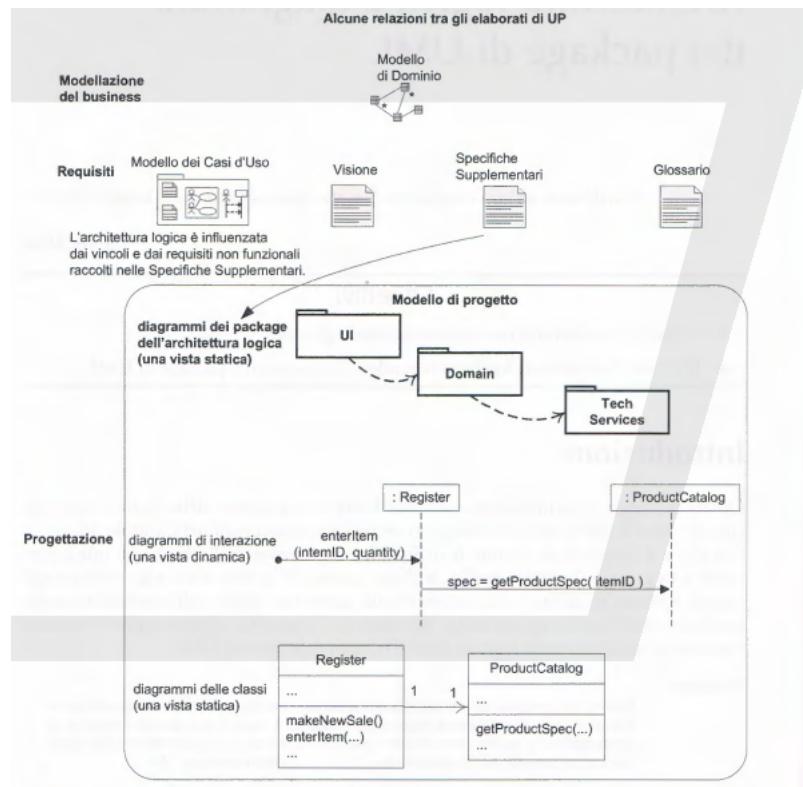
## Architettura Logica

La progettazione di un tipico sistema orientato agli oggetti è basata su diversi strati architettonici, come uno strato dell'interfaccia utente, uno strato della logica applicativa (o "del dominio") e così via.

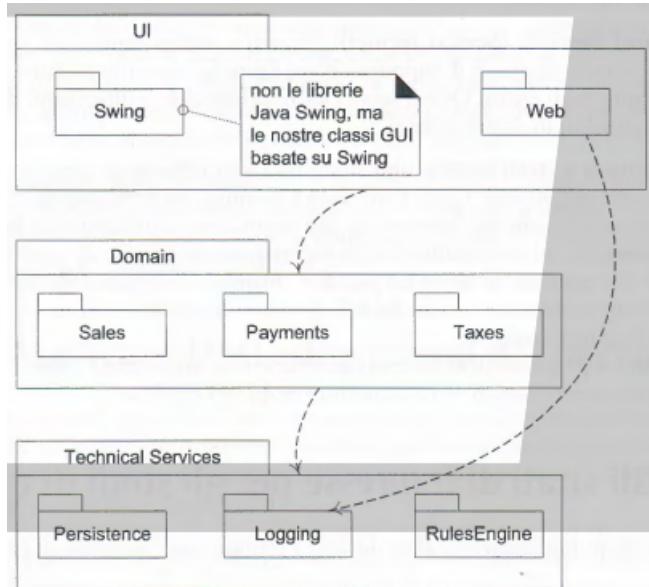
L'architettura logica può essere illustrata sotto forma di diagrammi dei package di UML, come parte del Modello di Progetto, e magari riassunta come vista nel Documento dell'Architettura Software. L'input principale sono le forze architettoniche raccolte nelle Specifiche Supplementari. L'architettura logica definisce i package all'interno dei quali sono definite le classi software.

## Capitolo 10. Architettura Logica

---



e vediamo anche un diagramma dei package:



**L'architettura logica** è l'organizzazione su larga scala delle classi software in package (o namespace), sottosistemi e strati. È chiamata architettura

logica poiché non ci sono decisioni da prendere su come questi elementi siano distribuiti sui processi o sui diversi computer fisici di una rete (queste ultime decisioni fanno parte dell'architettura di deployment). Uno **strato** è un gruppo a grana molto grossa di classi, package o sottosistemi, che ha delle responsabilità coese rispetto a un aspetto importante del sistema. Inoltre gli strati sono organizzati in modo tale che strati "più alti" (come lo strato dell'interfaccia utente) ricorrono ai servizi degli strati "più bassi", mentre normalmente non avviene il contrario.

In un sistema OOP si hanno normalmente i seguenti strati:

- **User Interface**
- **Application Logic e Domain Objects**, ovvero oggetti software che rappresentano concetti del dominio (per esempio, una classe software Sale) che soddisfano i requisiti dell'applicazione
- **Technical Services**, ovvero oggetti e sototosistemi d'uso generale che forniscono servizi tecnici di supporto, come l'interfacciamento con una base di dati o il logging degli errori. Questi servizi sono solitamente indipendenti dall'applicazione e riusabili in diversi sistemi

In un'**architettura a strati stretta**, uno strato può solo richiamare i servizi dello strato immediatamente sottostante. Questa struttura è comune nei protocolli di rete organizzati a pila, ma non nei sistemi informatici, che normalmente utilizzano un'**architettura a strati rilassata**, in cui uno strato più alto può richiamare i servizi di strati più bassi di diversi livelli.

**Un'architettura logica non deve necessariamente essere organizzata a strati, ma molto spesso lo è.**

Partiamo da una vecchia puntualizzazione:

*Sebbene la tecnologia orientata agli oggetti possa essere applicata a tutti i livelli, questa introduzione all'OOND si concentra sullo strato della logica applicativa principale (o strato "del dominio"), con alcune discussioni secondarie sugli altri strati*

Definiamo meglio il concetto di **architettura software**: *un'architettura è l'insieme delle decisioni significative sull'organizzazione di un sistema software, la scelta degli elementi strutturali da cui è composto il sistema e delle relative interfacce, insieme al loro comportamento specificato dalle collaborazioni tra questi elementi, la composizione di questi elementi strutturali e comportamentali in sottosistemi via via più ampi, e lo stile architettonico che guida questa organizzazione (questi elementi e le loro interfacce, le loro collaborazioni e la loro composizione).*

L’architettura software è quella che fare con la larga scala, con le cosiddette Grandi Idee nelle motivazioni, vincoli, organizzazione, pattern, responsabilità e connessioni di un sistema (o un sistema di sistemi).

## 10.1 Diagramma dei Package

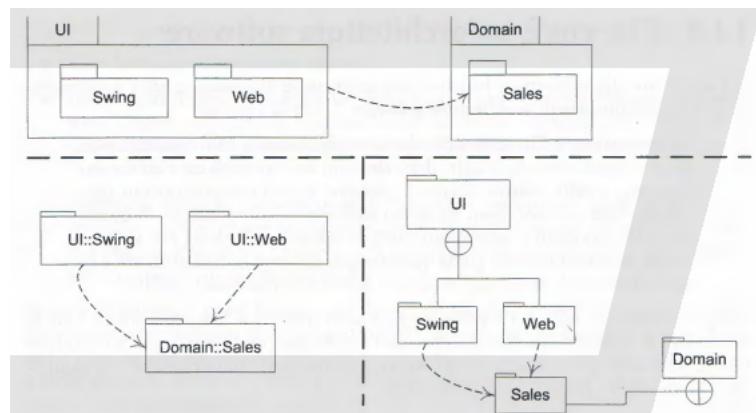
I diagrammi dei package di UML sono spesso utilizzati per illustrare l’architettura logica di un sistema: gli strati, i sottosistemi, i package (nel senso di Java) e così via. Uno strato può essere modellato come un package UML.

Un diagramma dei package di UML fornisce un modo per raggruppare degli elementi. Un package UML può raggruppare qualunque cosa: delle classi, altri package, casi d’uso e così via. È molto comune anche l’annidamento di package. **Il concetto di package in UML è più generale che non quello di un semplice package Java o di un namespace .NET; i package di UML possono rappresentare tutti questi, nonché altre cose.**

Il nome di un package può essere scritto sulla sua linguetta (tab), se all’interno del package sono mostrati dei membri, oppure nella cartella principale, in caso contrario. È comune voler mostrare delle dipendenze (accoppiamenti) tra i package, in modo che gli sviluppatori possano vedere l’accoppiamento su larga scala nel sistema. Per questo scopo viene utilizzata una dipendenza UML, una linea tratteggiata con una freccia che punta verso il package da cui si dipende.

Un package UML rappresenta un **namespace** (“spazio dei nomi”) in modo che due classi abbiano lo stesso nome se vivono in due package diversi.

Talvolta non è opportuno disegnare il rettangolo di un package esterno attorno ai package interni



Vediamo le idee essenziali per la progettazione a strati:

- organizzare la struttura logica su larga scala di un sistema in strati separati con responsabilità distinte e correlate, con una separazione netta e coesa degli interessi, come per esempio il fatto che gli strati "inferiori" sono servizi generali e di basso livello, mentre gli strati superiori sono più specifici per l'applicazione
- collaborazioni e accoppiamenti vanno dagli strati più alti a quelli più bassi, l'verso viene evitato

Vengono inoltre affrontati diversi problemi con l'uso degli strati:

- le modifiche al codice sorgente si propagano in tutto il sistema; molte parti del sistema sono altamente accoppiate
- la logica applicativa è intrecciata con l'interfaccia utente, per cui non può essere riusata con un'interfaccia diversa oppure distribuita su un altro nodo di elaborazione
- i servizi tecnici o la logica di business potenzialmente generali sono intrecciati con una logica più specifica per l'applicazione, per cui non può essere riusata, distribuita su un altro nodo o sostituita in modo semplice con un'implementazione differente
- c'è un accoppiamento alto tra diverse aree di interesse. Pertanto è difficile suddividere il lavoro, con dei confini precisi, tra i vari sviluppatori

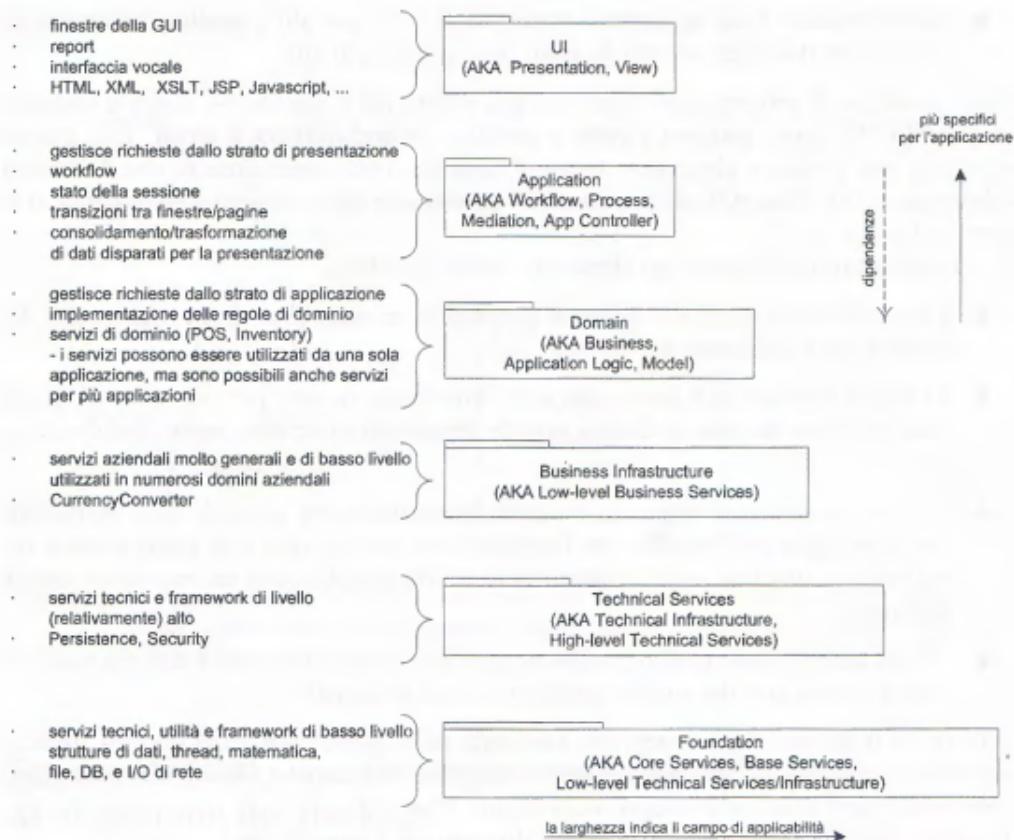
*Lo scopo e il numero degli strati varia a seconda delle applicazioni e dei domini applicativi (sistemi informatici, sistemi operativi e così via).*

Vediamo anche i vantaggi:

- in generale, c'è una separazione degli interessi, una separazione tra servizi di alto e di basso livello e tra servizi generali e quelli più specifici per l'applicazione. Questo riduce l'accoppiamento e le dipendenze, migliora la coesione, aumenta la possibilità di riuso e aumenta la chiarezza
- la complessità relativa a questi aspetti è incapsulata e può essere decomposta
- alcuni strati possono essere sostituiti da nuove implementazioni. Generalmente ciò non è possibile per gli strati di basso livello dei Technical Services (Servizi Tecnici) o Foundation), ma è possibile per gli strati UI, Application (Applicazione) e Domain (Dominio)
- gli strati più bassi contengono funzioni riusabili

- alcuni strati (in primo luogo Domain e Technical Services) possono essere distribuiti
- lo sviluppo da parte dei team è favorito dalla segmentazione logica

Vediamo quindi un esempio:



In uno strato, le responsabilità degli oggetti devono essere fortemente correlate l'uno all'altro, e non devono essere mischiare con le responsabilità degli altri strati. Per esempio, gli oggetti dello strato UI devono essere incentrati sulle attività dell'interfaccia utente, come creare finestre e widget, catturare gli eventi del mouse e della tastiera e così via. Gli oggetti dello strato della logica applicativa o del "dominio" devono essere incentrati sulla logica applicativa, come calcolare il totale o le imposte per una vendita, oppure spostare un segnalino su una plancia di gioco. Gli oggetti dell'interfaccia utente non devono implementare logica applicativa.

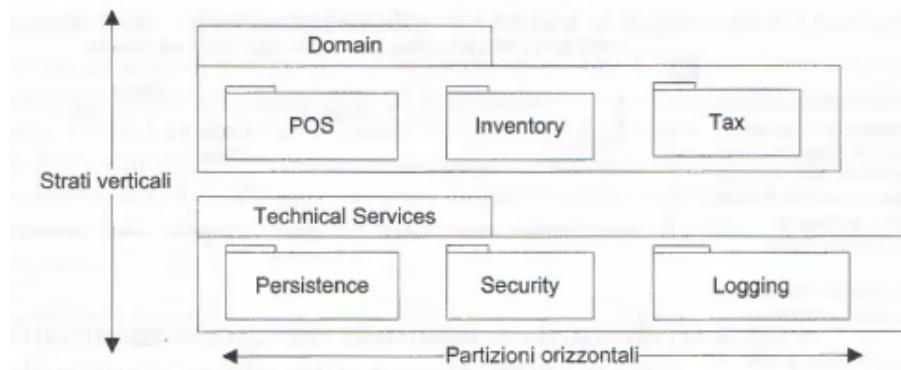
Ciò violerebbe i principi di una chiara **separazione degli interessi e del**

**mantenimento di una coesione alta**, due principi architetturali fondamentali.

Nella programmazione ad oggetti l'approccio consigliato consiste nel creare degli oggetti software con nomi e informazioni simili al dominio del mondo reale, e assegnare a essi responsabilità della logica applicativa. Un oggetto software di questo tipo è chiamato un **oggetto di dominio**. Esso rappresenta una cosa nello spazio del dominio del problema, e ha una logica applicativa o di business correlata. Progettando gli oggetti in questo modo si arriva a uno strato della logica applicativa che può essere chiamato, in modo più preciso, **strato del dominio dell'architettura**, ovvero lo strato che contiene oggetti di dominio per gestire il lavoro della logica applicativa.

Lo strato del dominio è parte del software, mentre il modello di dominio è parte dell'analisi da un punto di vista concettuale; dunque non si tratta della stessa cosa. Tuttavia, creando uno strato del dominio ispirandosi al modello di dominio, si ottiene un salto rappresentazionale basso tra il dominio del mondo reale e il progetto software.

La nozione originaria di **livello (tier)** nell'architettura era uno strato logico, non un nodo fisico; tuttavia questo termine è stato ampiamente utilizzato per indicare un nodo fisico di elaborazione (o un cluster di nodi), come il "livello client" (il computer client). Si dice che gli strati di un'architettura rappresentano le **sezioni verticali**, mentre le partizioni rappresentano una **divisione orizzontale** di sottosistemi relativamente paralleli di uno strato.



### 10.1.1 Principio di separazione Modello-Vista

Si hanno due parti:

1. gli oggetti non UI non devono essere connessi o accoppiati direttamente agli oggetti UI. Infatti le finestre sono relative ad una applicazione specifica, mentre (idealmente) gli oggetti non appartenenti all'interfaccia

grafica possono essere riusati in nuove applicazioni o vi si può accedere mediante una nuova interfaccia.

2. non mettere logica applicativa nei metodi di un oggetto dell'interfaccia utente. Gli oggetti UI devono solo inizializzare gli elementi dell'interfaccia utente, ricevere eventi UI e delegare le richieste di logica agli oggetti non UI

In questo contesto, modello è un sinonimo per lo strato degli oggetti del dominio. Vista è un sinonimo per gli oggetti dell'interfaccia utente, come finestre, pagine web, applet e report.

**Il principio di Separazione Modello-Vista afferma che gli oggetti del modello (dominio) non devono avere una conoscenza diretta degli oggetti della vista (UI), almeno in quanto oggetti della vista.** Un'ulteriore parte di questo principio è che le classi di dominio incapsulano le informazioni e il comportamento relativi alla logica applicativa. Le classi per le finestre sono relativamente leggere; esse sono responsabili dell'input e dell'output, e di catturare gli eventi della GUI, ma non mantengono i dati dell'applicazione né forniscono direttamente della logica applicativa.

Tutto ciò si basa sulle seguenti motivazioni:

- favorire la definizione coesa dei modelli, incentrati sui processi di dominio anziché sulle interfacce utente
- consentire lo sviluppo separato degli strati del modello e dell'interfaccia utente
- consentire di connettere facilmente nuove viste a uno strato del dominio esistente, senza ripercussioni sullo strato del dominio
- minimizzare l'impatto sullo strato del dominio dei cambiamenti dei requisiti relativi all'interfaccia
- consentire viste multiple simultanee sugli stessi oggetti modello, come una vista delle informazioni sulle vendite sia tramite tabella sia tramite rappresentazione grafica
- consentire l'esecuzione dello strato del modello indipendente da quella dello strato dell'interfaccia utente, come in sistema batch o basato sull'elaborazione di messaggi
- consentire un porting facile dello strato del modello a un altro framework per l'interfaccia utente

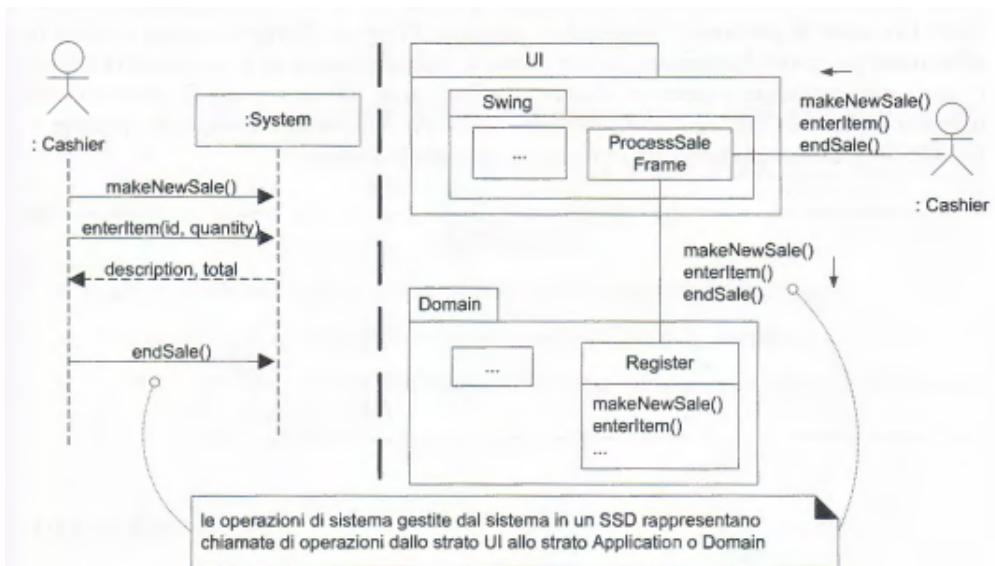
### 10.1.2 SSD e Operazioni di Sistema

Durante il lavoro di analisi, sono stati abbozzati alcuni SSD per gli scenari dei casi d'uso.

Gli SSD mostrano queste operazioni di sistema, ma nascondono gli oggetti specifici della UI. Ciononostante, normalmente saranno gli oggetti dello strato UI del sistema a catturare queste richieste di operazioni di sistema, solitamente con la GUI di un rich client o una pagina web. In un'architettura a strati ben progettata, che sostiene una coesione alta e una separazione degli interessi, gli oggetti dello strato UI inoltreranno (o delegheranno) le richieste da parte dello strato UI allo strato del dominio, affinché vengano gestite.

**I messaggi inviati dallo strato UI allo strato del dominio saranno i messaggi mostrati negli SSD, come enterItem.**

Per esempio:



# Capitolo 11

## GRASP

Viene studiata la progettazione principale orientata agli oggetti, la *OOD*: *dopo aver identificato i requisiti e aver creato un modello di dominio, si aggiungano i metodi alle classi appropriate, e si definiscano i messaggi tra gli oggetti per soddisfare i requisiti*

**Lo strumento critico della progettazione per lo sviluppo software è una mente ben istruita sui principi di progettazione. Non è né UML né qualsiasi altra tecnologia.**

Si parte dagli **input di processo**. Non tutti questi elaborati sono necessari. Si tenga presente che in UP tutti gli elementi sono opzionali, e vengono creati possibilmente per ridurre dei rischi.

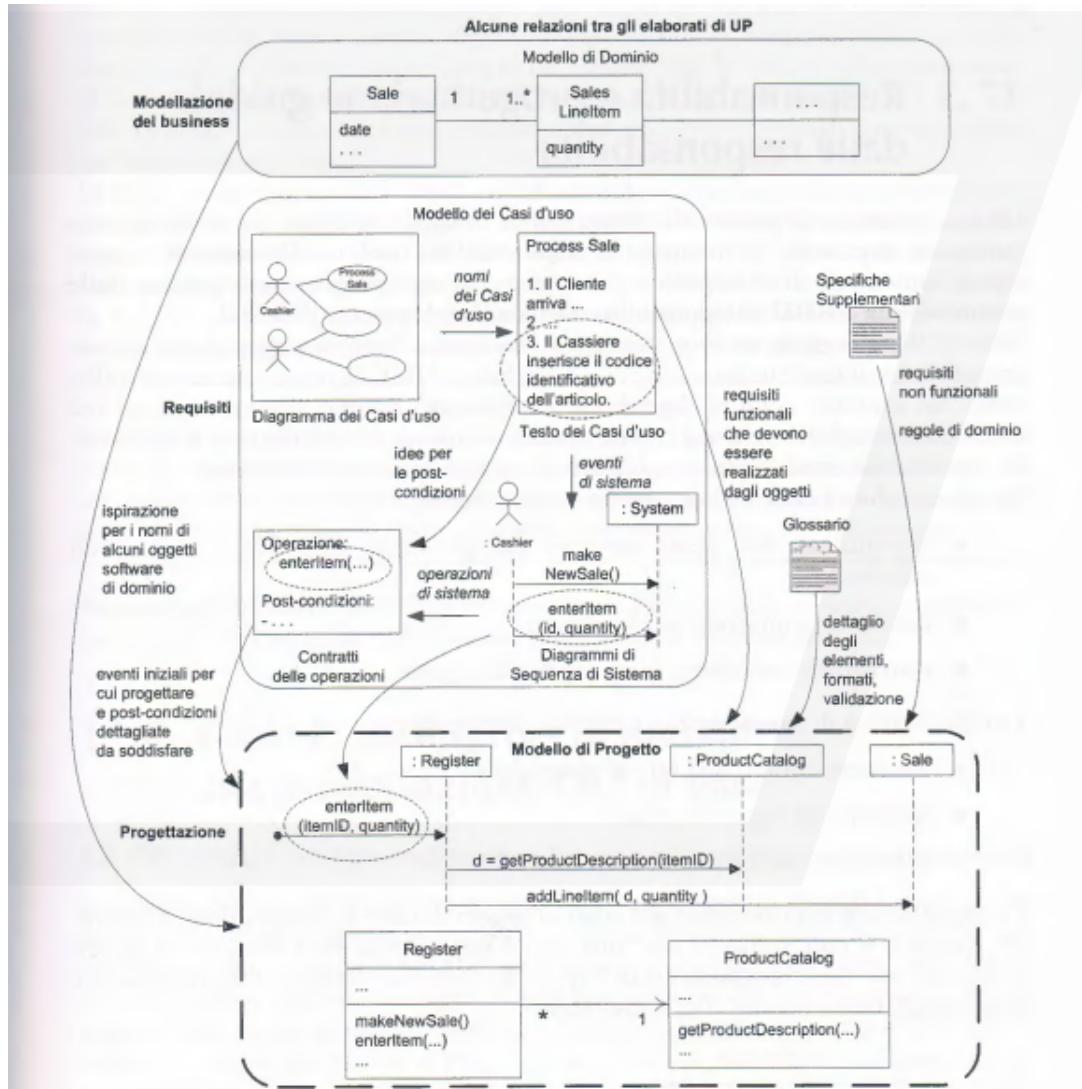
Dato uno o più di questi input, gli sviluppatori possono:

1. iniziare immediatamente a codificare (idealmente con lo sviluppo preceduto dai test)
2. iniziare con un po' di modellazione UML per la progettazione a oggetti
3. iniziare con un'altra tecnica di modellazione

Nel caso di UML, il punto cruciale non è UML, ma la modellazione visual<sup><':</sup>, ovvero usare un linguaggio che consente di esaminare le scelte di progetto in modo più visuale di quanto non si possa fare semplicemente con il testo grezzo.

La cosa più importante è che, durante le attività di disegno (e codifica), vengano applicati vari principi di progettazione OO, come i **pattern GRASP** e i design pattern Gang-of-Four (GoF). L'approccio complessivo al fare la modellazione per la progettazione OO si baserà sulla metafora della progettazione guidata dalle responsabilità (RDD) , ovvero pensare a come assegnare le responsabilità a degli oggetti che collaborano.

Nel complesso si ha quindi:



Un modo comune di pensare alla progettazione di oggetti software, ma anche di componenti su larga scala, è in termini di responsabilità, ruoli e collaborazioni. Questi aspetti fanno parte di un approccio più ampio chiamato **progettazione guidata dalle responsabilità o RDD (Responsibility-Driven Development)** [WM02].

Nella RDD, gli oggetti software sono considerati come dotati di responsabilità; per responsabilità si intende un'astrazione di ciò che fanno. UML definisce una responsabilità come "un contratto o un obbligo di un classificatore" [OMG03b]. Le responsabilità sono correlate agli obblighi o al comportamento di un oggetto in relazione al suo ruolo. Le responsabilità sono

fondamentalmente di due tipi: *di fare e di conoscere*. Le responsabilità “di fare” di un oggetto comprendono:

- fare qualcosa esso stesso, come per esempio creare un oggetto o eseguire un calcolo
- dare inizio a un’azione in altri oggetti
- controllare e coordinare le attività di altri oggetti

quelle “di conoscere”:

- conoscere i propri dati privati encapsulati
- conoscere gli oggetti correlati
- conoscere cose che può derivare o calcolare

**Le responsabilità sono assegnate alle classi di oggetti durante la progettazione a oggetti.**

*Per gli oggetti software di dominio, il modello di dominio spesso ispira le responsabilità più importanti relative al "conoscere", grazie agli attributi e alle associazioni che mostra.*

La traduzione delle responsabilità in classi e metodi è influenzata dalla granularità della responsabilità. Le responsabilità più grandi coinvolgono centinaia di classi e metodi, mentre le responsabilità minori possono coinvolgere un solo metodo; **una responsabilità non è la stessa cosa che un metodo, è un’astrazione; tuttavia i metodi soddisfano responsabilità.**

La RDD comprende anche l’idea di **collaborazione**. Le responsabilità sono implementate per mezzo di metodi che agiscono da soli oppure che collaborano con altri metodi e oggetti.

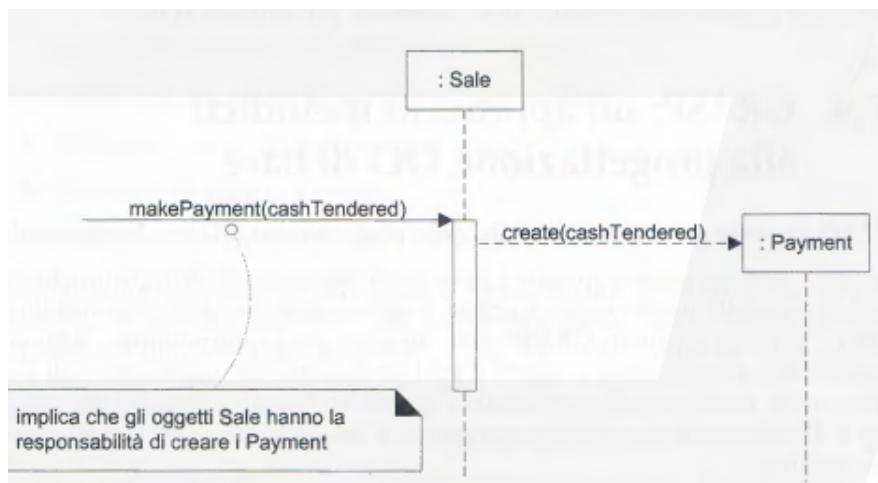
**La RDD è una metafora generale per pensare alla progettazione del software OO. Si pensi agli oggetti software come simili a persone che hanno delle responsabilità e che collaborano con altre persone per svolgere un lavoro. La RDD porta a considerare una progetto OO come una comunità di oggetti con responsabilità che collaborano.**

I pattern GRASP danno un nome e descrivono alcuni principi di base per assegnare le responsabilità, quindi è utile conoscerli, per sostenere la RDD.

I principi o pattern GRASP sono un aiuto per l’apprendimento degli aspetti essenziali della progettazione a oggetti e per l’applicazione dei ragionamenti di progettazione in un modo metodico, razionale e spiegabile. Questo approccio alla comprensione e all’utilizzo dei principi di progettazione si basa

su pattern per l'assegnazione di responsabilità.

Le decisioni sull'assegnazione delle responsabilità agli oggetti possono essere prese mentre si esegue la codifica oppure durante la modellazione. Nell'ambito di UML, il disegnare i diagrammi di interazione diventa l'occasione per considerare tali responsabilità (realizzate come metodi). Vediamo un esempio:



Pertanto, quando si disegna un diagramma di interazione di UML, vengono prese delle decisioni riguardo all'assegnazione di responsabilità. Questo capitolo sottolinea i principi fondamentali, espressi come pattern GRASP, per guidare le scelte nell'assegnazione delle responsabilità. In tal modo è possibile applicare i principi GRASP mentre si disegnano i diagrammi di interazione di UML, ma anche durante la codifica.

Gli sviluppatori 00 esperti (e altri sviluppatori del software) hanno accumulato (e continuano a farlo) un repertorio contenente sia principi generali che soluzioni idiomatiche che li guidino nella creazione del software. Questi principi e idiomi, se codificati in un formato strutturato che descrive il problema e la soluzione e a cui è assegnato un nome, possono essere chiamati **pattern**. Un pattern può essere per esempio:

Nome del pattern:	<b>Information Expert</b>
Problema:	Qual è un principio di base con cui assegnare responsabilità agli oggetti?
Soluzione:	Assegna una responsabilità alla classe che ha le informazioni necessarie per soddisfarla.

Nella progettazione 00, un pattern è una descrizione, con un nome, di un problema e una soluzione che può essere applicata a nuovi

contesti; idealmente, un pattern dà consigli su come applicare la sua soluzione in circostanze diverse e considera le forze e i compromessi. Molti pattern, data una categoria specifica di problemi, guidano l'assegnazione di responsabilità agli oggetti. *È semplicemente, un buon pattern è una coppia problema/soluzione ben conosciuta e con un nome, che può essere applicata in nuovi contesti, con consigli su come applicarla in situazioni nuove e con una discussione sui relativi compromessi, implementazioni, variazioni e così via.*

Dare un nome a un pattern, a un'idea di progetto o a un principio presenta i seguenti vantaggi:

- sostiene la segmentazione e l'assimilazione di quel concetto nella nostra mente e memoria
- facilita la comunicazione

*I pattern GRASP non affermano nuove idee, ma assegnano un nome e codificano dei principi di base ampiamente utilizzati. A un esperto di progettazione OO, i pattern GRASP appaiono fondamentali e familiari, per l'idea se non per il nome.*

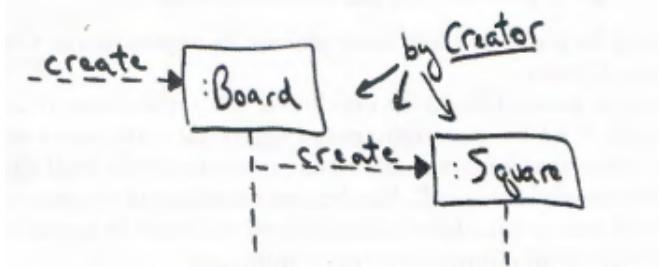
GRASP definiscono nove principi di progettazione OO di base o blocchi di costruzione elementari della progettazione. Vediamo i primi 5:

1. **Creator:** che tratta il creatore di un oggetto, ovvero di una responsabilità “di fare”. Si deve sostenere un **salto rappresentazionale basso** (*LRG*, per Low Representational Gap) tra il modo in cui si pensa al dominio e una corrispondenza diretta con gli oggetti software. Abbiamo quindi la definizione:

Nome:	<b>Creator</b> (Creatore)
Problema:	Chi crea un oggetto A?
Soluzione:	Assegna alla classe B la responsabilità di creare un'istanza della classe A se (può essere vista come un consiglio) ■ B “contiene” o aggrega con una composizione oggetti di tipo A. ■ B registra A. ■ B utilizza strettamente A. ■ B possiede i dati per l'inizializzazione di A.

Si nota che si ha a che fare con l'assegnazione di responsabilità. Innanzitutto, un punto delicato ma importante nell'applicazione di

Creatore degli altri pattern GRASP: B ed A fanno riferimento a oggetti software, non a oggetti del modello di dominio. Dapprima si tenta di applicare Creator cercando degli oggetti software esistenti che soddisfano il ruolo di B. Ma che cosa succede se la progettazione è appena iniziata e non è ancora stata definita alcuna classe software? In questo caso, per LRG, va guardato il modello di dominio per trarre ispirazione. A livello pratico



La creazione degli oggetti è una delle attività più comuni in un sistema orientato agli oggetti. Di conseguenza, è utile avere un principio generale per l'assegnazione delle responsabilità di creazione. Se queste vengono assegnate bene, il progetto può sostenere un accoppiamento basso, una chiarezza maggiore, encapsulamento e riusabilità. Quindi si segna alla classe B la responsabilità di creare un'istanza della classe A se una delle seguenti condizioni è vera (più sono vere, meglio è):

- B "contiene" o aggrega con una composizione oggetti di tipo A
- B registra A
- B utilizza strettamente A
- B possiede i dati per l'inizializzazione di A, che saranno passati ad A al momento della sua creazione. Pertanto B è un Expert rispetto alla creazione di A

B è un creatore di oggetti A. Se sono applicabili più opzioni, solitamente va preferita una classe B che aggredisce o contiene la classe A. *Creator guida l'assegnazione delle responsabilità relative alla creazione di oggetti, un'attività molto comune. Lo scopo fondamentale del pattern Creator è trovare un creatore che abbia bisogno, in ogni caso, di essere connesso all'oggetto creato. La scelta di un creatore con questa caratteristica favorisce l'accoppiamento basso.* È favorito l'accoppiamento basso, il che implica minori dipendenze di manutenzione e maggiori opportunità di riuso. I...accoppiamento probabilmente non aumenta,

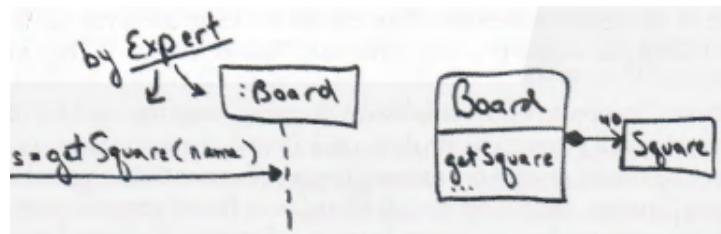
poiché la classe creata deve essere probabilmente già visibile alla classe creatore, grazie alle associazioni esistenti che ne hanno motivato la scelta come creatore.

2. **Information Expert:** è uno dei principi di base per l'assegnazione delle responsabilità nella progettazione a oggetti.

*Expert spiega chi conosce un oggetto data una chiave.* vediamo la definizione:

Nome:	<b>Information Expert</b> (Esperto delle Informazioni)
Problema:	Qual è un principio di base per assegnare responsabilità agli oggetti?
Soluzione: (consiglio)	Assegna una responsabilità alla classe che possiede le informazioni necessarie per soddisfarla.

Una responsabilità necessita di informazioni per essere soddisfatta: informazioni su altri oggetti, sullo stato di un oggetto, sul mondo che circonda l'oggetto, informazioni che l'oggetto può ricavare e così via. A livello pratico:



Un Modello di Progetto può definire centinaia o migliaia di classi software, e un'applicazione può richiedere centinaia o migliaia di responsabilità da soddisfare. Durante la progettazione a oggetti, quando vengono definite le interazioni tra gli oggetti, si effettuano delle scelte sull'assegnazione delle responsabilità alle classi software. Se le scelte vengono fatte bene, i sistemi tendono a essere più facili da comprendere, da mantenere e da estendere, e le scelte fatte consentono maggiori opportunità di riusare i componenti in applicazioni future.

**Assegna una responsabilità all'esperto delle informazioni, ovvero alla classe che possiede le informazioni necessarie per soddisfare la responsabilità.** Per Information Expert, si dovrebbe cercare la classe degli oggetti che possiede le informazioni necessarie per determinare il totale. Inoltre se ci sono già delle classi pertinenti nel Modello di Progetto, si analizzi questo per pmo altrimenti si

guardi nel Modello di Dominio, cercando di usare (o di estendere) le sue rappresentazioni per ispirare la creazione di classi di progetto corrispondenti.

Information Expert è usato frequentemente nell'assegnazione di responsabilità; è un principio guida di base, usato continuamente nella progettazione degli oggetti. Expert non intende essere un'idea oscura o capricciosa; esso esprime una "intuizione" comune secondo cui gli oggetti fanno cose correlate alle informazioni di cui dispongono. Si noti che la soddisfazione di una responsabilità spesso richiede informazioni che si trovano sparse tra varie classi di oggetti. Ciò implica che molti esperti delle informazioni "parziali" dovranno collaborare al compito. Per esempio, il problema del totale di una vendita ha richiesto, in definitiva, la collaborazione di tre classi di oggetti. Ogni volta che le informazioni sono sparse tra diversi oggetti, questi avranno bisogno di interagire attraverso messaggi per condividere il lavoro.

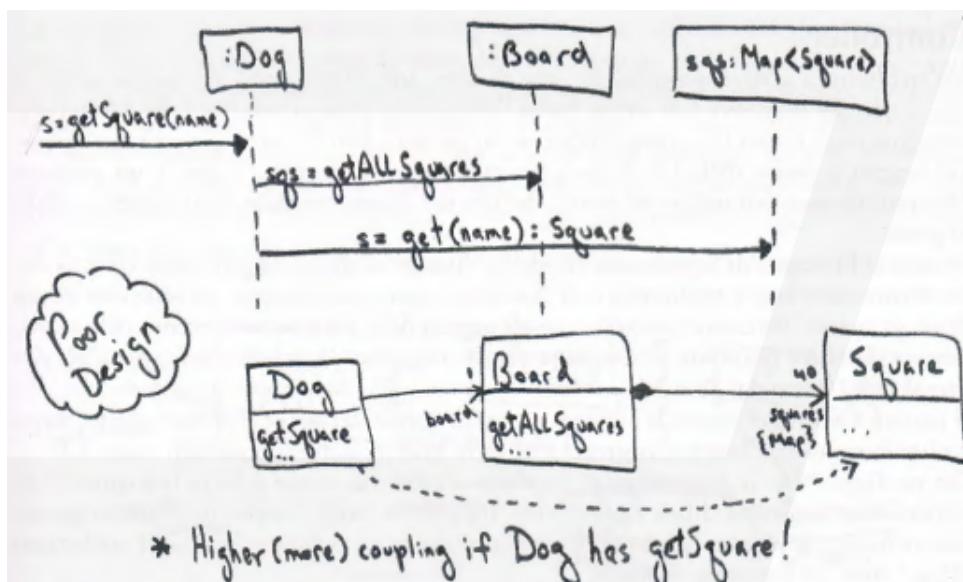
In alcune situazioni, una soluzione suggerita da Expert non è opportuna, solitamente a causa di problemi di accoppiamento e di coesione. SI hanno del resto due vantaggi:

- (a) l'incapsulamento delle informazioni viene mantenuto, poiché gli oggetti usano le proprie informazioni per adempiere ai propri compiti. Di solito questo sostiene un accoppiamento basso, che dà luogo a sistemi più robusti e mantenibili
- (b) il comportamento è distribuito tra tutte le classi che possiedono le informazioni richieste, incoraggiando in tal modo definizioni di classe più coese e "leggere", più facili da comprendere e da mantenere. Di solito è sostenuta una coesione alta

**3. Low Coupling:** spiega perché Expert è un principio utile e fondamentale della progettazione orientata agli oggetti. In modo breve e informale, l'accoppiamento è una misura di quanto fortemente un elemento è connesso ad altri elementi, li conosce o dipende da essi. Se esiste un accoppiamento o una dipendenza, quando l'elemento da cui si dipende subisce una modifica, ciò può ripercuotersi sull'elemento dipendente. Per esempio, una sottoclassificazione è fortemente accoppiata a una superclasse; un oggetto A che richiama le operazioni dell'oggetto B ha un accoppiamento ai servizi di B. Il principio del Low Coupling si applica a molte dimensioni nello sviluppo del software; è veramente uno degli obiettivi cardinali nella costruzione del software. In termini di progettazione a oggetti e di responsabilità, si può descrivere il consiglio come segue:

Nome:	<b>Low Coupling</b> (Accoppiamento Basso)
Problema:	Come ridurre l'impatto dei cambiamenti?
Soluzione: (consiglio)	Assegna le responsabilità in modo tale che l'accoppiamento (non necessario) rimanga basso. Usa questo principio per valutare le alternative.

Low Coupling è utilizzato per valutare progetti esistenti o per valutare la scelta tra nuove alternative; se tutte le altre cose sono uguali, va preferito un progetto in cui l'accoppiamento è più basso che nelle alternative. Nella pratica:



Per tornare alle motivazioni per Information Expert: esso guida verso una scelta che sostiene Low Coupling. Expert chiede di trovare l'oggetto che possiede la maggior parte delle informazioni richieste per la responsabilità (per esempio, Board) e di assegnare ad esso la responsabilità.

L'accoppiamento (coupling) è una misura di quanto fortemente un elemento è connesso ad altri elementi, ha conoscenza di altri elementi e dipende da altri elementi. Un elemento con un accoppiamento basso (o debole) non dipende da troppi altri elementi; "troppi" dipende dal contesto, e questo aspetto sarà esaminato più avanti. Per elementi si intendono classi, sottosistemi, sistemi e così via. Una classe con un accoppiamento alto (o forte) dipende da molte altre classi. Tali clas-

si fortemente accoppiate possono essere inopportune, e alcune di esse presentano i seguenti problemi:

- i cambiamenti nelle classi correlate obbligano a cambiamenti locali
- sono più difficili da comprendere in isolamento
- sono più difficili da riusare, poiché il loro uso richiede la presenza aggiuntiva delle classi da cui dipendono

**Assegna una responsabilità in modo che l'accoppiamento rimanga basso. Usa questo principio per valutare le alternative.**

*In pratica il livello di accoppiamento da solo non può essere considerato separatamente da altri principi quali Expert e High Cohesion. Nondimeno, è uno dei fattori da tenere in considerazione per migliorare un progetto.*

Low Coupling è un principio da tenere a mente durante tutte le decisioni di progetto; è un obiettivo basilare da tenere continuamente in considerazione. È un principio di valutazione da applicare quando si valutano tutte le decisioni della progettazione.

Low Coupling incoraggia ad assegnare una responsabilità in modo tale che la sua collocazione non aumenti l'accoppiamento a un livello tale da portare ai risultati negativi causati dall'accoppiamento alto. Low Coupling sostiene la progettazione di classi tra loro più indipendenti, riducendo così l'impatto dei cambiamenti. Esso non può essere considerato separatamente da altri pattern quali Expert e High Cohesion, ma piuttosto necessita di essere incluso come uno dei vari principi di progettazione che influenzano la scelta nell'assegnazione di una responsabilità.

**Un accoppiamento alto a elementi stabili e a elementi pervasivi costituisce raramente un problema.**

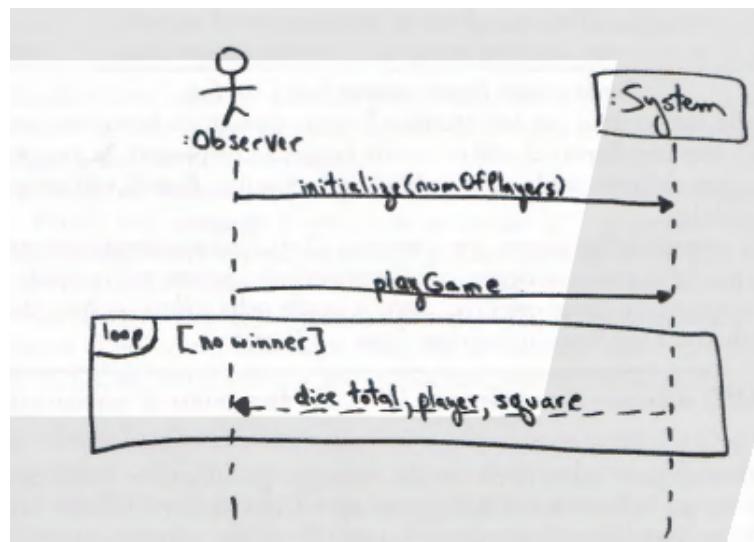
Ma si hanno i seguenti vantaggi:

- non influenzato dai cambiamenti negli altri componenti.
- semplice da capire separatamente dagli altri componenti.
- conveniente da riusare.

4. **Controller:** infatti un'architettura a strati semplice ha, tra gli altri, uno strato per l'interfaccia utente e uno strato del dominio. In base al Princípio di Separazione Modello-Vista, si sa che gli oggetti della UI non devono contenere logica applicativa o di "business", come per esempio calcolare una mossa di un giocatore. Pertanto, una volta che gli oggetti della UI rilevano l'evento del mouse, devono delegare (inoltrare

il compito a un altro oggetto) la richiesta agli oggetti di dominio nello strato del dominio. Il pattern Controller risponde a questa semplice domanda: *qual è il primo oggetto, dopo o oltre lo strato dell'interfaccia utente, che deve ricevere il messaggio dallo strato UI?*

Ovvero:



Pertanto, Controller si occupa di una domanda di base nella progettazione OO: *come connettere lo strato UI allo strato della logica applicativa?*

In alcuni metodi dell'OOA/D, il nome controller è stato dato all'oggetto della logica applicativa che riceve e "controlla" (coordina) la gestione della richiesta. Vediamo la descrizione:

Nome:	<b>Controller</b> (Controllore)
Problema:	Qual è il primo oggetto oltre lo strato UI a ricevere e coordinare ("controllare") un'operazione di sistema?
Soluzione: (consiglio)	<p>Assegna la responsabilità a un oggetto che rappresenta una delle seguenti scelte.</p> <ul style="list-style-type: none"> <li>■ Rappresenta il "sistema" complessivo, un "oggetto radice", un dispositivo all'interno del quale viene eseguito il software o un sottosistema principale (sono tutte varianti di un <i>facade controller</i>).</li> <li>■ Rappresenta uno scenario di un caso d'uso all'interno del quale si verifica l'operazione di sistema (un <i>controller di caso d'uso</i> o <i>controller di sessione</i>)</li> </ul>

Un controller è il primo oggetto oltre lo strato UI che è responsabile di ricevere o gestire un messaggio di un'operazione di sistema. Si assegna la responsabilità a una classe che rappresenta una delle seguenti scelte:

- (a) rappresenta il "sistema" complessivo, un "oggetto radice", un dispositivo all'interno del quale viene eseguito il software, o un sorrosistema principale; sono tutte varianti di un facade controller
- (b) rappresenta uno scenario di un caso d'uso all'interno del quale si verifica l'evento di sistema, spesso chiamato <UseCaseName>Handler, <UseCaseName>Coordinator, o <UseCaseName>Session (controller di caso d'uso o controller di sessione)

inoltre:

- si utilizzi la stessa classe controller per tutti gli eventi di sistema nello stesso scenario di caso d'uso
- informalmente, una sessione è un'istanza di una conversazione con un attore. Le sessioni possono avere una lunghezza qualsiasi, ma spesso sono organizzate in termini di casi d'uso (sessioni di caso d'uso).

Si hanno i seguenti vantaggi:

- maggiore potenziale di riuso e interfacce inseribili. Questi vantaggi assicurano che la logica applicativa non sia gestita nello strato dell'interfaccia. Le responsabilità di un controller potrebbero essere gestite, tecnicamente, in un oggetto dell'interfaccia, ma un tale progetto implicherebbe che il codice del programma e l'esecuzione della logica applicativa sono incorporati negli oggetti dell'interfaccia (nelle finestre). Un progetto "interfaccia come controller" riduce l'opportunità di riusare la logica in altre applicazioni future, poiché la logica che è legata a una particolare interfaccia (per esempio, oggetti di tipo finestra) può essere difficilmente riutilizzata in altre applicazioni. Al contrario, la delega delle responsabilità delle operazioni di sistema a un controller favorisce il riuso della logica in altre applicazioni future. E dal momento che la logica applicativa non è legata allo strato dell'interfaccia, è possibile anche utilizzarla con un'interfaccia diversa
- opportunità di ragionare sullo stato del caso d'uso . A volte è necessario assicurarsi che le operazioni di sistema si susseguano in una sequenza legale, oppure si desidera poter ragionare sullo

stato corrente dell'attività e delle operazioni all'interno del caso d'uso in corso di esecuzione. Per esempio, se bisogna garantire che l'operazione makePayment non possa essere eseguita fino a che non è stata eseguita l'operazione endSale. In questo caso, occorre catturare da qualche parte queste informazioni sullo stato; il controller è una scelta ragionevole, soprattutto se lo stesso controller è utilizzato per l'intero caso d'uso (come consigliato)

e progettata in modo mediocre, una classe controller ha una coesione bassa, poiché non è focalizzata e gestisce responsabilità in troppe aree; si parla in questo caso di controller gonfio, con le seguenti caratteristiche:

- nel sistema c'è un'unica classe controller che riceve tutti gli eventi di sistema, che sono numerosi. Ciò avviene talvolta quando viene scelto un facade controller.
- il controller stesso svolge molti dei compiti necessari per soddisfare l'evento di sistema, senza delegare il lavoro. Ciò solitamente coinvolge una violazione di Information Expert e High Cohesion.
- un controller ha numerosi attributi, e conserva informazioni significative sul sistema o sul dominio, che avrebbero dovuto essere distribuite ad altri oggetti, oppure duplica informazioni trovate altrove.

Con due possibili soluzioni:

- (a) aggiungere più controller; un sistema non deve necessariamente avere un solo controller. Anziché un facade controller, si utilizzino dei controller di caso d'uso
  - (b) progettare il controller in modo tale che esso, in primo luogo, deleghi la soddisfazione della responsabilità di ciascuna operazione di sistema agli altri oggetti
5. **High Cohesion:** ovvero utile per una qualità fondamentale nota come **coesione** è (informalmente) una misura di quanto sono correlate le operazioni di un elemento software da un punto di vista funzionale, e altresì una misura di quanto lavoro sta eseguendo un elemento software. In definitiva, sia la quantità di codice che la correlazione del codice sono indicatori della coesione di un oggetto. Questo porta al principio di High Cohesion, che va utilizzato per valutare scelte di progetto diverse. Se tutte le altre cose sono equivalenti, va preferito un progetto con una coesione più elevata:

Nome:	<b>High Cohesion</b> (Coesione Alta)
Problema:	Come mantenere gli oggetti focalizzati, comprensibili e gestibili e, com e effetto collaterale, sostenere Low Coupling?
Soluzione: (consiglio)	Assegna le responsabilità in modo tale che la coesione rimanga alta. Usa questo principio per valutare le alternative.
Si può affermare che il progetto sulla destra sostiene High Cohesion meglio della versione sulla sinistra.	

In termini di progettazione a oggetti, la coesione (o più specificamente la coesione funzionale) è una misura di quanto fortemente siano correlate e concentrate le responsabilità di un elemento. Un elemento con responsabilità altamente correlate che non esegue una quantità di lavoro eccessiva ha una coesione alta. Per elementi si intendono classi, sottosistemi e così via. Si ha la seguente soluzione: assegna una responsabilità in modo tale che la coesione rimanga alta. Usa questo principio per valutare le alternative. Una classe con una coesione bassa fa molte cose non correlate tra loro o svolge troppo lavoro. Tali classi non sono opportune, e presentano i seguenti problemi:

- sono difficili da comprendere.
- sono difficili da riusare.
- sono difficili da mantenere.
- sono delicate; sono continuamente soggette a cambiamenti

Le classi a coesione bassa spesso rappresentano un'astrazione a "grana molto grossa" o hanno assunto responsabilità che avrebbero dovuto essere delegate ad altri oggetti.

**In alcuni casi, è giustificabile accettare una coesione più bassa.**  
Si hanno i seguenti vantaggi:

- maggiore chiarezza e facilità di comprensione del progetto
- la manutenzione e i miglioramenti risultano semplificati
- low coupling è spesso sostenuto
- maggiore riuso di funzionalità a grana fine e altamente correlate, poiché una classe coesa può essere usata per uno scopo molto specifico.

Torniamo ora a **GRASP**, che è un acronimo per *General Responsibility Assignment Software Patterns* ovvero di Pattern Generali per l'Assegnazione di

## *Capitolo 11. GRASP*

---

Responsabilità nel Software. Il nome è stato scelto per suggerire l'importanza di capire, afferrare (in inglese, grasp) questi principi per poter progettare con successo il software orientato agli oggetti. Comprendere ed essere in grado di applicare le idee che stanno alla base dei pattern GRASP, mentre si codifica o mentre si disegnano i diagrammi di interazione e delle classi, consente agli sviluppatori per i quali le esigenze della tecnologia a oggetti costituiscono una novità, di acquisire il più rapidamente possibile la padronanza di questi principi di base; essi costituiscono un fondamento per la progettazione di sistemi OO. I 4 pattern mancabnti sono: *Polymorphism*, *Pure Fabrication*, *Indirection* e *Protected variations*.

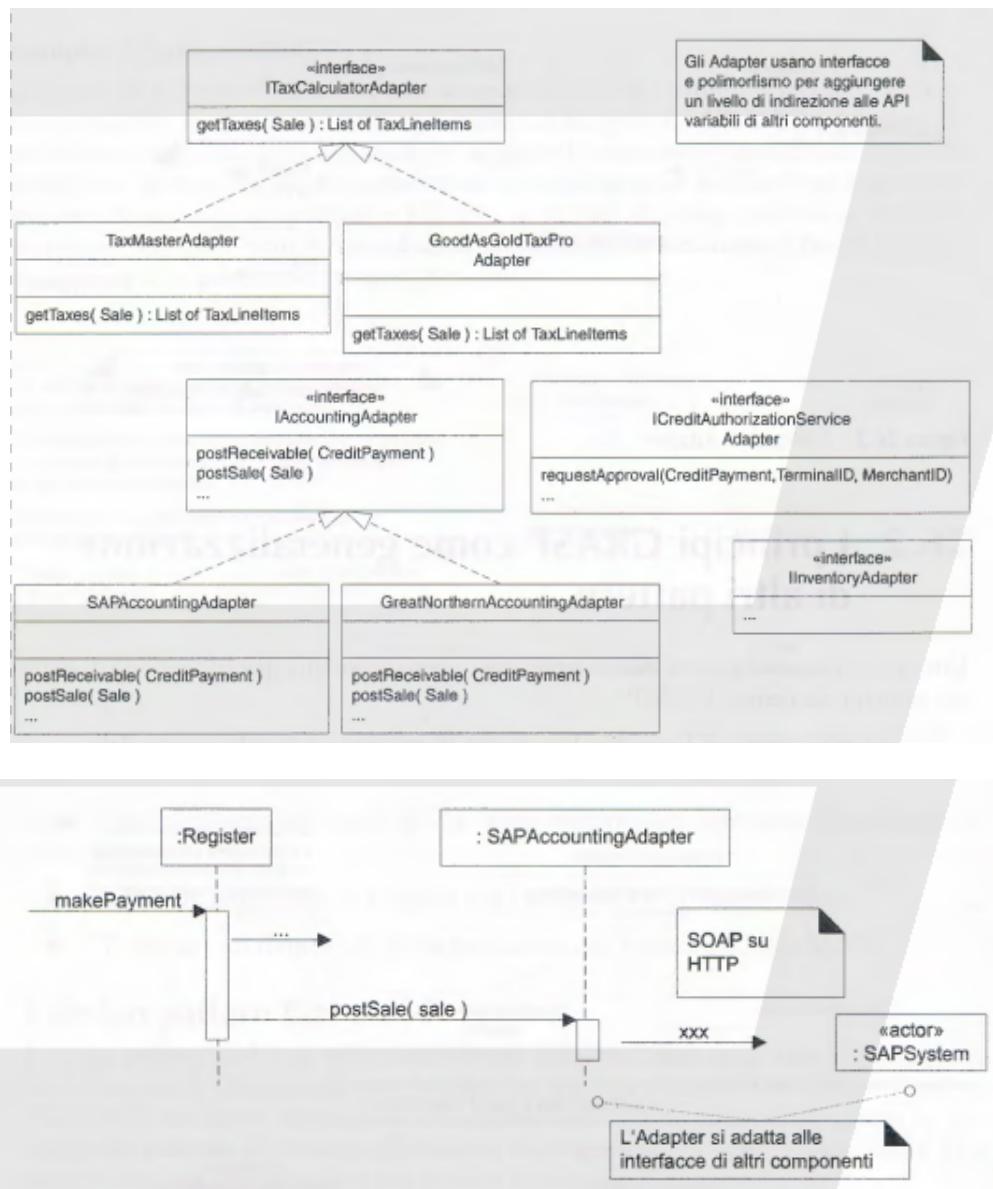
## Capitolo 12

# Applicare i design pattern GoF

L'accento è posto su come applicare i design pattern Gang-of-Four (GoF) e i pattern GRASP più di base.

Vediamo un esempio di pattern GoF:

Nome:	<b>Adapter</b>
Problema:	Come gestire interfacce incompatibili, o fornire un'interfaccia stabile a componenti simili con interfacce diverse?
Soluzione: (consiglio)	Converti l'interfaccia originale di un componente in un'altra interfaccia, attraverso un oggetto adattatore intermedio.



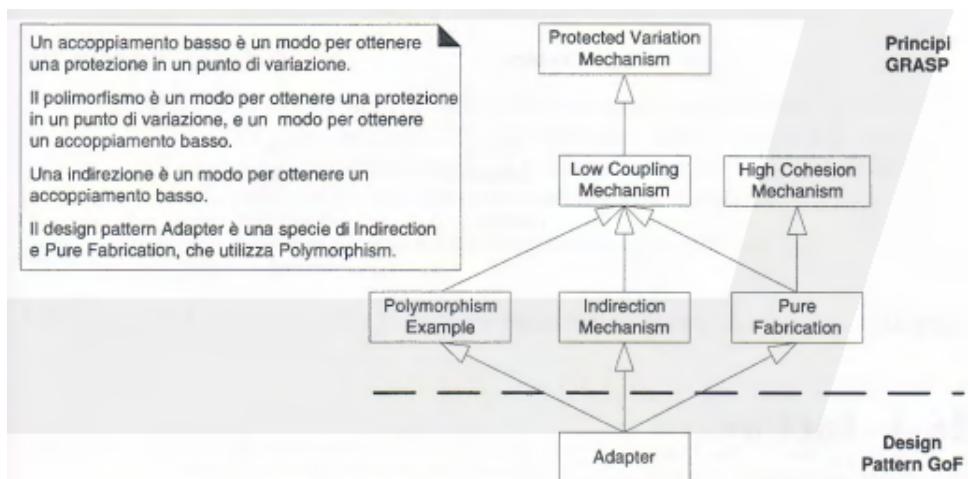
i noti che i nomi dei tipi comprendono il nome del pattern “Adapter”. Si tratta di uno stile relativamente comune che ha il vantaggio di comunicare facilmente agli altri che leggono il codice o i diagrammi quali sono i design pattern utilizzati.

Un adattatore di una risorsa che nasconde un sistema esterno può anche essere considerato un oggetto Facade, design pattern Gof che verrà discusso, poiché "avvolge" l'accesso al sottosistema o al sistema con un oggetto singolo (che è l'essenza di Facade). Tuttavia, la motivazione a chiamarlo un adattatore di risorsa sussiste in modo particolare quando l'oggetto che avvolge

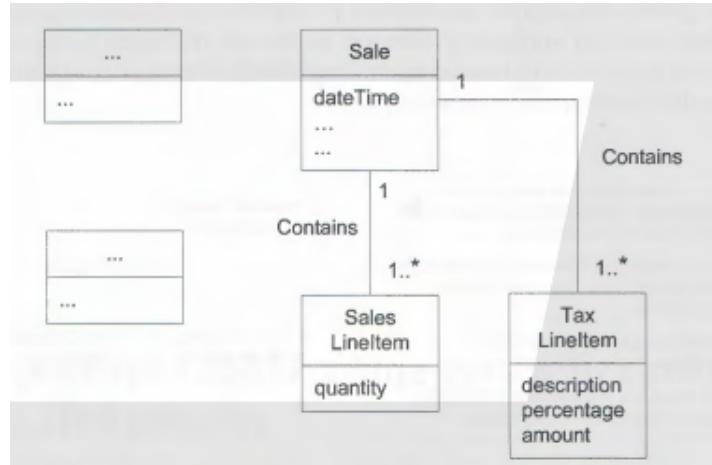
fornisce l'adattamento a interfacce esterne variabili.

L'uso precedente del pattern Adapter può essere visto come una specializzazione di alcuni dei principi elementari GRASP: *Adapter* supporta *Protected Variations* in relazione al cambiamento delle interfacce esterne o di package di terze parti attraverso l'uso di un oggetto *Indirection* che applica le interfacce e *Polymorphism*.

È importante per un progettista esperto conoscere nel dettaglio e a memoria oltre 50 dei design pattern più importanti, ma solo pochi possono imparare o ricordare anche 1000 pattern, oppure iniziare a organizzare quella pletora di pattern in una tassonomia utile. La bella notizia è che la maggior parte dei design pattern può essere vista come una specializzazione di alcuni principi GRASP di base. Anche se è effettivamente utile studiare alcuni design pattern in modo dettagliato per accelerare l'apprendimento, è ancora più utile vedere i temi che stanno alla loro base (Protected Variations, Polymorphism, Indirection, ... ) per avere un aiuto nel ridurre le miriadi di dettagli e vedere "l'alfabeto" essenziale delle tecniche di progettazione che vengono applicate. Per esempio:



È assolutamente normale e comune scoprire concetti di dominio significativi e una comprensione raffinata dei requisiti durante la progettazione e la programmazione: lo sviluppo iterativo sostiene questo tipo di scoperta incrementale. Se il Modello di Dominio sarà utilizzato in futuro come fonte di ispirazione per successivi lavori di progettazione, o come aiuto visuale di apprendimento per comunicare i concetti chiave del dominio, in quel caso aggiungerlo potrebbe avere un valore. Vediamo un modello di dominio aggiornato:



Vediamo il pattern chiamato **Simple Factory o Concrete Factory**.

Non si tratta di un design pattern GoF vero e proprio, ma è estremamente diffuso. Si tratta piuttosto di una semplificazione del design pattern GoF Abstract Factory), e spesso viene descritto come una sua variante, anche se ciò non è completamente vero. Tuttavia, viene presentato a questo punto per la sua larga diffusione e per la sua associazione con GoF.

l'uso dell'adattatore fa sorgere un nuovo problema nella progettazione: nella precedente soluzione basata sul pattern Adapter per i servizi esterni con interfacce variabili, *chi crea gli adattatori? E come stabilire quale classe di adattatore creare?*

Se questi vengono creati da un oggetto di dominio, le responsabilità dell'oggetto di dominio vanno oltre la logica applicativa pura (come il calcolo del totale di una vendita) ed entrano in altri interessi relativi alla connessione con i componenti software esterni. Questo punto sottolinea un altro principio fondamentale di progettazione (solitamente considerato un principio di progettazione architettonale): progettare per mantenere una **separazione degli interessi**. Ciò significa modularizzare o separare interessi distinti in aree diverse, in modo che ciascuno abbia uno scopo coeso. Sostanzialmente è un'applicazione del principio GRASP High Cohesion.

Pertanto la scelta di un oggetto di dominio (come Register) per creare gli adattatori non sostiene l'obiettivo della separazione degli interessi, e riduce la sua coesione. Un'alternativa comune in questo caso consiste nell'applicare il pattern Factory, in cui viene definito un oggetto "factory" ("fabbrica") Pure Fabrication per creare gli oggetti. Gli oggetti factory presentano diversi vantaggi:

- separano le responsabilità delle creazioni complesse in oggetti di supporto coesi

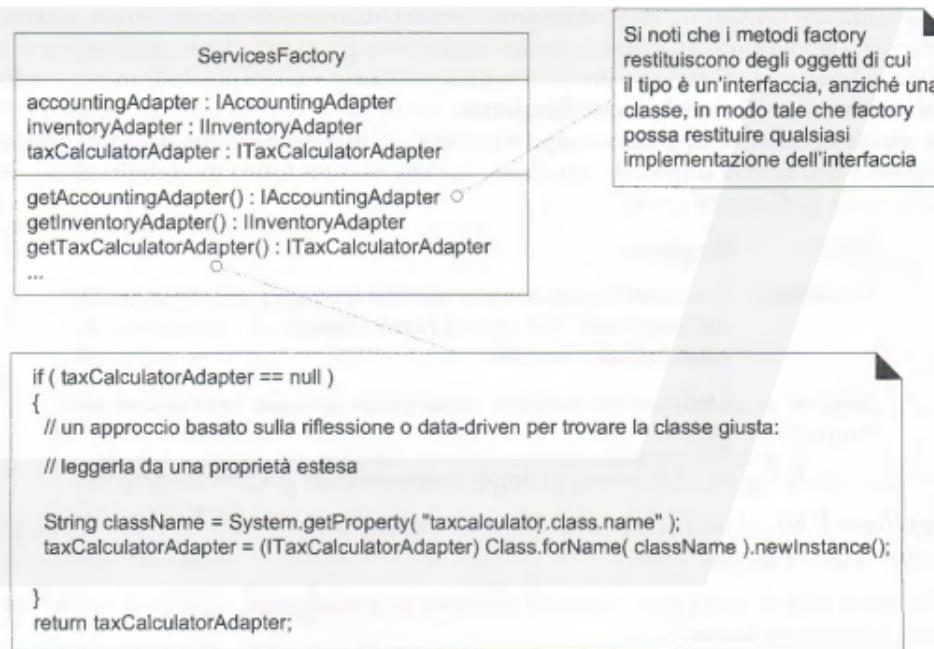
- nascondono la logica di creazione potenzialmente complessa.
- consentono l'introduzione di strategie per la gestione della memoria che possono migliorare le prestazioni, come il caching o il riciclaggio degli oggetti.

Nome: **Factory**

Problema: chi deve essere responsabile della creazione di oggetti quando ci sono delle considerazioni speciali, come una logica di creazione complessa, quando si desidera separare le responsabilità di creazione per una coesione migliore, e così via?

Soluzione: Crea un oggetto Pure Fabrication chiamato una Factory che (consiglio) gestisce la creazione.

con il seguente esempio:



Si noti che in ServicesFactory, la logica per stabilire quale classe creare è risolta leggendo il nome della classe da una sorgente esterna (per esempio da una proprietà di sistema, se viene usato Java), quindi caricando dinamicamente la classe. Questo è un esempio di **progettazione guidata dai dati parziale**. Passiamo ora al pattern GoF **Singleton**.

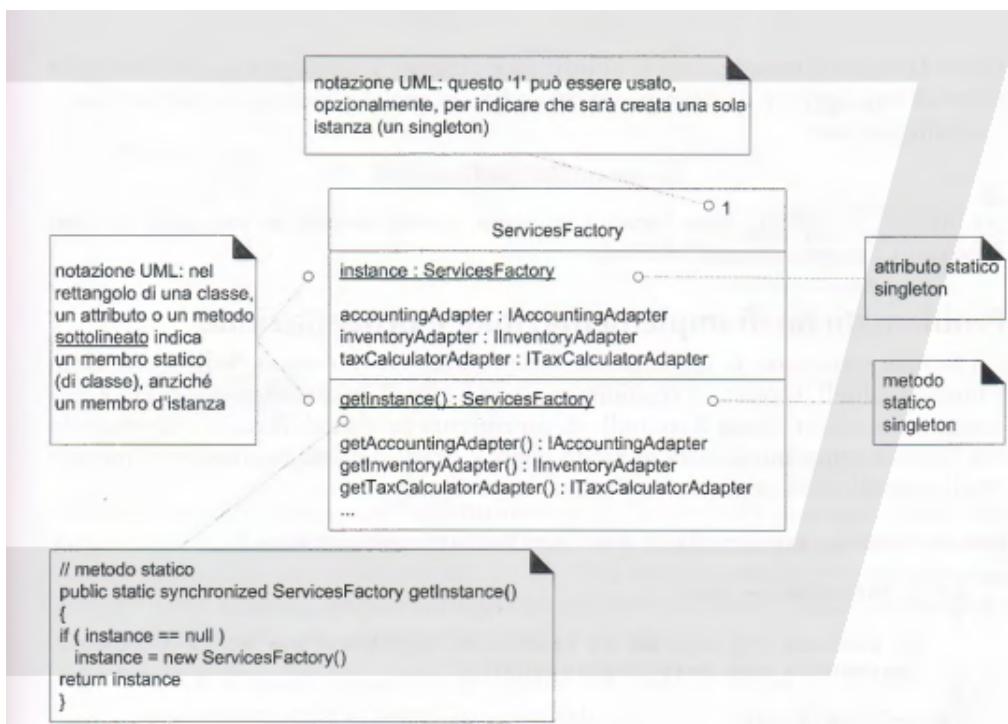
Innanzitutto si osservi che all'interno del processo è necessaria una sola istanza della factory. In secondo luogo, una rapida riflessione suggerisce che i

metodi di questa factory devono poter essere richiamati da vari punti del codice, poiché diversi punti necessitano di accesso agli adattatori per avvalersi dei servizi esterni. C'è dunque un problema di visibilità: *come ottenere la visibilità verso questa singola istanza di ServicesFactory?*

Una soluzione consiste nel passare l'istanza ServicesFactory come parametro ovunque sia necessaria la visibilità della stessa, oppure inizializzare gli oggetti che necessitano di visibilità nei confronti di essa con un riferimento permanente. Ciò è possibile ma scomodo; un'alternativa è offerta dal pattern **Singleton**.

In alcune occasioni è opportuno supportare la visibilità globale o un punto di accesso singolo a una istanza singola di una classe, anziché un'altra forma di visibilità. È il caso dell'istanza di ServicesFactory:

Nome:	<b>Singleton</b>
Problema:	È consentita esattamente una sola istanza di una classe, ovvero un "singleton". Gli oggetti hanno bisogno di un punto di accesso globale e singolo.
Soluzione:	Definisci un metodo statico della classe che restituisce il singleton.



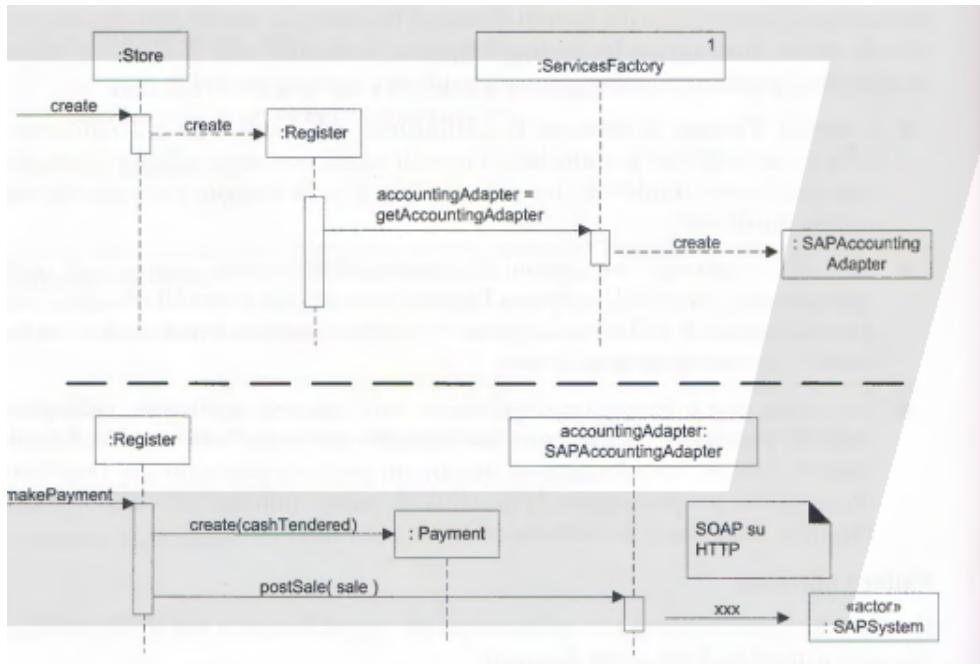
Un metodo `getInstance` di un Singleton viene chiamato molto spesso. Nelle applicazioni a thread multipli, il passo di creazione con una logica di **iniziazione**

**lizzazione pigra** è una sezione critica che richiede il controllo di concorrenza del thread. Pertanto, supponendo che l'istanza venga inizializzata in modo pigro, è prassi comune racchiudere il metodo con il controllo della concorrenza.

Un'istanza e un metodo d'istanza sono solitamente preferiti per i seguenti motivi:

- i metodi d'istanza consentono la ridefinizione nelle sottoclassi e il raffinamento della classe singleton in sottoclassi; i metodi statici non sono polimorfi (virtuali) e non consentono la ridefinizione nelle sottoclassi (nella maggior parte dei linguaggi, escluso Smalltalk)
- la maggior parte dei meccanismi di comunicazione remota orientati agli oggetti (per esempio, Java RMI) supporta l'accesso remoto solo a metodi d'istanza, e non a metodi statici. A un'istanza singleton si potrebbe accedere remotamente, anche se indubbiamente ciò avviene di rado
- una classe non è sempre un singleton in tutti i contesti applicativi. Nell'applicazione X, può essere un singleton, ma potrebbe essere un "multi-ton" nell'applicazione Y. Non è inusuale neppure iniziare un progetto pensando che l'oggetto sia un singleton, per poi scoprire la necessità di istanze multiple nello stesso processo. Pertanto, la soluzione lato istanza offre una certa flessibilità

Vediamo un'applicazione complessa con i vari pattern:



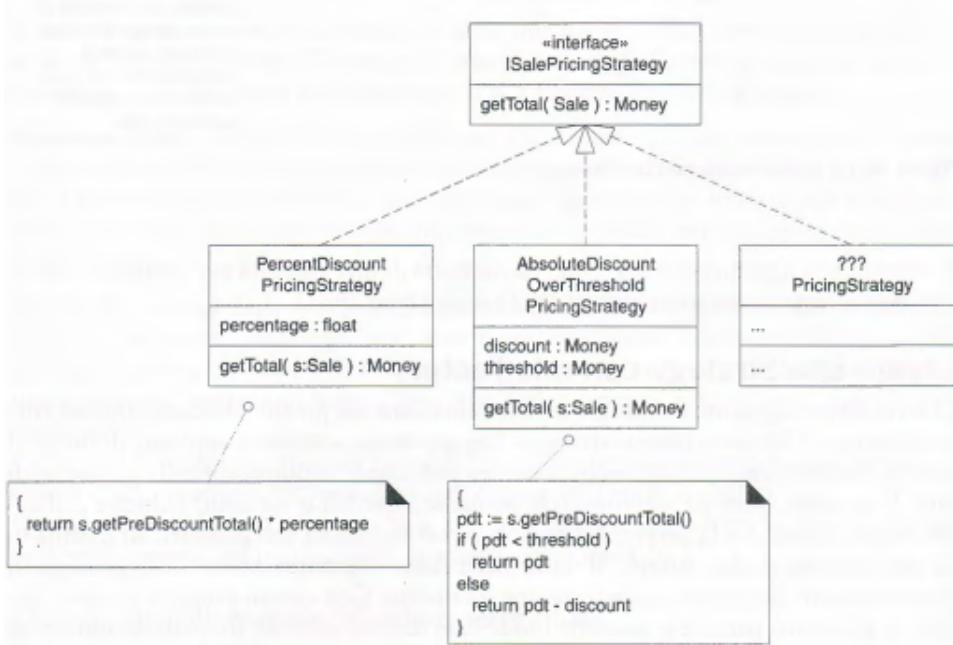
Il prossimo problema di progettazione da risolvere è fornire una logica più complessa per il nostro scopo. Abbiamo quindi il pattern **strategy**:

Nome: **Strategy**

Problema: Come progettare per gestire un insieme di algoritmi o politiche variabili ma correlati? Come progettare per consentire di modificare questi algoritmi o politiche?

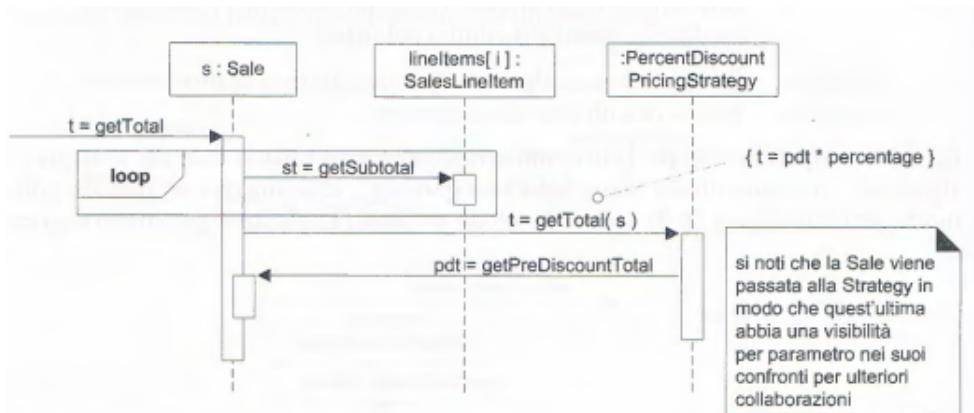
Soluzione: Definisci ciascun algoritmo/politica/strategia in una classe separata, con un'interfaccia comune.

ovvero:



Un oggetto strategia è associato a un oggetto contesto, ovvero l'oggetto a cui va applicato l'algoritmo.

Non è obbligatorio che il messaggio al- l' oggetto contesto e quello all'oggetto strategia abbiano lo stesso nome, ma si tratta di una prassi comune. Tuttavia, è comune (e di fatto necessario) che l'oggetto contesto passi un riferimento a se stesso (*this*) all'oggetto strategia, in modo che la strategia abbia una visibilità per parametro nei confronti dell'oggetto contesto, per ulteriori collaborazioni.



Una factory diversa da ServicesFactory può essere usata per creare una strategia, ciò sostiene l'obiettivo di High Cohesion: ciascuna factory è focalizzata in modo coeso sulla creazione di una famiglia correlata di oggetti. Quindi in

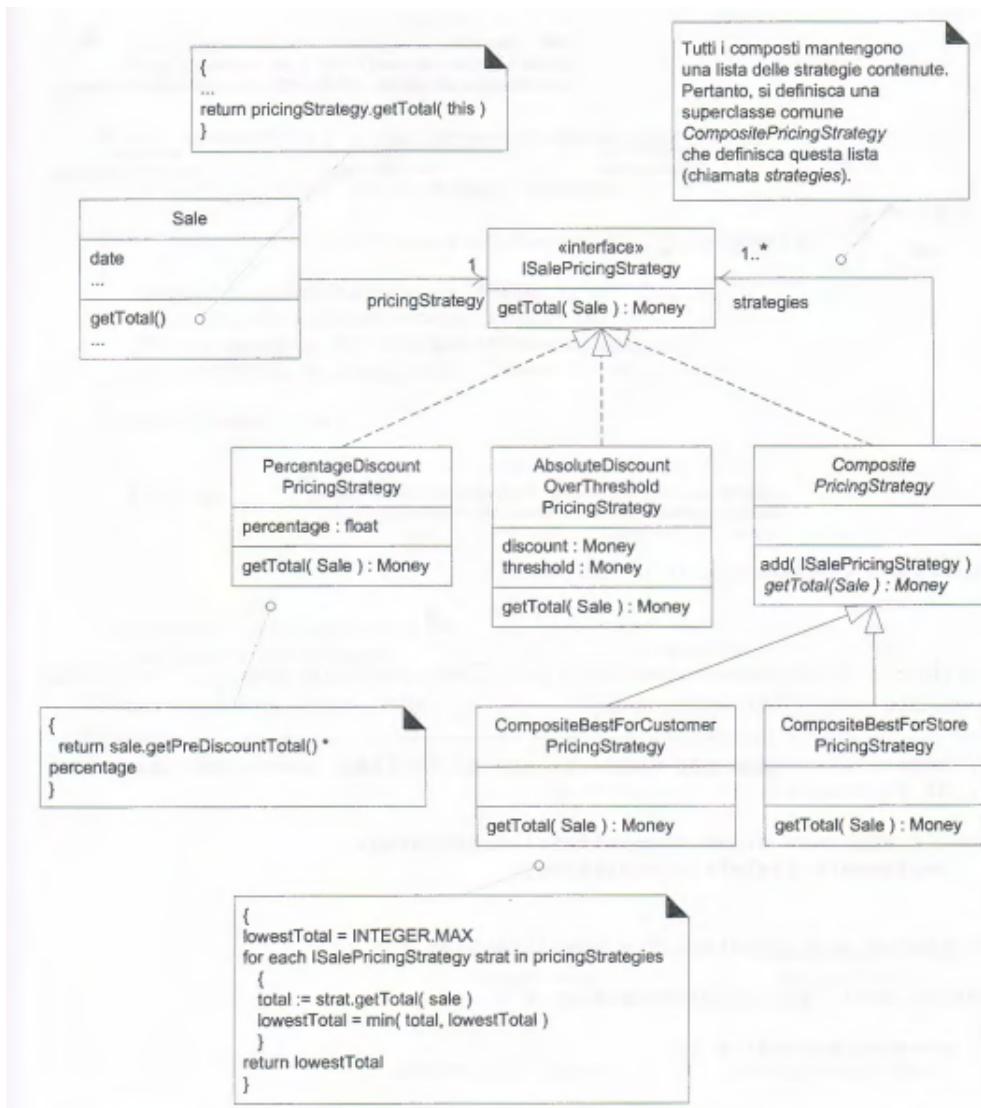
definitiva: *Strategy si basa su Polymorphism e fornisce Protected Variations rispetto agli algoritmi variabili. Le strategie sono spesso create da una Factory.*

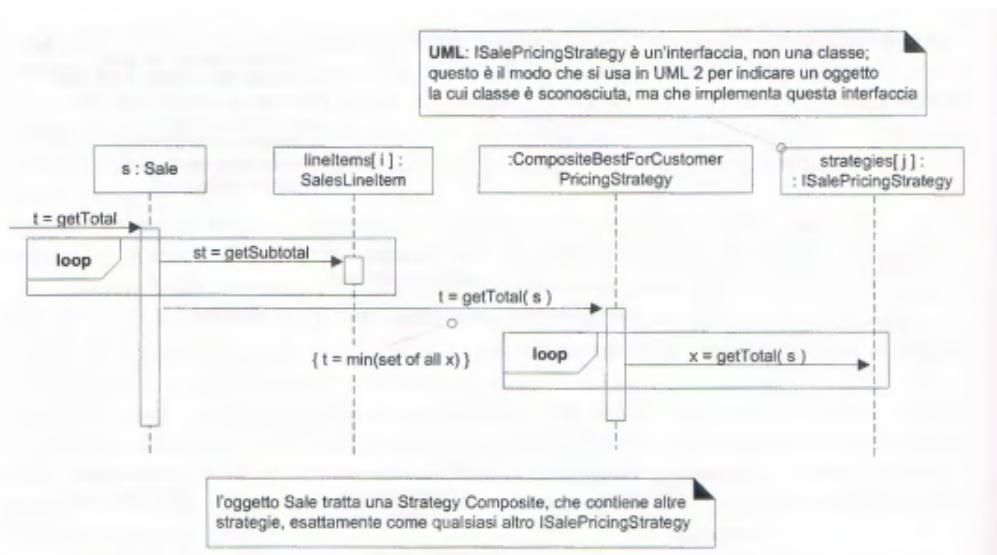
Passiamo ora a **Composite**:

Nome: **Composite**

Problema: Come trattare un gruppo o una struttura composta di oggetti nello stesso modo (polimorficamente) di un oggetto non composto (atomico)?

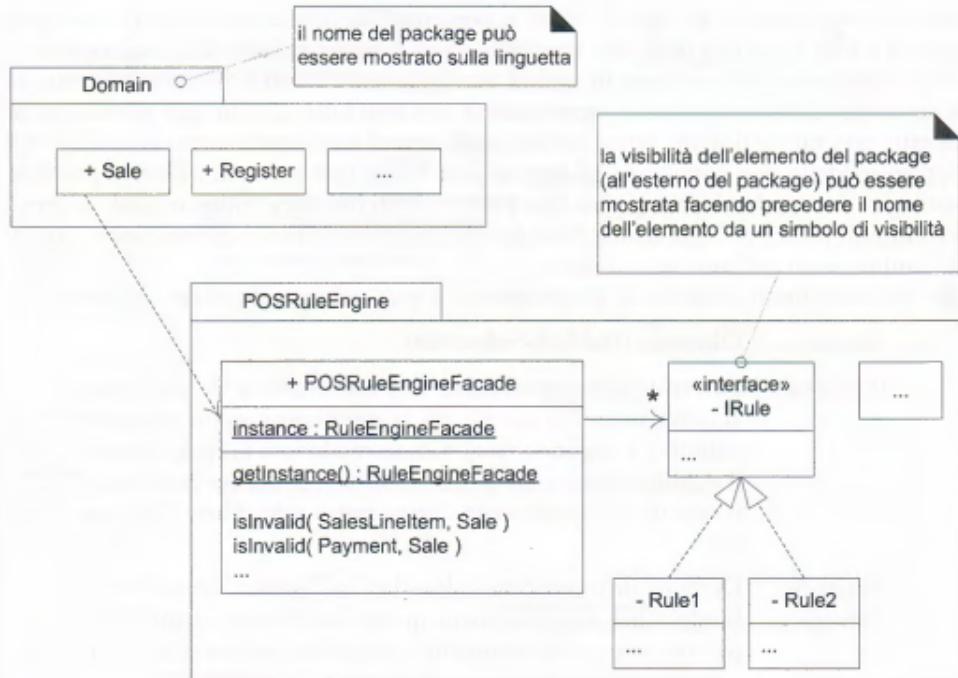
Soluzione: Definisci le classi per gli oggetti composti e atomici in modo (consiglio) che implementino la stessa interfaccia.





e vediamo anche **facade**.

Nome:	<b>Facade</b>
Problema:	È richiesta un'interfaccia comune e unificata per un insieme disparato di implementazioni o interfacce, come per definire un sottosistema. Può verificarsi un accoppiamento indesiderato a molti oggetti nel sottosistema, oppure l'implementazione del sottosistema può cambiare. Che cosa fare?
Soluzione: (consiglio)	Definisci un punto di contatto singolo con il sottosistema, ovvero un oggetto facade (facciata) che copre il sottosistema. Questo oggetto facade presenta un'interfaccia singola e unificata ed è responsabile della collaborazione con i componenti del sottosistema.

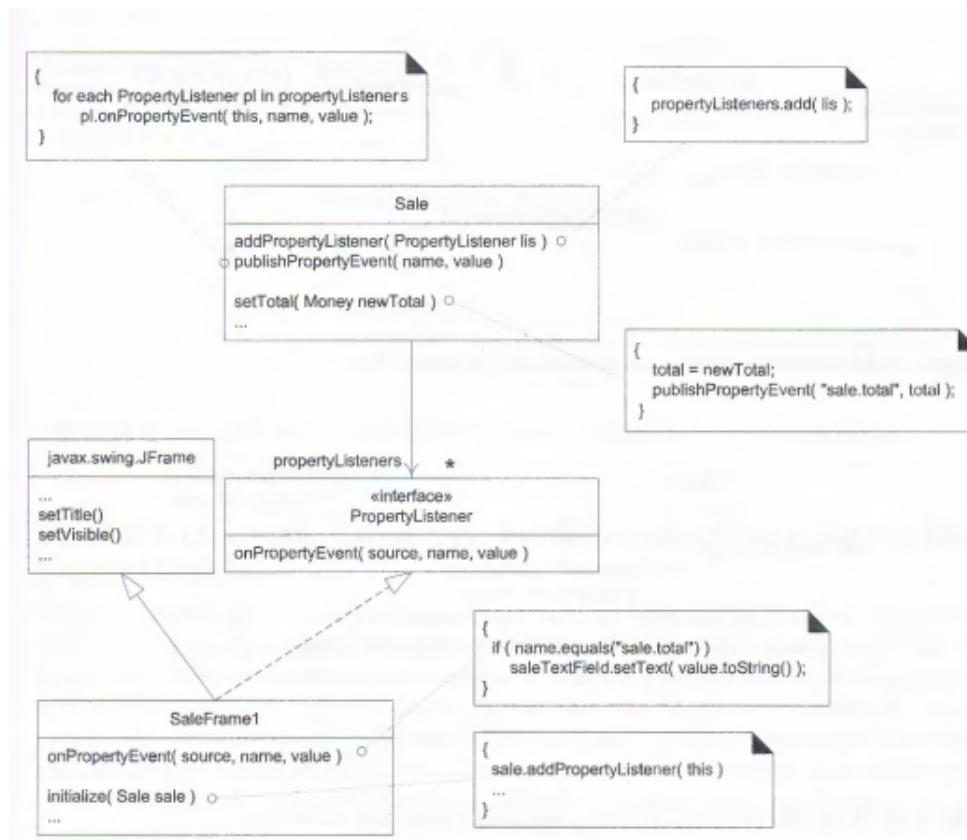


*Una Facade è un oggetto "front-end" che rappresenta il punto di entrata singolo ai servizi di un sottosistema; l'implementazione e gli altri componenti del sottosistema sono privati e non possono essere visti dai componenti esterni. Una Facade fornisce Protected Variations da cambiamenti nell'implementazione di un sottosistema.*

Un altro requisito per l'iterazione è quello di aggiungere la capacità per una finestra della GUI di aggiornare la visualizzazione del totale della vendita, quando il totale cambia. L'idea è di risolvere il problema per questo singolo caso, quindi nelle successive iterazioni estendere la soluzione all'aggiornamento della visualizzazione nella GUI anche per altri dati che cambiano. Il principio di Separazione Modello-Vista scoraggia questa soluzione. Esso afferma che gli oggetti "modello" non devono conoscere gli oggetti "vista" o presentazione (come una finestra). Esso promuove il Low Coupling dagli altri strati allo strato di presentazione (UI) degli oggetti. Una conseguenza del sostegno di questo accoppiamento basso è che esso consente la sostituzione dello strato vista o presentazione con uno nuovo, o un tipo particolare di finestre con nuove finestre, senza influire sugli oggetti non appartenenti alla UI. Se gli oggetti modello non conoscono gli oggetti Java Swing (per esempio), allora è possibile collegare un'interfaccia Swing o un tipo particolare di finestre e collegare qual cos'altro. Pertanto, il principio Separazione Modello-Vista supporta Protected Variations rispetto al cambiamento dell'interfaccia

utente. Per risolvere questo problema di progettazione, si può utilizzare il pattern Observer:

Nome:	<b>Observer (Publish-Subscribe)</b>
Problema:	Diversi tipi di oggetti subscriber (abbonato) sono interessati ai cambiamenti di stato o agli eventi di un oggetto publisher (editore), e vogliono reagire in un modo loro proprio quando il publisher genera un evento. Inoltre, il publisher vuole mantenere un accoppiamento basso verso i subscriber. Che cosa fare?
Soluzione: (consiglio)	Definisci un'interfaccia "subscriber" o "listener" (ascoltatore). I subscriber implementano questa interfaccia. Il publisher può registrare dinamicamente i subscriber che sono interessati a un evento e avvisarli quando l'evento si verifica.



## 12.1 Basi per la Progettazione di un Framework

Vediamo innanzitutto diversi meccanismi di memorizzazione, in quanto si studia un framework per la persistenza per **oggetti persistenti**, che sono

quelli che richiedono una memorizzazione persistente:

- **basi di dati a oggetti.** Se è utilizzata una base di dati a oggetti per memorizzare e recuperare gli oggetti, non sono necessari ulteriori servizi di persistenza personalizzati o di terze parti. Questo è uno degli aspetti che ne motivano l'utilizzo; tuttavia, sono relativamente rari.
- **basi di dati relazionali.** A causa della prevalenza degli RDB, il loro utilizzo è spesso richiesto, piuttosto che quello delle basi di dati a oggetti, più naturali nelle applicazioni orientate agli oggetti. In questo caso, sorgono una serie di problemi a causa del disaccoppiamento tra le rappresentazioni dei dati orientate ai record e quelle orientate agli oggetti; questi problemi sono esaminati più avanti. È necessario un servizio speciale di mapping tra oggetti e relazioni (O-R, Object-Relational)
- **altri.** Oltre agli RDB, è talvolta opportuno memorizzare gli oggetti in altri meccanismi o formati di memorizzazione, come file piatti, strutture XML, file PDB per Palm OS, basi di dati gerarchiche e così via. Come per le basi di dati relazionali, esiste un disaccoppiamento nella rappresentazione tra gli oggetti e questi formati non orientati agli oggetti, e di conseguenza anche in questo caso sono necessari servizi particolari per farli funzionare con gli oggetti.

*Un framework per la persistenza è un insieme estensibile di tipi, generico e riusabile, che fornisce funzionalità a supporto degli oggetti persistenti. Un servizio di persistenza (o sottosistema) fornisce effettivamente il servizio, e sarà creato con un framework per la persistenza. Un servizio di persistenza viene solitamente scritto per l'utilizzo con le basi di dati relazionali, e in questo caso è chiamato anche un servizio di mapping O-R. Normalmente un servizio di persistenza deve tradurre gli oggetti in record (o un'altra forma di dati strutturati, come XML) e salvarli in una base di dati, nonché tradurre i record in oggetti quando li recupera da una base di dati. Inoltre un framework è un insieme estensibile di oggetti per delle funzioni correlate.* La qualità distintiva di un framework è che essa fornisce un'implementazione per le funzioni fondamentali e invariabili, e include un meccanismo che consente a uno sviluppatore di inserire le funzioni variabili o di estendere le funzioni.

In generale, un framework:

- è un insieme coeso di interfacce e classi che collaborano per fornire servizi per la parte fondamentale e invariabile di un sottosistema logico
- contiene classi concrete e (soprattutto) classi astratte che definiscono le interfacce a cui conformarsi, le interazioni tra oggetti a cui partecipare, e altri invarianti

- di solito (ma non necessariamente) richiede all'utente del framework di definire sottoclassi di classi esistenti del framework per fare uso, personalizzare, ed estendere i servizi del framework
- ha classi astratte che possono contenere sia metodi concreti che astratti;
- si basa sul principio di Hollywood: "Non chiamateci, vi chiameremo noi". Ciò significa che le classi definite dall'utente (per esempio, le nuove sottoclassi) riceveranno dei messaggi dalle classi predefinite del framework. Esse sono di solito gestite implementando i metodi astratti della superclasse

Si hanno i seguenti concetti fondamentali:

- **mapping.** Deve esserci un mapping (ovvero, una corrispondenza) tra una classe e la propria memorizzazione persistente (per esempio, una tabella in una base di dati), e tra gli attributi di un oggetto e i campi (colonne) di un record. In sostanza, deve esserci un mapping tra schemi tra i due schemi
- **identità d'oggetto.** Per correlare facilmente i record agli oggetti, e per garantire che non ci siano duplicati inopportuni, i record e gli oggetti hanno un identificatore d'oggetto univoco
- **mapper della base di dati.** Un mapper della base di dati, una Pure Fabrication, è responsabile della materializzazione e dematerializzazione
- **Materializzazione e dematerializzazione.** La materializzazione è l'atto di trasformare in oggetti una rappresentazione di dati non a oggetti (per esempio, record) da una memorizzazione persistente. La dematerializzazione è l'attività opposta (nota anche come passivazione).
- **Cache.** I servizi di persistenza memorizzano gli oggetti materializzati in una cache, per motivi di prestazioni.
- **Stato transazionale degli oggetti.** È utile conoscere lo stato degli oggetti in termini della loro relazione con la transazione corrente. Per esempio, è utile conoscere quali oggetti sono stati modificati (sono dirty) in modo tale che sia possibile stabilire se devono essere salvati di nuovo nella memoria persistente.
- **Operazioni per le transazioni.** Operazioni commit e rollback

# Capitolo 13

## Refactoring e Code Smell

### 13.1 refactoring

Con **refactoring** ci si riferisce al processo consistente nel cambiare un sistema software in modo da non alterarne il comportamento ma da migliorare la struttura interna, leggendolo più comprensibile e semplice da modificare.  
Si hanno diversi passi per fare il refactor:

- partire da un codice funzionante
- trovare le parti di codice imperfette
- capire come semplificare il codice
- semplificare il codice e testarlo
- ripetere finché non si è sistemato tutto il codice

e in generale le motivazioni per fare il refactor sono:

- migliorare il design del software
- ridurre la quantità di codice necessaria per completare un task
- rendere il codice più facile da comprendere
- aiutare a trovare bug, avendo un codice più pulito

si effettua il refactor quando:

- quando si aggiunge una funzione
- quando si deve fixare un bug

- quando si deve effettuare code review
- quando si ha palesemente del codice "brutto"

e non si effettua quando:

- il codice è così mal messo da dover essere rescritto
- se il codice non funziona
- se si è vicini alla deadline del progetto
- se non si ha una suit di test

e si hanno i seguenti problemi:

- in caso di database bisogna effettuare la migrazione dei dati
- si hanno problemi per la ownership del codice
- le interfacce non vanno cambiate e questo potrebbe generare problemi

vediamo i metodi principali:

- **extract method** per il quale si trasforma un metodo lungo in uno più breve, estraendone una parte in un metodo di supporto, per esempio:

```
public class Player
{
    private Piece piece;
    private Board board;
    private Die[] dice;
    // ...

    public void takeTurn()
    {
        // roll dice
        int rollTotal = 0;
        for (int i = 0; i < dice.length; i++)
        {
            dice[i].roll();
            rollTotal += dice[i].getFaceValue();
        }

        Square newLoc =
            board.getSquare(piece.getLocation(),
                            rollTotal);
        piece.setLocation(newLoc);
    }

} // end of class
```

```
public class Player
{
    private Piece piece;
    private Board board;
    private Die[] dice;
    // ...
    public void takeTurn()
    {
        // the refactored helper method
        int rollTotal = rollDice();

        Square newLoc =
            board.getSquare(piece.getLocation(),
                            rollTotal);
        piece.setLocation(newLoc);
    }

    private int rollDice()
    {
        int rollTotal = 0;
        for (int i = 0; i < dice.length; i++)
        {
            dice[i].roll();
            rollTotal+=dice[i].getFaceValue();
        }
        return rollTotal;
    }
} // end of class
```

```
// good method name, but the logic of the body is not clear
boolean isLeapYear( int year )
{
    return ( ( ( year % 400 ) == 0 ) ||
             ( ( ( year % 4 ) == 0 ) && ( ( year % 100 ) != 0 ) ) );
}



```

- **extract class** per il quale si crea una nuova classe e vi muove alcuni campi e metodi di un'altra classe
- **move method** per il quale si crea un nuovo metodo, con un corpo simile, nella classe che lo usa di più

## 13.2 Code Smell

Se si ha un design a oggetti ci si può concentrare sul migliorarne il design, sistemando appunto situazioni "brutte" nel codice come:

- codice duplicato, *code clone*
- metodi (più di 20 righe) o classi troppo estesi (problemi di astrazione e di code clone), si risolve con l'extract method e con l'extract class o l'extract subclass (che crea una sottoclass per ogni feature)
- lunghe liste di parametri
- feature envy, ovvero una classe usa tanti metodi da altre, rovinando l'astrazione etc... Si risolve con il move method
- switch statement che può creare problemi di code clone risolubili creando sottoclassi e usando l'extract method

- data class, ovvero una classe con solo variabili di classe, getter, setter metodi e proprietà. Si ha quindi il problema di feature envy e che si ha un design estremamente procedurale. Si usa il move method refactoring per spostare i metodi
- long parameters list, ovvero troppi metodi passati in un metodo. Si risolve cambiando parametri con metodi o introducendo oggetti come parametri
- shotgun surgery, ovvero quando cambiando una classe si deve cambiare anche un gran numero di altre classi. Si risolve con il move method e il move field
- ...

Infine i commenti devono essere pochi e brevi, con codice autoesplicativo.

# **Capitolo 14**

## **Testing**