

# SAS

venerdì 6 marzo 2020 13:27

Questi appunti sono condivisi da Stefano Vittorio Porta (Stefa168) con licenza  
<https://creativecommons.org/licenses/by-nc-nd/4.0/>.

# Sviluppo Incrementale

venerdì 6 marzo 2020 13:27

Migliore rispetto allo sviluppo a cascata in quanto ci permette di avere dei tempi limite per sviluppare in dei "workshop" un piccolo set di caratteristiche progettate nei dettagli riferendoci ai casi d'uso.

In questo modo è possibile avere feedback durante lo sviluppo che, se negativi, ci possono aiutare a correggere i problemi riscontrati.

Esempio di sviluppo iterativo:

1. Prima della prima iterazione, si tiene un workshop dei requisiti timeboxed, per esempio di due giorni esatti.

Si lavora assieme ai responsabili dell'organizzazione e dello sviluppo.

Giorno 1: Si analizzano i requisiti ad alto livello, identificando solo i nomi dei casi d'uso e le caratteristiche, oltre ai requisiti non funzionali più importanti.

Si sceglie un piccolo set dei casi d'uso, dando priorità:

- 1) alla significatività dal punto di vista dell'architettura
- 2) all'elevato valore di business
- 3) ai casi d'uso a rischio elevato

Tempo rimanente: si analizzano intensamente e nel dettaglio i requisiti funzionali e non funzionali per i casi d'uso identificati. Al termine, solo il 10% dei requisiti totali è stato analizzato in profondità e il rimanente 90% solo ad alto livello.

2. Prima della prima iterazione, si sceglie un sottoinsieme dei requisiti dei casi d'uso identificati, su cui fare progettazione, implementazione e test in un tempo specificato.

Si esegue la prima iterazione in un tempo scelto per la timebox (ad esempio 3 settimane).

3. I primi due giorni si fa modellazione e progettazione a coppie, abbozzando diagrammi UML.

Gli sviluppatori, successivamente, passano alla programmazione. Programmano le parti progettate e fanno test, oltre che integrazione in modo continuo, utilizzando i modelli abbozzati come punto di partenza e di ispirazione.

Una settimana prima della fine, si analizza assieme al team se gli obiettivi originari possono essere raggiunti nell'iterazione; altrimenti si rimodula la portata dell'iterazione rimettendo gli obiettivi non raggiunti nell'elenco delle cose da fare per le iterazioni successive.

Negli ultimi giorni della timebox, il codice è "congelato": deve essere interamente caricato, integrato e testato per creare l'iterazione.

Si effettua una dimostrazione del sistema parziale, nelle parti interessate in modo da rendere visibili i progressi. Si riceve un feedback.

4. Si esegue il secondo workshop sui requisiti verso la fine della prima iterazione, durante il quale si decidono i casi d'uso da utilizzare per la seconda iterazione.
5. Si pianifica la seconda iterazione nei giorni successivi

6. Si esegue l'iterazione 2 con gli stessi passi
7. Si continua a ripetere tutto per quattro iterazioni e cinque workshop dei requisiti, in modo che alla fine della quarta iterazione probabilmente l'80-90% dei requisiti sia stato scritto in modo dettagliato; solo il 10% del sistema invece è stato implementato.

## Pianificazione iterativa/adattiva

In ogni iterazione si stabilisce il piano di lavoro per una sola iterazione. Si può fare:

- **UP**: alla fine di ciascuna iterazione per l'iterazione successiva
- **Scrum**: all'inizio di ciascuna iterazione per stabilire il piano dell'iterazione corrente.

**Gli obiettivi dell'iterazione in corso non vengono cambiati.** Ciò è accettabile perché le iterazioni sono brevi e il feedback è frequente. Il team deve lavorare al suo meglio, concentrandosi sul lavoro stabilito per l'iterazione.

La pianificazione è guidata dal **rischio** (gli elementi più importanti e rilevanti) e dal **cliente** (in modo da concentrarsi sulle richieste a cui il cliente tiene di più).

## Unified Process

Processo iterativo ed evolutivo/incrementale.

Le iterazioni iniziali sono guidate dal **rischio**, dal **cliente** e dall'**architettura**.

UP è flessibile e può essere applicato con Extreme Programming e Scrum.

Non si tratta di uno standard ma usa UML. La versione commerciale è *Rational Unified Process*.

UP incoraggia all'uso di pratiche agili di altre metodologie:

- Timeboxing e iterazioni corte
- Raffinamento di piani, requisiti e progettazione
- Gruppi di lavoro auto organizzati e riunioni regolari (Scrum)
- Programmazione a coppie e sviluppo guidato dai test (XP)
- Comprensione del software > documentazione dello stesso

Agile: risposta rapida e flessibile di iterazione in iterazione. Si usa UML per la "scoperta", in modo da agevolare la comunicazione

In UP c'è:

- Un'organizzazione del piano del progetto in fasi sequenziali
- Indicazioni sulle attività da svolgere nell'ambito di discipline e sulle loro inter-relazioni
- Ruoli predefiniti
- Un insieme di artefatti (documenti) da produrre

Ruoli => Documenti => Artefatti

Ci sono 4 fasi:

1. **Ideazione**: fase iniziale. Ha una visione approssimativa dell'intero progetto (studio economico, costi, tempi) e degli obiettivi del progetto. Alla fine si dovrebbe avere una idea di questi elementi.
2. **Elaborazione**: visione raffinata, implementazione iterativa del nucleo dell'architettura, risoluzione dei rischi maggiori, identificazione dei requisiti e della portata, stime più realistiche sulle loro inter-relazioni
3. **Costruzione**: implementazione iterativa degli elementi rimanenti, più facili e a minor rischio (siamo verso la fine del progetto).
4. **Transizione**: beta testing e rilascio

Quando *alla fine di una iterazione* si verifica una decisione o una valutazione significativa che porta alla terminazione di una fase, si raggiunge una **milestone**.

L'ideazione non è solo una fase di requisiti, ma si decide se proseguire con il progetto o interromperlo.

Durante la fase di **Elaborazione** si implementa in modo iterativo l'architettura del sistema e vengono mitigati (affrontati e se ci sono problemi, li si risolvono) i rischi maggiori.

Ogni iterazione ha l'applicazione di varie discipline (insieme di attività e relativi elaborati in una determinata area; gli elaborati/artefatti/work product sono i termini utilizzati per identificare un qualsiasi prodotto del lavoro: codice, schemi di DB, documenti di vario genere, ...):

- **Modellazione** del business: si modella il dominio del problema e il suo ambito
- **Requisiti**: cosa ci aspettiamo che faccia, quali sono le sue funzionalità
- **Progettazione**: attività di analisi dei requisiti e progetto del sistema
- **Implementazione**: attività di progettazione dettagliata e codifica del sistema. Si effettua testing sui componenti
- **Test**: attività di controllo della qualità, test di integrazione e di sistema
- **Rilascio**: attività di consegna e messa in opera.

In una qualsiasi iterazione, in genere, vengono svolte tutte le discipline.

In proporzione diversa e in funzione delle fasi, si applicano anche delle discipline di supporto:

- **Gestione delle configurazioni e del cambiamento**: manutenzione del progetto
- **Gestione del progetto**: attività di pianificazione e gestione del progetto
- **Infrastruttura**: attività che supportano il team di progetto, riguardo i processi e gli strumenti utilizzati.

Le iterazioni iniziali tendono in modo naturale a dare una maggiore enfasi relativa sui requisiti e sulla progettazione, mentre quelle successive lo faranno in misura minore.

Durante l'elaborazione le iterazioni tendono ad avere un livello relativamente alto di lavoro sui requisiti e la progettazione, sebbene prevedano anche un certo livello di implementazione. Durante la costruzione invece l'enfasi è sulla implementazione.

In UP, UML è usato solo come linguaggio di modellazione; i diagrammi si usano con variabilità, seguendo le caratteristiche di iterazione ed incremento. Quindi quello che accadrà è che il diagramma sarà lo stesso, ma verrà incrementato e aggiornato, ricevendo le nuove componenti dell'iterazione raggiunta.

La scelta delle pratiche e degli artefatti UP per un progetto si indicano in un documento chiamato "scenario di sviluppo" (che a sua volta è un artefatto della disciplina **Infrastruttura**).

# Requisiti evolutivi

venerdì 6 marzo 2020 15:47

**Requisito:** capacità o condizione a cui il sistema e il progetto devono essere conformi.

Le **sorgenti dei requisiti** sono le richieste degli utenti del sistema, per risolvere dei problemi e raggiungere degli obiettivi degli utenti.

Tipi di requisiti:

- **Funzionali:** offrono capacità e descrivono il comportamento del sistema, in termini di funzionalità fornite agli utenti.
- **Non funzionali:** Le proprietà del sistema nel suo complesso. Ad esempio *sicurezza, prestazioni, scalabilità, usabilità, ...*

UP promuove un insieme di best practice. Comprende l'importanza di accettare il fatto che i desideri delle parti interessate possono cambiare. Offrono un approccio sistematico per trovare, documentare, organizzare e tracciare i requisiti che cambiano (durante lo sviluppo) di un sistema.

In UP si iniziano programmazione e test quando è stato specificato solo il 10-20% dei requisiti più significativi dal punto di vista del business, del rischio e dell'architettura.

L'acquisizione dei requisiti deve essere effettuata attraverso tecniche quali:

- Scrivere i casi d'uso coi clienti
- Workshop dei requisiti a cui partecipano sia sviluppatori che clienti
- Gruppi di lavoro con dei rappresentanti dei clienti
- Dimostrazione ai clienti dei risultati di ciascuna iterazione, per ottenere più facilmente feedback

## Modello FURPS+

- **Funzionale:** requisiti funzionali e di sicurezza
- **Usabilità:** facilità d'uso del sistema, documentazione e aiuto per l'utente
- **Affidabilità (Readability):** la disponibilità del sistema, la capacità di tollerare guasti o di essere ripristinato in seguito a fallimenti
- **Prestazioni:** tempi di risposta, throughput, capacità e uso delle risorse
- **Sostenibilità:** facilità di modifica per riparazioni e miglioramenti, adattabilità, manutenibilità, verificabilità, localizzazione, configurazione e compatibilità
- Altre (+): vincoli di progetto (risorse, hardware, ...), interoperabilità, operazionali, fisici, legali, ...

## Elaborati di UP

UP ha diversi elaborati (molti sono opzionali) in cui vengono codificati i vari requisiti:

- **Modello dei casi d'uso:** scenari tipici dell'uso di un sistema (requisiti funzionali e di comportamento)
- **Specifiche Supplementari:** ciò che non rientra nei casi d'uso, come requisiti non funzionali o funzionali *non esprimibili attraverso casi d'uso* (ad esempio la generazione di un report mensile automatica)
- **Glossario:** termini significativi, dizionario dei dati, requisiti relativi ai dati, regole di validazione, valori accettabili. Se si sviluppa una applicazione per qualcuno, potrebbero non comprendersi nemmeno i termini del cliente. Il primo lavoro quindi è fare un glossario dei termini, individuando i sinonimi e gli attori.
- **Visione:** riassume i requisiti ad alto livello. È un documento sintetico che permette di apprendere rapidamente le idee principali del progetto. È utile per i nuovi programmati che entrano in un progetto in corso per avere una visione complessiva.
- **Regole di business:** le regole del dominio, i requisiti o le politiche che trascendono un unico progetto software e a cui questo deve conformarsi (ad esempio leggi statali)

# Fase 1: Ideazione

venerdì 6 marzo 2020 15:45

È uno studio di fattibilità.

- Si analizzano il 10% dei casi d'uso in dettaglio.
- Si analizzano i requisiti non funzionali più critici
- Si realizza uno studio economico per stabilire l'ordine di grandezza del progetto e la stima dei costi
- Si prepara l'ambiente di sviluppo

La **durata** è breve.

Lo scopo non è quello di definire tutti i requisiti (ma solo quelli più critici), né di generare una stima o un piano di progetto affidabili (che avvengono durante la fase di elaborazione)

Durante l'ideazione le due parti decidono se il progetto merita un'indagine più seria durante la fase di elaborazione.

La visione, le specifiche supplementari, il glossario e le regole di business servono appositamente. Lo sviluppo completo può avvenire solo durante l'Elaborazione.

Durante l'Ideazione i documenti sono abbozzati e definiti in maniera "leggera"

Durante l'Elaborazione i documenti vengono raffinati sulla base del feedback proveniente dalla costruzione incrementale delle parti del sistema, dall'adattamento e dai vari workshop sui requisiti tenuti durante le iterazioni di sviluppo.

Il "congelamento e firma per accettazione" è la stipulazione di un accordo tra le parti interessate su ciò che verrà fatto nel resto del progetto e porta all'assunzione di impegni (contrattuali) sui requisiti e sui tempi. **Ciò avviene alla fine dell'Elaborazione. I requisiti a maggior rischio sono analizzati per primi prima della definizione del contratto.**

Durante la Costruzione, i requisiti principali, funzionali o meno, dovrebbero essere stabilizzati.

La scelta delle pratiche e degli elaborati UP per un progetto può essere scritta in un documento chiamato **Scenario di Sviluppo**

## Artefatti

- Visione e studio economico
- Modello dei Casi d'Uso
- Specifiche supplementari
- Glossario
- Lista dei Rischi e Piano di Gestione dei rischi
- Prototipi e proof of concept
- Piano dell'iterazione
- Piano delle Fasi e piano di Sviluppo del Software
- Scenario di Sviluppo

Bisogna scegliere gli artefatti che aggiungono valore al progetto, quelli che servono per capire e per raggiungere il successo nello studio di fattibilità.

Gli artefatti vengono completati parzialmente; sono preliminari e possono solo approssimare le necessità che verranno identificate più tardi.

Il valore della modellazione è quello di migliorare la comprensione, e non quello di documentare

delle specifiche affidabili (che cambierebbero data la natura di UP).

Le **specifiche supplementari** raccolgono altri requisiti, informazioni e vincoli che non sono facilmente colti dai casi d'uso o nel glossario, compresi attributi di qualità e i requisiti "URPS+" a livello di intero sistema.

Un esempio sono la cronologia delle versioni. (vedi capitolo 8)

### Documento di Visione

Riassume alcune delle informazioni contenute del modello dei casi d'uso e nelle specifiche supplementari.

Describe il progetto come contesto per i partecipanti, col fine di stabilire una visione comune del progetto.

Contiene:

- Obiettivi e problemi fondamentali ad alto livello delle parti interessate (utile soprattutto per i requisiti non funzionali)
- Riepilogo delle caratteristiche di sistema (raggruppamenti ed astrazioni)

### Glossario

Elenco dei termini significativi e delle relative definizioni. Include pseudonimi e termini composti. È importante eliminare le discrepanze per ridurre i problemi di comunicazione e di ambiguità dei requisiti.

È spesso utile iniziare da questo documento.

Il glossario spesso svolge anche il ruolo di dizionario dei dati, un documento dei dati relativi ad altri dati (metadati).

### Regole di dominio (o regole di business)

Stabiliscono come può funzionare un dominio o un business.

# Casi D'uso

lunedì 9 marzo 2020 10:27

La disciplina dei requisiti in ogni iterazione viene svolta. Il maggior utilizzo sarà sicuramente nelle fasi di Ideazione ed Elaborazione, ma anche dopo sarà ancora usata.

**Disciplina dei Requisiti:** Il processo per scoprire cosa deve essere costruito ed orientare lo sviluppo verso il sistema corretto. È una fase di scoperte ed indagine volte a capire cosa deve essere costruito. Questa comprensione deve orientare lo sviluppo del sistema.

Nelle metodologie iterative ed evolutive, la disciplina dei requisiti non è una fase che viene completata prima di andare a realizzare, ma è distribuita all'interno di ogni iterazione del processo. Il feedback va anche ad aiutare il processo di scoperta e comprensione della costruzione dell'iterazione successiva.

**Requisiti di sistema:** le capacità e le condizioni alle quali il sistema deve essere conforme, scritti nel "linguaggio" del committente.

## Flusso delle attività in UP

Passi principali (non per forza eseguiti separatamente):

- Produrre una **lista dei requisiti potenziali** (candidati)
- Capire il **contesto del sistema**
- Catturare i **requisiti funzionali** (di comportamento)
- Catturare i **requisiti non funzionali**

### **Lista dei requisiti potenziali (feature list)**

Dobbiamo identificare dei possibili casi d'uso e si analizzano quelli più rischiosi e rilevanti. Per capire quali sono, si costruisce la lista dei requisiti potenziali.

Sono caratterizzati da:

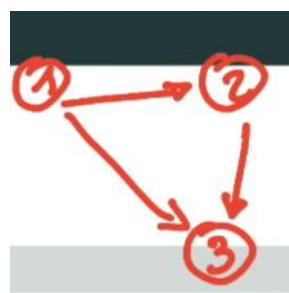
- Una breve **descrizione**
- **Stato** (proposto, approvato, incorporato, validato)
- **Costi** di implementazione stimati
- **Priorità**
- **Rischio** associato per la sua implementazione

La lista dei requisiti è usata anche per stimare la taglia del progetto e per decidere come suddividere il progetto in sequenze di iterazioni. Idealmente la lista dei requisiti potenziali è stilata durante **l'Ideazione**.

### **Capire il contesto del Sistema (MDPA)**

Per poterlo fare, esistono due approcci:

- **Modellazione di dominio:** Si descrivono i concetti importanti del sistema come oggetti di dominio relazionando i concetti con associazioni.  
(Si descrivono tutti i domini/stati che sono accettabili)
- **Modellazione di business:**  
(Spiega quali transizioni tra i domini/stati sono accettabili)
  - È un super-insieme del modello di dominio; descrive i processi di business (le collezioni di attività correlate e strutturate che producono uno specifico servizio o prodotto, compiono uno specifico scopo o sono destinate a un cliente specifico)
  - È un prodotto dell'ingegneria del business
  - Ha lo scopo di migliorare i processi di business



### **Catturare i requisiti funzionali**

Si catturano in UP tramite i **casi d'uso**, che rappresentano una maniera di utilizzare il sistema da parte di un utente per raggiungere un goal. Si tratta di **descrizioni testuali**.

## Catturare i requisiti non funzionali

I requisiti **non** funzionali (cioè tutto quello che non è requisito funzionale) possono essere inclusi nei casi d'uso se sono relazionati a dei requisiti funzionali descritti da quel casi d'uso.

(Esempio: Il pagamento deve avvenire tramite questo strumento. Il completamento del pagamento deve sempre avvenire entro 20 secondi.)

Il caso d'uso è il pagamento tramite strumento, mentre il requisito non funzionale che è descritto nello stesso momento, è il tempo del pagamento)

Alternativamente, vengono sempre descritti nelle **Specifiche Supplementari**

## Approccio UP-Agile

- In questo approccio, i requisiti funzionali sono catturati con i casi d'uso (UC). Se ci sono requisiti non funzionali relazionati, questi vengono inclusi nel caso d'uso stesso.
- I requisiti non funzionali **generali** sono inclusi nel documento di **Specifiche Supplementari** (SS).
- Il contesto del sistema è catturato dai **diagrammi UML dei casi d'uso**.
- UC e SS costituiscono l'input per definire il **modello di dominio**, che quindi parte avendo dei casi d'uso e specifiche.

(Non considereremo il modello di business.)

## Casi d'uso: come trovarli

Unified Process è una metodologia "**use-case driven**". Questo significa che è costruito e guidato dai casi d'uso che hanno un ruolo essenziale.

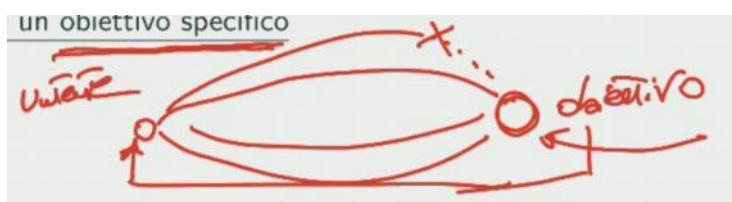
I requisiti funzionali si descrivono con casi d'uso che si usano per pianificare a loro volta le iterazioni.

L'**analisi** e la **progettazione** (OOA e OOP), i **test**, la redazione dei **manuali utente** e la **definizione** del progetto sono basati sui casi d'uso.

I casi d'uso sono una descrizione **testuale** di scenari d'uso interessanti del sistema software che si deve realizzare. È necessario scriverli bene.

Gli elementi coinvolti sono:

- **Attori**: qualcosa (un altro sistema) o qualcuno, dotato di un comportamento
- **Scenario** (o istanza di caso d'uso): una sequenza specifica di azioni ed interazioni tra il sistema ed alcuni attori. Uno scenario descrive una particolare storia nell'uso del sistema, un percorso attraverso il caso d'uso.
- **Caso d'uso** (o casi di utilizzo): una collezione di scenari correlati che descrivono un attore che usa il sistema per **raggiungere un obiettivo specifico (riuscendo o fallendo!)** Ci possono essere multiple strade per compiere un goal e altre che porteranno al non raggiungimento di esso. È identificato dall'obiettivo che viene raggiunto.



Esempio:

### Gestisci Restituzione (Handle Returns)

**Scenario principale di successo:** Un cliente arriva alla cassa con alcuni articoli da restituire. Il **cassiere** utilizza il sistema POS per registrare articolo restituito..

#### Scenari alternativi:

Se il cliente aveva pagato con carta di credito, e l'operazione di rimborso sulla relativa carta di credito è stata respinta, allora il cliente viene informato e viene rimborsato in contanti.

Se il codice identificativo dell'articolo non viene trovato nel sistema, il sistema

avvisa il cassiere e suggerisce l'inserimento manuale del codice (può darsi che sia danneggiato).

Se il sistema rileva un fallimento nella comunicazione con il sistema esterno di gestione della contabilità...

I casi d'uso in UP sono documenti di **testo**, non sono diagrammi! La modellazione dei casi d'uso è innanzitutto un atto di scrittura di testi e non di diagrammi.

**Il modello dei casi d'uso** poi riassumerà insieme **tutti i casi d'uso descritti** (usando anche UML) e fungerà da modello di contesto del sistema e da indice dei nomi.

I casi d'uso è bene notare che non sono una pratica di analisi di progettazione classica. Sono utili a rappresentare requisiti che sono in input all'analisi e alla progettazione. Analisi e progettazione sono orientati agli oggetti, i casi d'uso no: non fanno minimamente riferimento a soluzioni.

I casi d'uso sono descrizioni testuali che devono essere scritte bene. **Sono l'input dell'analisi e della progettazione, che viene dopo.**

I casi d'uso definiscono i contratti, che indicano il comportamento del sistema.

Bisogna porre attenzione sull'utente e sul suo punto di vista.

- Chi usa il sistema?
- Quali sono i loro scenari d'uso tipici?
- Quali sono i loro obiettivi?

Sono il meccanismo centrale per la scoperta e la **definizione** dei requisiti (funzionali)

Attenzione: NO "Il sistema dovrà fare..."

### Attore

Un attore è qualcosa o qualcuno dotato di **comportamento**.

(Attenzione: il sistema stesso è considerato un attore!)

Gli attori sono ruoli svolti da persone, organizzazioni, software, macchine.

Si dividono in:

- **Attore primario:**
  - Raggiunge gli obiettivi utente utilizzando i servizi del sistema.
  - Utile per trovare gli obiettivi utente
- **Di supporto:**
  - Offre un servizio al sistema (*Esempio POS: servizio di validazione delle carte di credito*)
  - Utile per chiarire le interfacce esterne e i protocolli
- **Fuori scena:**
  - Ha un interesse nel comportamento del caso d'uso (*Esempio: il responsabile/direttore del supermercato che ha l'interesse che tutto funzioni correttamente coi POS*)
  - È utile per garantire che tutti gli interessi necessari vengano soddisfatti

### Formati del caso d'uso

- **Formato breve:**
  - Riepilogo conciso di un solo paragrafo, relativo (in genere) al solo scenario principale di successo. Non si analizzano diramazioni o fallimenti. È lo scenario più diretto per raggiungere lo scopo.
  - Serve a capire rapidamente l'argomento e la portata
- **Formato informale:**
  - Più paragrafi, scritti in modo informale, relativi a vari scenari.
  - Ha la stessa funzione del formato breve, ma ha un maggiore dettaglio
- **Formato dettagliato:**
  - Tutti i passi e le variazioni sono scritti in dettaglio

- Include pre-condizioni (perché il caso d'uso possa avvenire) e garanzie di successo (condizioni per cui sia avvenuto con successo il caso d'uso)
- Si scrivono a partire dal formato breve o informale

Si ricorda che durante l'ideazione si scrivono circa il 10% dei casi d'uso tra più critici in formato dettagliato utilizzando template appositi (casi d'uso dettagliato e strutturato)

Esempio di UC **breve**:

#### **Elabora vendita**

Un cliente arriva alla cassa con gli articoli da comprare. Il cassiere usa POS NextGen per registrare gli articoli. Il sistema presenta il totale e la lista dettagliata degli articoli. Il cliente inserisce le informazioni per il pagamento, che il sistema valida e registra. Il sistema aggiorna l'inventario. Il cliente ottiene la ricevuta dal sistema e se ne va con gli articoli comprati.

Esempio di UC **informale**:

#### **Gestire restituzioni**

- **Scenario Principale:** Un cliente arriva alla cassa con gli articoli da restituire. Il cassiere usa POS NextGen per registrare ciascun articolo restituito...
- Scenari Alternativi:
  - Se il cliente aveva pagato con carta di credito, e la transazione di rimborso è rifiutata, informare il cliente e pagarlo in contanti
  - Se l'id dell'articolo non è trovato nel sistema, notificare al cassiere e suggerire l'inserimento manuale del codice id
  - Se il sistema rileva guasti di comunicazione con sistemi esterni di contabilità...

[Template del caso d'uso dettagliato](#)

Sezione del caso d'uso	Commento
<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>	Usa direttamente il sistema; gli chiede di fornirgli i suoi servizi per raggiungere un obiettivo.
<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 – e vale la pena di dire al lettore.
<b>Garanzia di successo</b>	Che cosa deve essere vero se il caso d'uso viene completato con successo – e vale la pena di dire al lettore.
<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 varianti 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, come per esempio i problemi aperti.

Vedere caso d'uso dettagliato "Elabora Vendita" (Process Sale), a pagina 77-81 di C. Larman  
Può servire vedere anche "Il sistema POS NextGen" a pagina 50

# Scrittura di un caso d'uso

sabato 14 marzo 2020 11:21

## Preambolo

Il caso d'uso ha un **Preambolo** che precede lo scenario principale e le varie estensioni.

Il preambolo include:

- **Portata:** descrive i confini del sistema in progettazione
  - **Livello:** di solito livello di obiettivo utente o sottofunzione (un caso d'uso potrebbe essere come una procedura riutilizzabile in più punti; non è molto ricorrente l'uso).
  - **Attore finale, attore primario:**
    - L'attore finale è l'attore che vuole raggiungere un obiettivo e questo richiede l'esecuzione dei servizi del sistema
    - L'attore primario è l'attore che usa direttamente il sistema.
- Spesso i due attori coincidono.
- **Parti interessate:** indica chi ha interessi nel raggiungimento dell'obiettivo espresso dal caso d'uso in oggetto
  - **Pre-condizioni:** che devono essere vere prima di iniziare uno scenario del caso d'uso; **non** vengono verificate all'interno del caso d'uso
  - **Garanzie di successo** (post-condizioni): che cosa deve essere vero quando è stato completato con successo il caso d'uso

## Caso d'uso UC1: Elabora Vendita (Process Sale)

---

**Portata:** Applicazione POS NextGen

**Livello:** Obiettivo utente

**Attore primario:** Cassiere

**Attore finale:** Cliente

**Parti interessate e Interessi**

- Cliente (Customer, in alcune figure): 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.
- Cassiere (Cashier): Vuole un inserimento dei dati preciso e rapido. Non vuole errori nei pagamenti, perché gli ammanchi di cassa vengono detratti dal suo stipendio.
- 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.
- Addetto alle vendite: Vuole che le commissioni sulle vendite siano aggiornate.
- Direttore: Vuole essere in grado di eseguire rapidamente operazioni di sovrascrittura, e risolvere in modo semplice i problemi del Cassiere.
- Enti Governativi Fiscali: Vogliono riscuotere le imposte su ciascuna vendita. Possono essere più enti: nazionale, regionale e provinciale.
- Servizio di Autorizzazione di Pagamento (Payment Authorization Service): 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 registrata. Le imposte sono calcolate correttamente. La contabilità e l'inventario sono aggiornati. Le commissioni sono registrate. Le approvazioni alle autorizzazioni di pagamento sono registrate. Viene generata una ricevuta.

## Scenario principale di successo

Lo **scenario principale di successo** viene anche chiamato "percorso felice", "flusso di base" o "flusso tipico". In qualche modo descrive la sequenza di passi che porta a raggiungere l'obiettivo. Non è da descrivere in maniera algoritmica, ma come una sequenza di passi che può anche ripetersi più volte, ma che di solito *non comprende condizioni o diramazioni*.

La gestione del comportamento condizionale e delle alternative viene solitamente descritta nella (successiva) sezione delle Estensioni.

I passi possono essere di tre tipi:

- **Un'interazione** tra attori (anche il sistema si ricorda essere un attore)
  - Un attore interagisce con il sistema inserendo dati o effettuando una richiesta
  - Il sistema interagisce con un attore comunicandogli dati o una risposta
  - Il sistema interagisce con altri sistemi
- **Un cambiamento di stato** da parte del sistema
- **Una validazione** (normalmente fatta dal sistema)

Il primo passo a volte può essere un po' speciale: a volte è l'evento trigger che scatena l'esecuzione dello scenario (e a volte non è classificabile nei tre tipi di passi indicati sopra).

#### **Scenario principale di successo (o Flusso di base):**

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 di un 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.

#### **Scenario principale di successo**

#	Attore	Sistema
1	Decide di creare un nuovo menu	
2	Specifica un titolo per il menù.	Mostra i dettagli (titolo) del menù creato
3	Definisce una sezione del menù <u>assegnandole un nome.</u>	Mostra la sezione con il suo nome
4	Inserisce una voce nel menù associandola ad una ricetta del ricettario. La voce può avere un testo suo o corrispondere al nome della ricetta.	Registra la nuova voce di menu e mostra la sezione aggiornata
	<i>Ripete il passo 4 finché non ha completato la sezione.</i>	
	<i>Se vuole lavorare su un'altra sezione torna al passo 3.</i>	
5	Indica che il menù è a suo avviso completo e quindi utilizzabile.	Segnala che il menù è ora completo.
6	Conclude il lavoro su questo menù.	

Dal progetto di laboratorio 2017/2018.

#### **Estensioni**

Le estensioni hanno lo scopo di descrivere tutti gli altri scenari oltre a quello principale, sia di successo sia di fallimento (modi alternativi di fare la stessa cosa o di fallire).

Solitamente le estensioni sono descritte per differenza dallo scenario principale: non sono ridecritti tutti i punti che condiviscono, ma il punto in cui differisce e il punto in cui si ri-innesta nello scenario principale.

Gli scenari relativi alle estensioni sono diramazioni dello scenario principale di successo; vengono indicati con riferimento ai suoi passi.

Un'estensione è costituita da due parti:

- **Condizione** (in cima a volte): quando è possibile, la condizione va scritta come qualcosa che **possa essere rilevato dal sistema o da un attore**.
- **Gestione**: è riassumibile in un unico passo, oppure può comprendere una sequenza di passi al termine dei quali solitamente lo scenario si fonde di nuovo con lo scenario principale di successo.

Possono essere usate per gestire almeno tre tipi di situazioni:

- **L'attore** vuole che l'esecuzione del caso d'uso proceda in modo **diverso** da quanto previsto nello scenario principale
- Il caso d'uso deve procedere **diversamente da quanto previsto** nello scenario principale ed è **il sistema ad accorgersene** mentre esegue un'azione o effettua una validazione
- Un passo dello scenario principale descrive un'azione "generica" o "astratta", mentre le estensioni relative a questo passo descrivono le possibili azioni "specifiche" o "concrete" per eseguire il passo. (*Astratto*: il cliente paga. *Concreto*: paga con carta/contanti/buoni)

#### **Estensione 1a: lavora su un menu esistente**

#	Attore	Sistema
<b>1a.1</b>	Sceglie di lavorare su un menù precedentemente creato.	Mostra i dettagli (titolo, sezioni, voci) del menù scelto
	<i>Prosegue con il passo 3 dello scenario principale</i>	

#### **Estensione 1b: crea un menu come copia di un altro**

#	Attore	Sistema
<b>1b.1</b>	Crea una copia di un menù esistente	Mostra i dettagli (titolo, sezioni, voci) del menù scelto. Il titolo è "Copia di [titolo dell'originale]"
	<i>Prosegue con il passo 3 dello scenario principale</i>	

#### **Estensione 3a: lavora su una sezione esistente**

#	Attore	Sistema
<b>3a.1</b>	Sceglie una sezione precedentemente creata.	Mostra la sezione con il suo nome e le voci contenute
	<i>Prosegue con il passo 4 dello scenario principale</i>	

#### **Estensione 3b: elimina una sezione esistente**

#	Attore	Sistema
<b>3b.1</b>	Elimina una sezione precedentemente creata.	Mostra il menù aggiornato (senza la sezione eliminata). Se sono state eliminate tutte le sezioni e il menù era indicato come completo, segnala che non lo è più.
	<i>Se vuole lavorare su un'altra sezione torna al passo 3 se no prosegue con il passo 5.</i>	

Dal progetto di laboratorio 2017/2018.

## Estensione (3-4)a: modifica il titolo del menù

#	Attore	Sistema
(3-4)a.1	Specifica un nuovo titolo per il menù.	Mostra i dettagli del menu aggiornati (con il nuovo titolo)

## Estensione (3-5)a: conclude anticipatamente

#	Attore	Sistema
	Va al passo 6 dello scenario principale	

## Estensione (3-5)b: elimina questo menù

#	Attore	Sistema
(3-5)b.1	Elimina definitivamente il menù su cui sta lavorando	Notifica l'avvenuta eliminazione
	Va al passo 6 dello scenario principale	

(1a => prima estensione del passo 1)

Da notare il titolo dell'estensione che specifica anche la situazione di inizio del ramo alternativo

### Scrittura

Scrivere i casi d'uso in uno stile essenziale: bisogna ignorare l'interfaccia utente e concentrarsi sull'obiettivo utente.

**Sì:**

- L'amministratore si identifica
- Il Sistema autentica l'identità

**No:**

- L'amministratore inserisce ID e password nella finestra di dialogo
- Il sistema autentica l'amministratore
- Il sistema visualizza la finestra "edit users"

Lo stile concreto non è adatto durante le attività iniziali dell'analisi dei requisiti; sono un valido aiuto invece nella progettazione di GUI concrete e dettagliate.

### Stile essenziale e conciso!

La narrativa di un caso d'uso viene espressa a livello delle **intenzioni** dell'utente e delle **responsabilità** del sistema, senza alcun riferimento ad azioni concrete.

Queste intenzioni e responsabilità rimangono indipendenti dai dettagli tecnologici e dai movimenti degli attori (soprattutto quelli relativi all'uso della UI).

### A scatola nera

Il sistema è descritto come dotato di responsabilità.

Si deve descrivere **cosa deve fare** (comportamento o requisiti funzionali) senza decidere **come lo farà** (progettazione).

*Esempio:* Il sistema registra la vendita

Si considera il sistema a scatola nera, **evitando** di prendere decisioni sul "come".

**No:**

- Il Sistema memorizza la vendita in una base di dati
- Il sistema esegue un'istruzione SQL INSERT per la vendita

Successivamente, durante la progettazione, andrà creata una soluzione che soddisfa le specifiche.

## [Adottare il punto di vista dell'attore](#)

Un caso d'uso è un insieme di istanze di casi d'uso in cui ciascuna istanza è una sequenza di azioni che un sistema esegue per produrre **un risultato osservabile e di valore per uno specifico attore**

Bisogna scrivere i requisiti concentrandosi sugli utenti o attore di un sistema, chiedendosi quali sono i loro obiettivi e le situazioni tipiche. Ci si deve concentrare sulla comprensione di ciò che l'attore considera un risultato di valore.

## [Come trovarli](#)

1. Scegliere i confini del sistema
2. Identificare gli attori primari (cioè coloro che raggiungono i propri obiettivi attraverso l'uso dei servizi del sistema)
3. Identificare gli obiettivi di ciascun attore primario
4. Definire i casi d'uso che soddisfano gli obiettivi degli utenti (il loro nome va scelto in base all'obiettivo)

# Fase 2: Elaborazione

venerdì 27 marzo 2020 18:22

13 marzo 2020

*L'elaborazione è la serie iniziale di iterazioni durante le quali il team esegue un'indagine seria, implementa il nucleo dell'architettura, chiarisce la maggior parte dei requisiti e affronta le problematiche di alto rischio.*

Il codice progettato deve essere di qualità perché verrà già usato nel sistema finale. Non si tratta di prototipi "usa e getta".

Si sviluppano immediatamente le parti più a rischio, che potrebbero far fallire il progetto.

- Viene programmato e verificato il nucleo (rischioso) dell'architettura software
- Viene scoperta e stabilizzata la maggior parte dei requisiti
- I rischi maggiori sono attenuati o rientrano

Il rischio analizzato è di tipo tecnico, legato all'usabilità o all'incertezza dello sforzo.

I requisiti e le iterazioni si organizzano, oltre che per il rischio, anche per la **copertura** (le prime iterazioni devono coprire tutto il sistema in ampiezza e poco in profondità) e **criticità** (le funzioni che il cliente considera di elevato valore di business)

## Iterazione 1

Non si implementano tutti i requisiti in una volta sola: nella prima iterazione i requisiti sono dei sottoinsiemi dei requisiti o dei casi d'uso completi.

Si inizia la programmazione di qualità-produzione e il test per un sottoinsieme dei requisiti; si inizia lo sviluppo **prima** che l'analisi di tutti i requisiti sia stata completata, al contrario del processo a cascata.

I casi d'uso non sono completati su un'unica iterazione, ma potenzialmente su molte. Si lavora spesso su più casi d'uso in contemporanea.

l'elaborazione produce degli artefatti:

Elaborato	Commento
Modello di Dominio	È una visualizzazione dei concetti del dominio, simile a un modello statico delle informazioni delle entità del dominio.
Modello di Progetto	È l'insieme dei diagrammi che descrivono la progettazione logica. Comprende diagrammi delle classi software, diagrammi di interazione degli oggetti, diagrammi dei package e così via.
Documento dell'Architettura Software	Un aiuto per l'apprendimento che riassume gli aspetti principali dell'architettura e la loro risoluzione nel progetto. È un riepilogo delle idee di progettazione più significative all'interno del sistema e delle loro motivazioni.
Modello dei Dati	Comprende gli schemi della base di dati e le strategie di mapping tra la rappresentazione a oggetti e la base di dati.
Storyboard dei casi d'uso, Prototipi UI	Una descrizione dell'interfaccia utente, della navigazione, dei modelli di usabilità e così via.

# Modellazione del business: modello del dominio

sabato 28 marzo 2020 11:48

È una disciplina che è maggiormente collocata nelle iterazioni iniziali.

Il modello del dominio parte durante l'elaborazione

È un artefatto che ha in input i modelli dei casi d'uso e si evolve per mostrare i concetti significativi.

È una sorta di dizionario visivo.

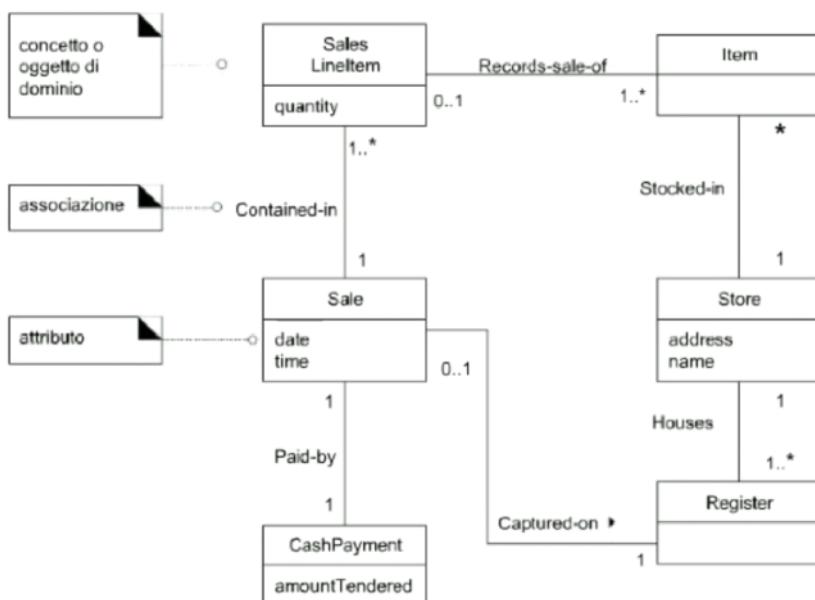
Il modello di dominio influenzerà i contratti delle operazioni e il glossario, artefatti che si raffineranno nelle varie iterazioni incrementalmente.

Il modello di dominio servirà per il modello di progetto.

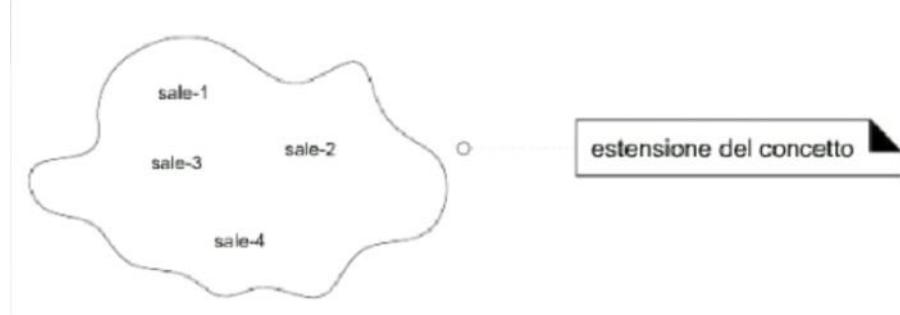
Si pone di rappresentare visualmente delle classi concettuali (non sono oggetti software). Quello che si vuole ottenere è un distillato delle proprietà dei concetti, gli elementi e le relazioni che compongono il dominio dell'applicazione.

Si usano diagrammi UML con:

- **Oggetti** di dominio-classi concettuali
- **Associazioni** tra classi concettuali
- **Attributi** di classi concettuali
- Non appaiono operazioni (firme di metodi o responsabilità)

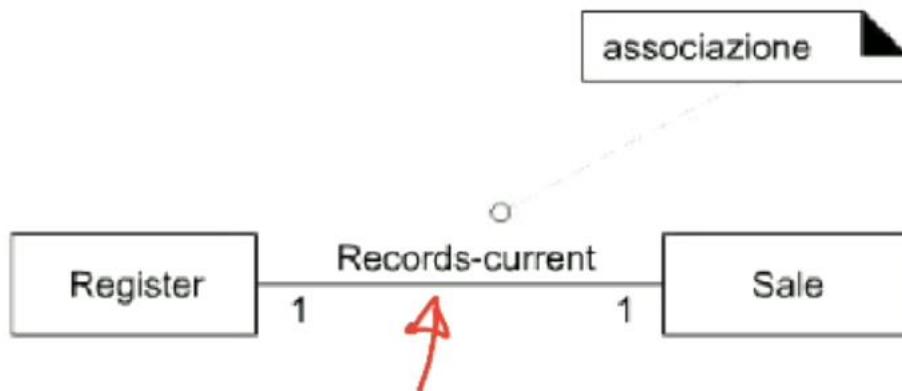


- Il **Simbolo** è una parola o un'immagine usata per rappresentare la classe concettuale
- L'**intensione** è la definizione (in linguaggio naturale) della classe concettuale
- L'**estensione** è l'insieme degli oggetti della classe concettuale



Una classe concettuale rappresenta un **concetto** del mondo reale o nel dominio di interesse di un sistema che si sta modellando.

**Associazione:** una relazione tra classi (tra le istanze) che indica una connessione significativa e interessante.

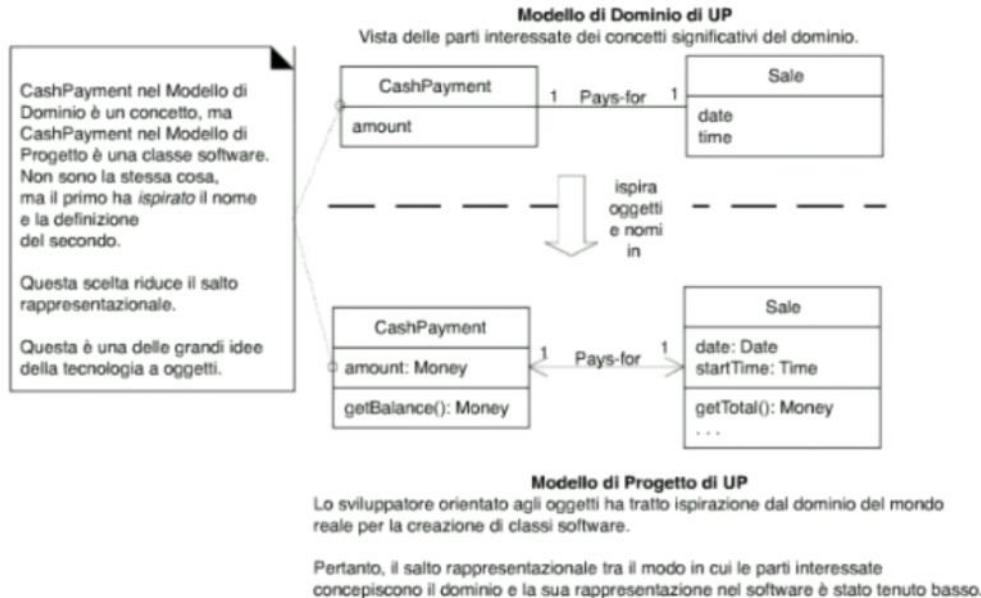


**Attributo:** un valore logico (un dato, una proprietà elementare) degli oggetti di una classe. Ciascun oggetto della classe ha un proprio valore (separato) per quella proprietà.



Lo scopo del modello di dominio è:

- **Comprendere** il dominio del sistema da realizzare e il suo vocabolario
- Definire un **linguaggio comune** che abiliti la comunicazione tra le diverse parti interessate al sistema (cliente e progettisti possono capirsi meglio)
- Come **fonte di ispirazione** per la progettazione dello strato del dominio



## Creare un modello di dominio

Ci restringiamo ai requisiti scelti per la progettazione nell'iterazione corrente:

1. Trovare le classi concettuali
2. Disegnarle come classi in un diagramma delle classi UML
3. Aggiungere le associazioni
4. Aggiungere gli attributi

### Trovare le classi concettuali

Ci restringiamo ai requisiti scelti per la progettazione nell'iterazione corrente (non si va a considerare altre cose).

Ci sono tre modi:

- Riuso-modifica di modelli esistenti (da altri progetti)
- Utilizzo di elenchi di categorie (preparando un elenco di classi candidate; si usa comunque della conoscenza pregressa)
- Analisi linguistica (delle descrizioni testuali di un dominio):
  - Il mapping nome-classe non è automatico
  - Il linguaggio naturale è ambiguo; spesso non è facile comprenderlo. Il feedback del cliente è essenziale
  - Le fonti sono i casi d'uso descritti in modo dettagliato.

Un modello di dominio conterrà una collezione di astrazioni e termini del dominio che i modellatori considerano significative.

### Classe descrizione

Una classe descrizione contiene informazioni che descrivono qualcos'altro. È utile avere un'associazione che collega la classe e la sua classe descrizione.

L'utilità principale è la riduzione delle ridondanze.

### Associazioni

Un'associazione rappresenta una relazione tra due o più classi che indica una connessione significativa tra le istanze di quella classe.

È utile includere:

- Associazioni la cui conoscenza della relazione deve essere conservata per una qualche durata
- Associazioni derivate dall'elenco di associazioni comuni

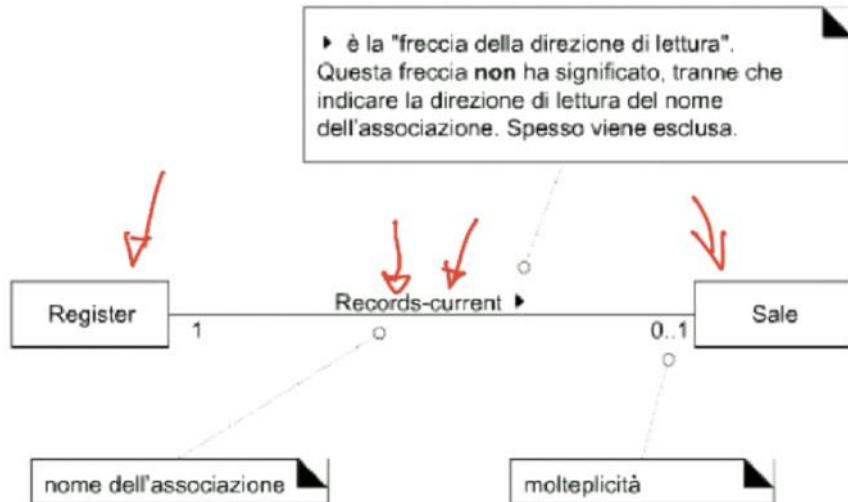
È importante introdurre solo le associazioni davvero rilevanti al dominio modellato.

Una associazione deve essere caratterizzata con:

- Un nome significativo seguendo la traccia NomeClasse-FraseVerbale-NomeClasse
- Molteplicità e direzione di lettura

Si ricorda che un'associazione è per natura bidirezionale. La direzione di lettura non è una specifica

di visibilità.



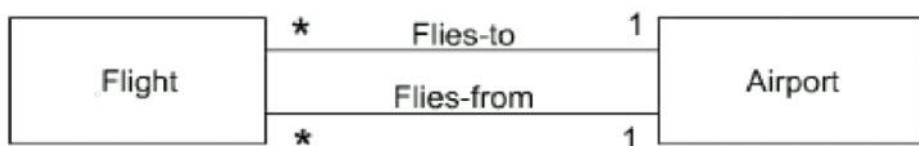
È possibile avere dei **ruoli** nelle associazioni. Ciascuna estremità di una associazione è un ruolo.  
I ruoli possono avere, opzionalmente:

- Espressione di **molteplicità**
- **Nome**
- **Navigabilità**

La molteplicità di un ruolo definisce quante istanze di una classe possono essere associate ad una istanza di un'altra.

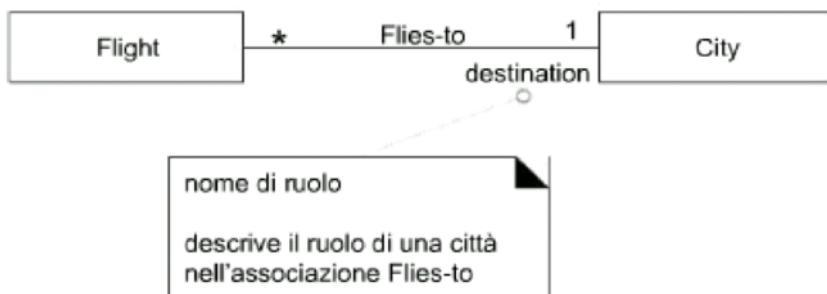
Il valore di una molteplicità comunica quante istanze possono essere associate in modo valido a un'altra istanza, in un particolare momento piuttosto che in un arco di tempo.

È possibile che due classi siano collegate da più di una associazione.

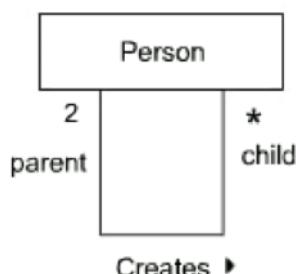


$\star$	T	zero o più; "molti"
$1..\star$	T	uno o più
$1..40$	T	da uno a 40
5	T	esattamente 5
3, 5, 8	T	esattamente 3, 5 o 8

È anche possibile indicare il **nome** di un ruolo.



Possiamo anche avere associazioni riflessive. I nomi di ruolo sono in questo caso utili per indicare la molteplicità.



### Aggregazione/Composizione

L'aggregazione in UML è un tipo di associazione che suggerisce una relazione intero-parte. Un

esempio di aggregazione è la relazione fra un'automobile e le sue ruote.

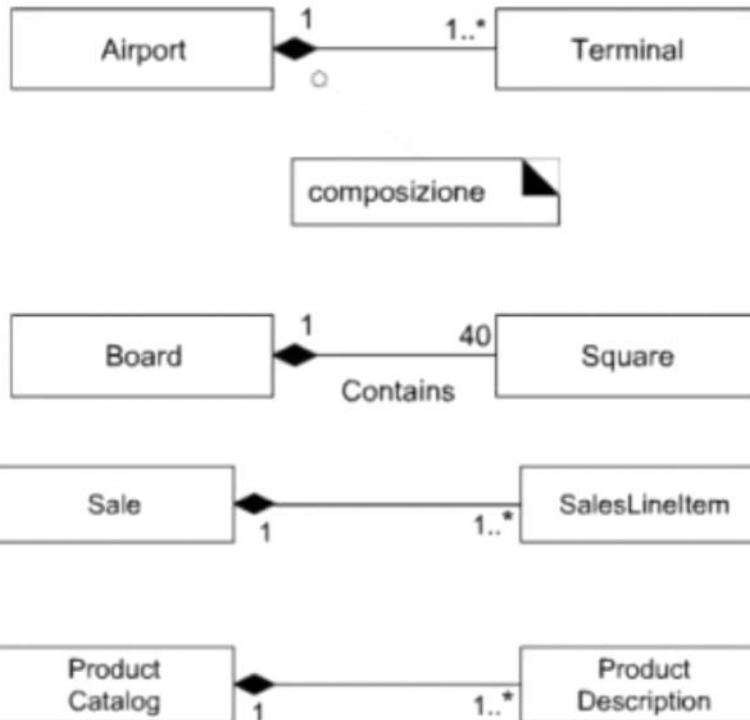
La **Composizione** (o aggregazione composta) è un tipo di **forte aggregazione intero-parte**:

- Ciascuna istanza della parte appartiene ad **una sola** istanza del composto alla volta
- Ciascuna parte deve sempre appartenere a **un** composto
- La vita delle parti è limitata da quella del composto: le parti possono essere create dopo il composto (ma non prima) e possono essere distrutte prima del composto (ma non dopo)

Chiarificazione:

*'l'aggregazione tra automobile e le sue ruote non è una composizione.'*

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



## Attributi

Un attributo di una classe rappresenta un valore (un dato) degli oggetti di quella classe.

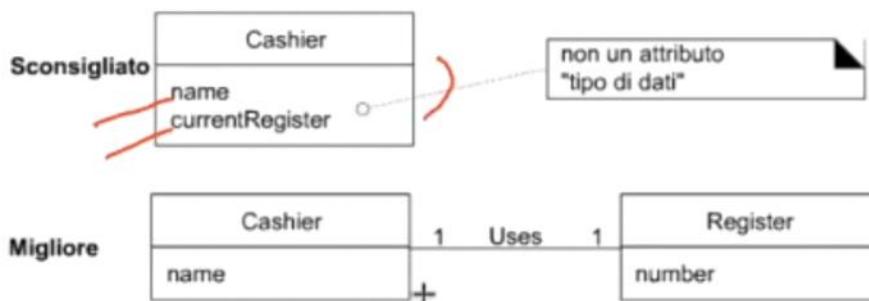
Sono da aggiungere attributi che:

- Sono tipi di dati primitivi (boolean, date, number, char, string...)
- Sono tipi enumerativi (tipo servizio: oro, argento)

Un attributo deve essere caratterizzato con:

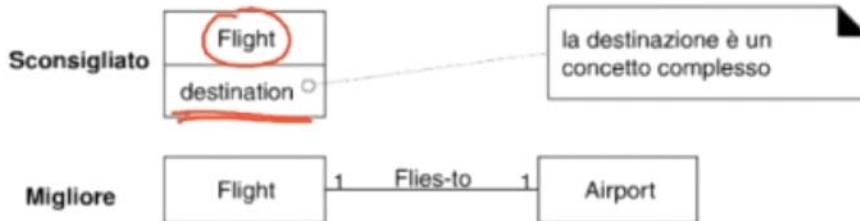
- Origine: derivato/non derivato
- Tipo di dato

Gli attributi in un modello di dominio devono preferibilmente essere di **tipo di dato**



Se nel mondo reale non pensiamo a un concetto X come a un numero, un testo o un valore di un tipo di dato, allora probabilmente X è una classe concettuale, non un attributo.

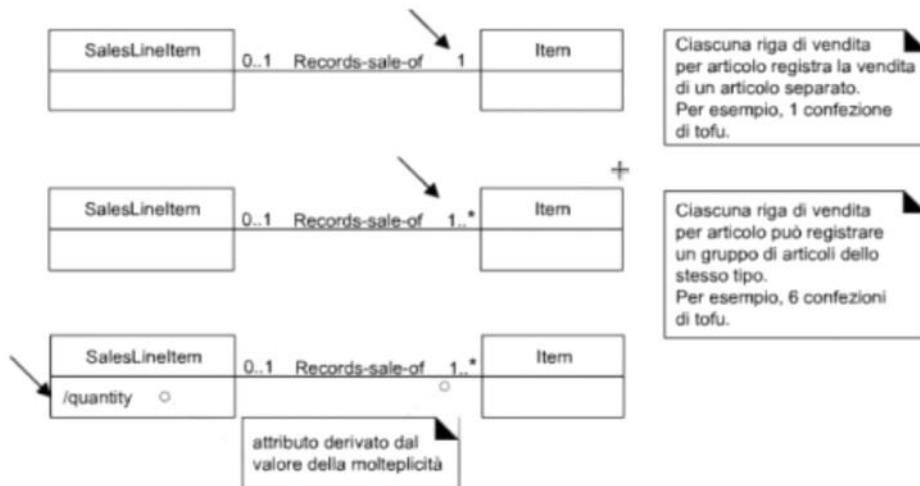
Fare attenzione a correlare le classi concettuali con un'associazione, non con un attributo!



Gli attributi sono mostrati nella seconda sezione del rettangolo per una classe. Opzionalmente è possibile mostrare il loro tipo e altre informazioni, ad esempio la visibilità o il tipo di lettura.



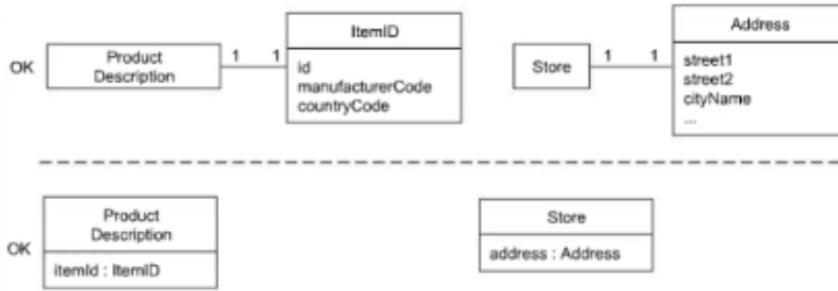
Un attributo può essere calcolato o derivato dalle informazioni contenute degli oggetti associati.  
Un attributo derivato può essere indicato dal simbolo '/' prima del nome dell'attributo.



## Classi tipo di dato

Quando introdurle?

- Dati composti da sezioni separate (numero di telefono, nome)
- Quando ci sono operazioni associate ai dati, come la validazione o il parsing (codice fiscale)
- Quantità con unità di misura (totale da pagare è caratterizzato da una valuta)



## Verifiche

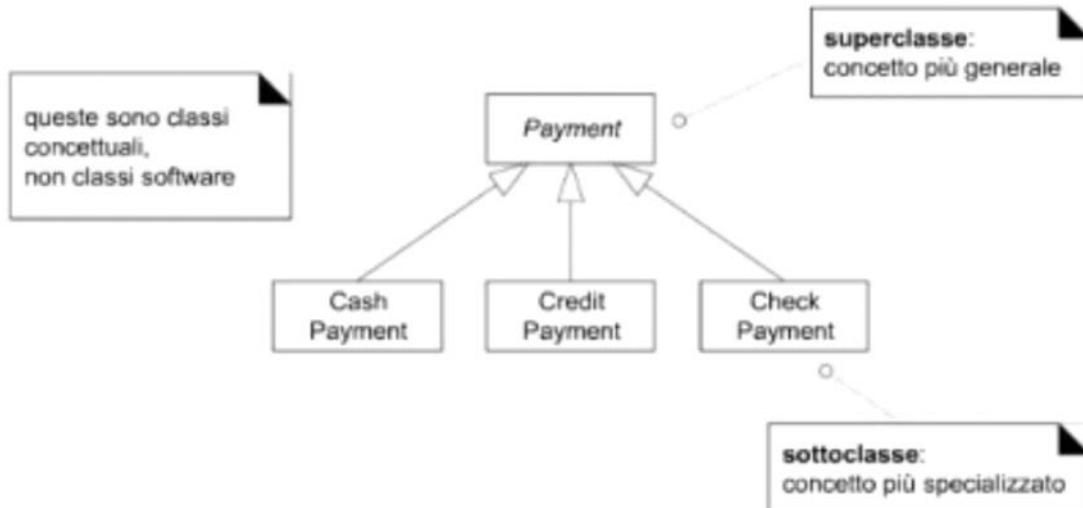
È necessario verificare le classi concettuali introdotte:

- Alternativa classe classe-attributo attributo
- Classi descrizione
- Verificare le associazioni: Indipendenza delle associazioni diverse che sono relative alle stesse classi
- Verificare gli attributi: non introdurre attributi per riferirsi ad altre classi concettuali (chiavi esterne). Bisogna usare le associazioni in questo caso.

# Modellazione dominio

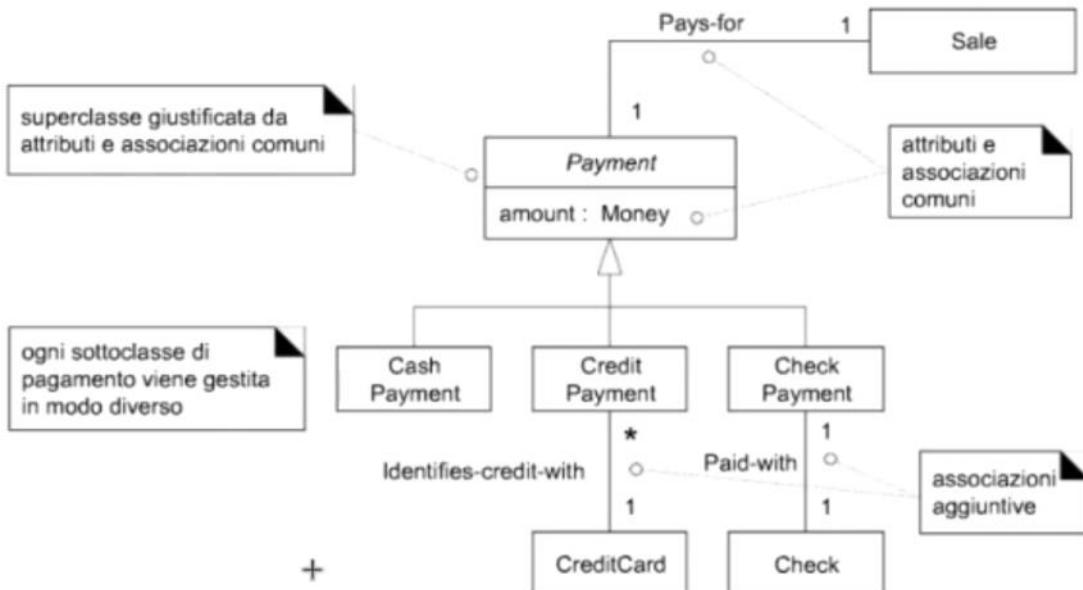
sabato 28 marzo 2020 13:17

La **generalizzazione** è un'astrazione basata sull'identificazione di caratteristiche comuni tra concetti, che porta a definire una relazione tra un concetto più generale (superclasse) e un concetto più specializzato o specifico (sottoclasse). Vale il principio di **sostituibilità**.

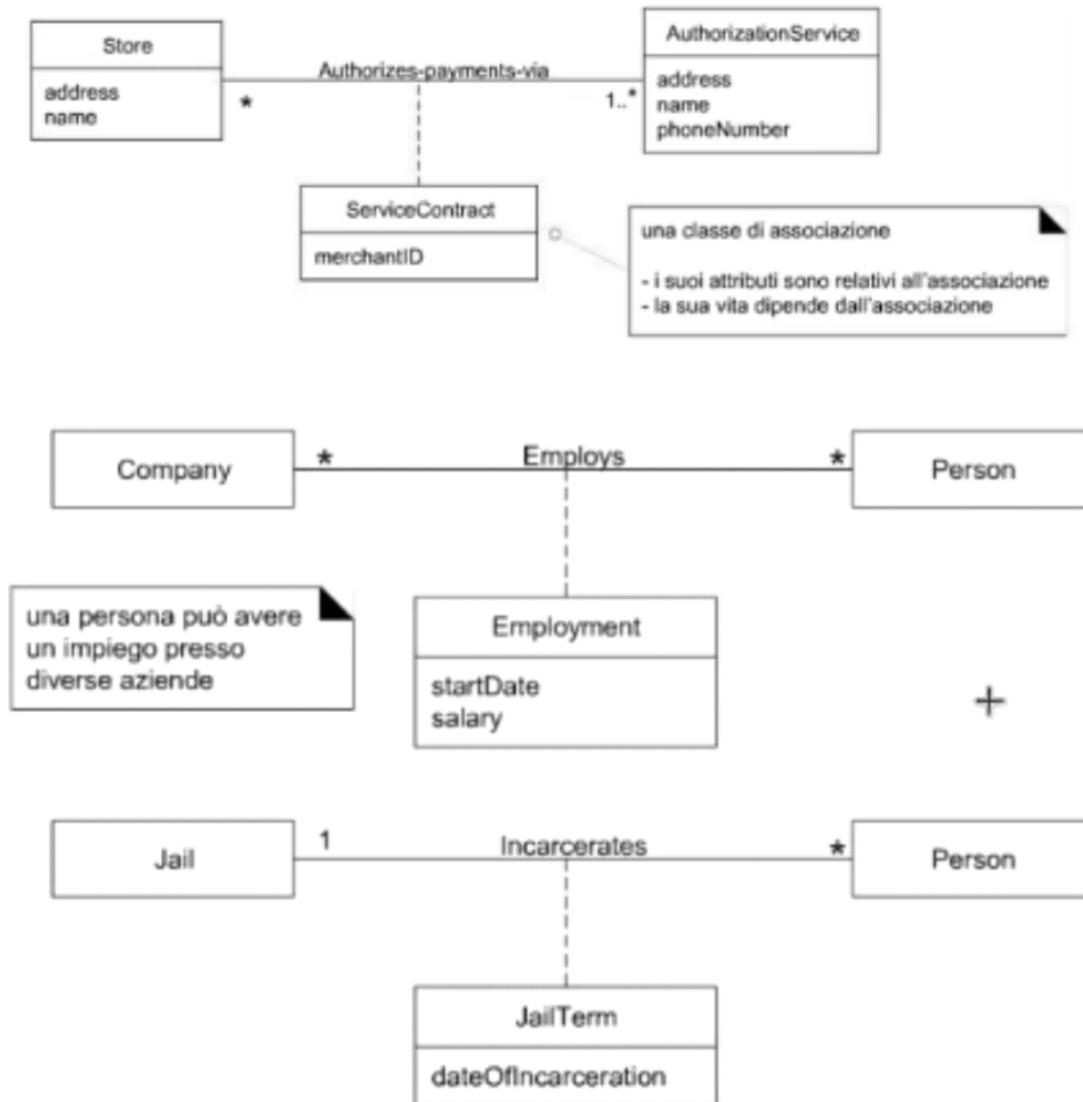


Una classe concettuale è **astratta** se ciascun suo elemento è anche elemento di una delle sue sottoclassi.





Le classi di associazioni permettono di aggiungere attributi e altre caratteristiche alle associazioni:



# Diagramma di sequenza di sistema

lunedì 30 marzo 2020 20:06

Si tratta di un elaborato della disciplina dei requisiti che illustra **eventi** di input e di output relativi ai sistemi in discussione.

Gli SSD sono espressi **attraverso i diagrammi di sequenza UML**.

Il sistema è modellato come una scatola nera. Vediamo gli eventi in input e i risultati in output. Non ha come obiettivo di vedere "come" svolge il lavoro il sistema, ma cosa ci si aspetta che dia come risultato.

Si usa un SSD per ogni caso d'uso, uno per ogni scenario principale o alternativo.

Lo SSD è l'input per i contratti delle operazioni e per la progettazione degli oggetti.

I casi d'uso descrivono il modo in cui gli attori esterni interagiscono con il sistema software che interessa creare.

Durante un'interazione con il sistema software, un attore genera degli **eventi di sistema** che costituiscono un input per il sistema, di solito per richiedere l'esecuzione di alcune **operazioni di sistema**.

- Le operazioni di sistema sono operazioni che il sistema deve definire proprio per gestire tali eventi
- Un evento è qualcosa di importante o degno di nota che avviene durante l'esecuzione di un sistema
- Un evento di sistema è un evento esterno al sistema, di input, di solito generato da un attore per interagire con il sistema

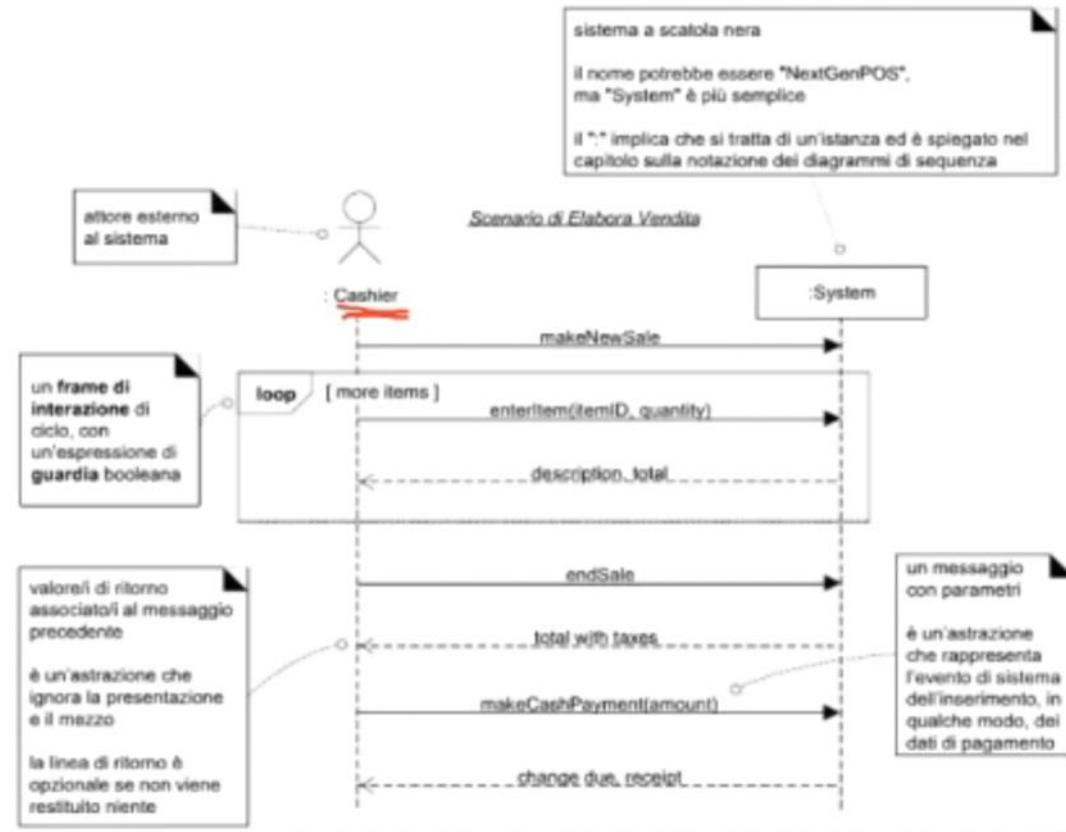
I diagrammi di sequenza sono utili per illustrare interazioni tra attori e le operazioni iniziate da essi.

I diagrammi di sequenza di sistema sono una figura che mostra, per un particolare scenario di un caso d'uso, gli **eventi** generati dagli attori esterni al sistema, il **loro ordine** e gli eventi inter-sistema.

Un sistema reagisce a tre cose:

- **Eventi esterni** da parte di attori umani o sistemi informatici
- **Eventi temporali**
- **Guasti o eccezioni**

Il software deve essere progettato proprio per gestire questi eventi e generare delle risposte.

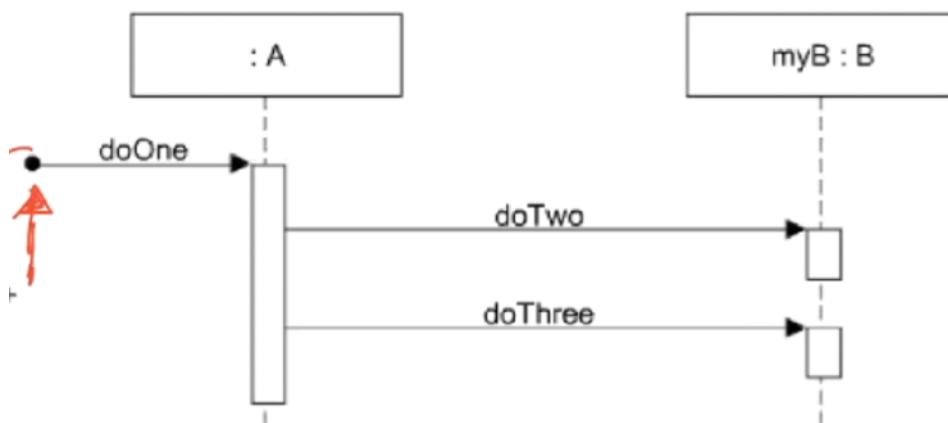


Usualmente un SSD mostra gli eventi di sistema per un solo scenario di un caso d'uso e può essere generato per ispezione da tale scenario.

Un SSD mostra:

- l'attore primario del caso d'uso
- Il sistema in discussione
- I passi che rappresentano le interazioni tra il sistema e l'attore

Le interazioni iniziate dall'attore primario nei confronti del sistema sono mostrate come messaggi con parametri.

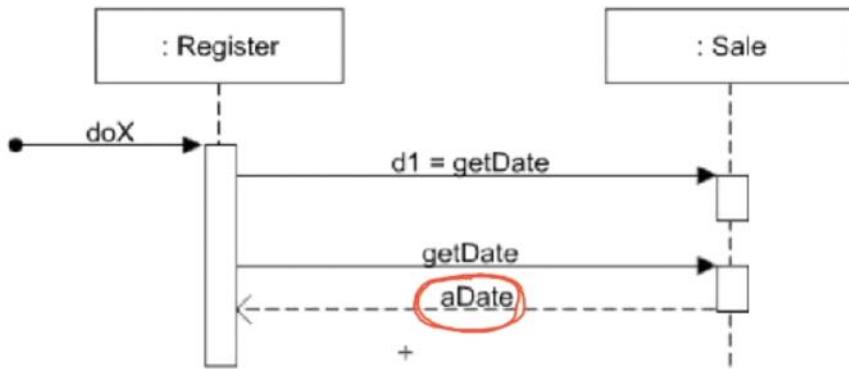


A sinistra, il messaggio che scaturisce l'SSD.

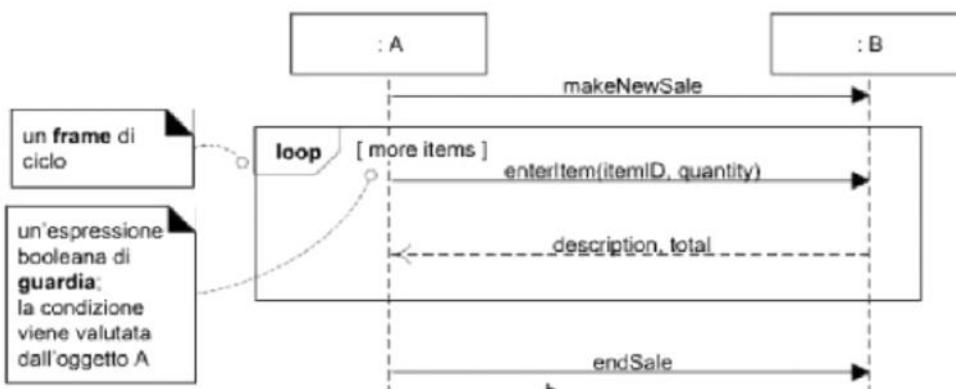
Nei rettangoli sono indicati gli attori, istanze (:) di classi A e B. myB è l'istanza nominata.

Il rettangolo a sinistra in verticale è la **lifeline**, la linea di esecuzione. Indica l'attività dell'attore verso quello del sistema.

Gli eventi sono indicati dalle frecce nominate e possono anche presentare dei parametri.

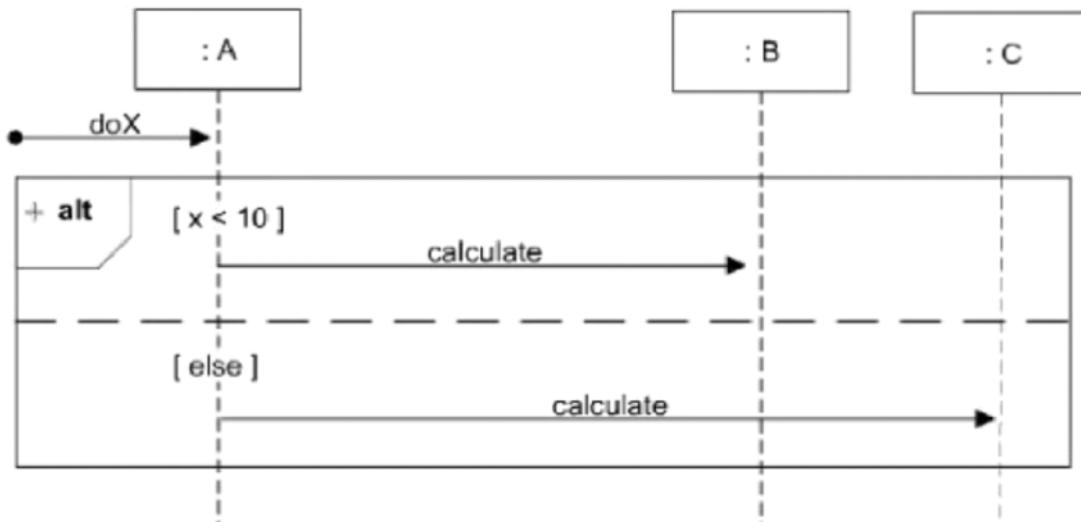


La freccia tratteggiata è una risposta. La lifeline del sistema inizia con una attività e termina con la risposta.

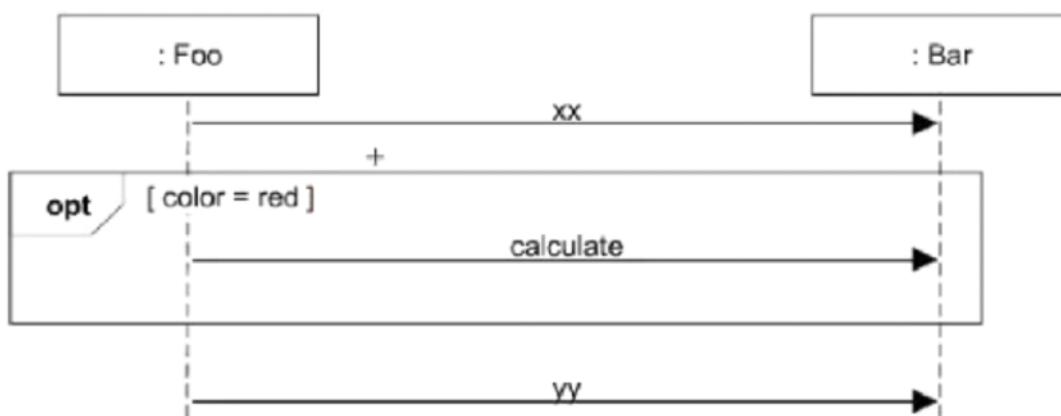


Si possono racchiudere in dei frame delle operazioni. Deve avere un'etichetta che descriva il funzionamento del frame.

Operatore frame	Significato
alt	Frammento alternativo per logica mutuamente espressa nella guardia (un'istruzione <i>if-else</i> di Java o del C).
opt	Frammento opzionale che viene eseguito se la guardia è vera (un'istruzione <i>if</i> ).
loop	Frammento da eseguire ripetutamente finché la guardia è vera (un'istruzione <i>while</i> o <i>for</i> ). Si può anche scrivere <i>loop(n)</i> per indicare un ciclo da ripetere n volte. Può rappresentare anche l'istruzione <i>foreach</i> del C# o l'istruzione <i>for "avanzata"</i> di Java.
par	Frammenti che vengono eseguiti in parallelo.
region	Regione critica all'interno della quale può essere in esecuzione un solo thread.

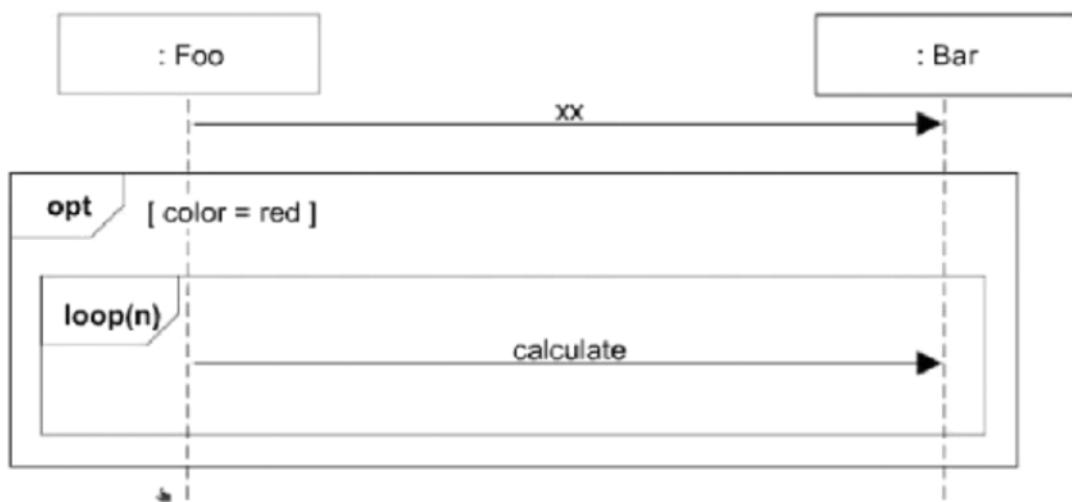


Da notare tra parentesi quadre la guardia.



L'opzione può anche non avere una guardia per indicare una scelta totalmente a carico dell'utente

I frame possono essere annidati uno nell'altro:



# Contratti

sabato 11 aprile 2020 11:25

I contratti sono le caratteristiche delle operazioni di sistema. Vengono definiti ad ogni iterazione! Fanno parte della disciplina dei **requisiti**.

I **contratti delle operazioni** usano **pre-condizioni** e **post-condizioni** per descrivere nel dettaglio i cambiamenti agli oggetti in un modello di dominio.

## Contratto CO2: enterItem

---

**Operazione:** enterItem(itemID: ItemID, quantity: integer)

**Riferimenti:** casi d'uso: Elabora Vendita

**Pre-condizioni:** è in corso una vendita s

**Post-condizioni:**

- è stata creata un'istanza sli di SalesLineItem (*creazione di oggetto*).
- sli è stata associata con la Sale (vendita) corrente s (*formazione di collegamento*).
- sli è stata associata con una ProductDescription, in base alla corrispondenza con itemID (*formazione di collegamento*).
- sli.quantity è diventata quantity (*modifica di attributo*).

- **Operazione:** Nome e parametri (firma) 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. Si tratta di ipotesi non banali, che dovrebbero essere comunicate al lettore.
- **Post-condizione:** è la sezione più importante. Descrive i cambiamenti di stato degli oggetti nel Modello di Dominio dopo il completamento dell'operazione.
  - Le operazioni di sistema sono operazioni che il sistema, considerato scatola nera, offre nell'interfaccia pubblica.
  - Le operazioni di sistema possono essere identificate mentre si abbozzano gli SSD (che mostrano eventi di sistema: implica che il sistema definisca un'operazione di sistema per gestire quell'evento)

### Post condizioni:

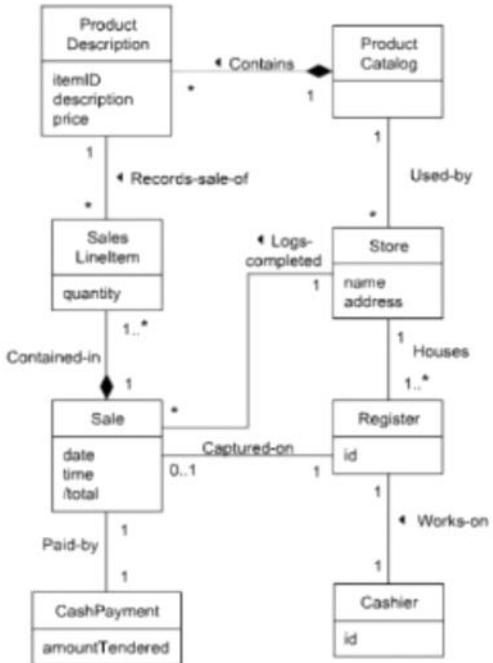
Descrivono i cambiamenti nello stato degli oggetti del modello di dominio. I cambiamenti dello stato del modello di dominio comprendono gli **oggetti creati**, i **collegamenti formati o rotti**, e gli **attributi modificati**.

Le post condizioni **non sono azioni da eseguire nel corso dell'operazione**, si tratta di osservazioni sugli oggetti del modello di dominio che risultano al **termine** dell'operazione. Il "come" vengono soddisfatte le post, è compito della progettazione.

### Contratto per un'operazione di sistema

Describe (specialmente la post-condizione) i cambiamenti nello stato del sistema causato dall'esecuzione di un'operazione di sistema.

Questo cambiamento va espresso in **termini di oggetti e collegamenti del Modello di Dominio**, secondo un **punto di vista concettuale** (ovvero si parla di oggetti e collegamenti del mondo reale o nel modello di interesse).



### Pre-condizioni

Descrivono le ipotesi significative sullo stato del sistema **prima** dell'esecuzione dell'operazione.

### Scrivere contratti

- Identificare le operazioni di sistema dagli SSD. A ogni evento faccio corrispondere un'operazione
- 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. Non è necessario farlo per tutti (eccetto che nel laboratorio)
- Per descrivere le post-condizioni si usano anche le seguenti sotto-categorie:
  - Creazione o cancellazione di oggetto (o istanza)
  - Formazione o rottura di collegamento
  - Modifica di attributo

## Fase 3: Progettazione

martedì 14 aprile 2020 12:49

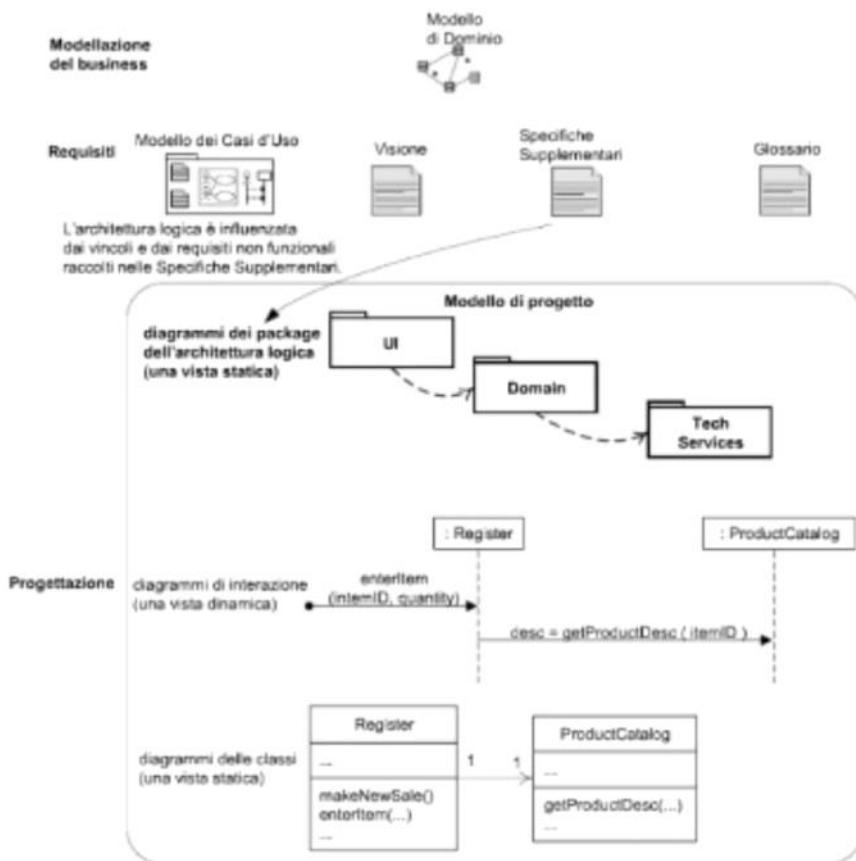
Nei requisiti il fuoco è "fare la cosa giusta" (capire alcuni degli obiettivi più importanti e le relative regole e vincoli), mentre nella progettazione è "fare la cosa bene" (progettare abilmente una soluzione che soddisfa i requisiti per l'iterazione corrente).

# Architettura Logica

martedì 14 aprile 2020 12:51

La progettazione di un tipico sistema orientato agli oggetti è basata su *diversi strati architetturali*, 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 UML.



L'architettura logica di un sistema software è la macro-organizzazione su larga scala delle classi software in package (**namespace**), **sottoinsiemi** e **strati**.

È chiamata architettura "logica" perché **non** vengono prese decisioni su come questi elementi siano distribuiti sui processi o sui diversi computer fisici di una rete (queste fanno parte dell'architettura di deployment): *platform independent architecture*.

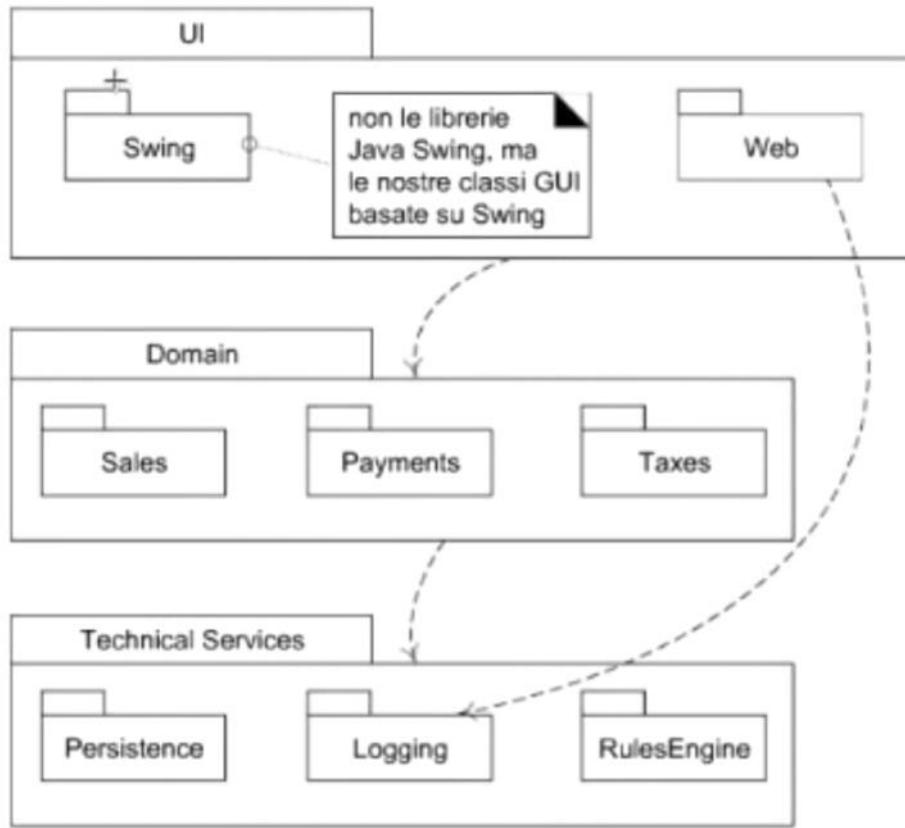
## Layer

È un gruppo di classi software, packages, sottoinsiemi con responsabilità condivisa su un aspetto importante del sistema.

Gli strati di un'applicazione software comprendono normalmente:

- **User interface**, oggetti software per gestire l'interazione con l'utente e la gestione degli eventi
- **Application logic o domain objects** (logica applicativa o oggetti del dominio), oggetti software che rappresentano concetti di dominio
- **Technical services**, oggetti e sottoinsiemi d'uso generale che forniscono servizi tecnici di supporto





```

/** Strato UI **/
com.mycompany.nextgen.ui.swing
com.mycompany.nextgen.ui.web

/** Strato DOMAIN **/
// package specifici del progetto NextGen
com.mycompany.nextgen.domain.sales
com.mycompany.nextgen.domain.payments

/** Strato TECHNICAL SERVICES **/
// il nostro strato di persistenza (accesso alla base di dati)
com.mycompany.service.persistence

// terze parti
org.apache.log4j
org.apache.soap.rpc

/** Strato FOUNDATION **/
// package foundation creati dal nostro team
com.mycompany.util

```

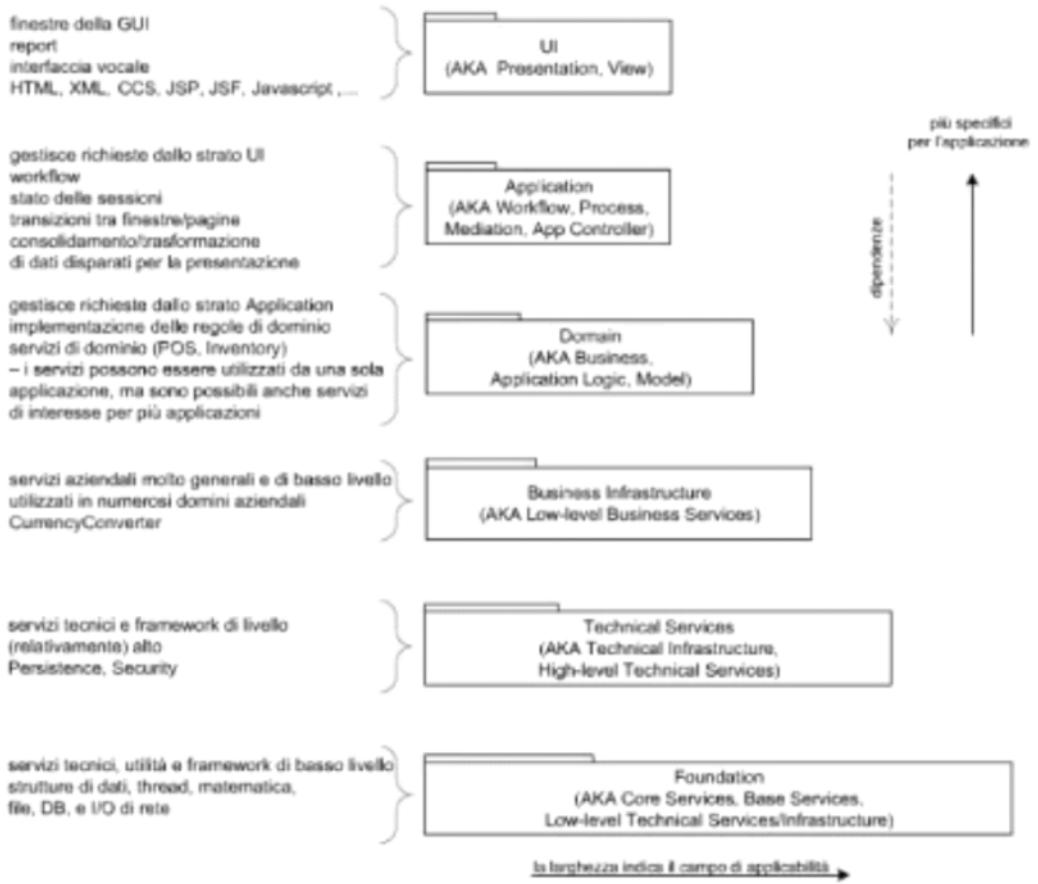
L'obiettivo dell'architettura a strati è la suddivisione di un sistema complesso in un insieme di elementi software che, per quanto possibile, possano essere sviluppati e modificati ciascuno indipendentemente dagli altri.

**Separation of concerns** (separazione degli interessi): ridurre l'accoppiamento e le dipendenze, aumenta la possibilità di riuso, facilita la manutenzione e aumenta la chiarezza.

**Alta coesione:** in uno strato le responsabilità degli oggetti devono essere fortemente correlate e non devono essere mischiare con le responsabilità degli altri strati.

*Esempio:*

gli oggetti dell'interfaccia utente non devono implementare logica applicativa.



## Come va progettata la logica applicativa con gli oggetti?

L'approccio consigliato consiste nel creare degli oggetti software con nomi e informazioni simili al dominio del mondo reale, e assegnare ad essi responsabilità della logica applicativa.

Un oggetto software di questo tipo è chiamato un **oggetto di dominio**, rappresenta una cosa nello spazio di dominio del problema e ha una logica applicativa o di business correlata.

## Principio di separazione Modello-Vista

1. **Non relazionare o accoppiare oggetti non UI con oggetti UI**: gli oggetti non UI non devono essere connessi o accoppiati direttamente agli oggetti UI (non si deve permettere ad un oggetto software di "dominio", non UI, di avere un riferimento ad un oggetto UI).
2. **Non incapsulare la logica dell'applicazione in metodi UI**: non mettere la logica applicativa (calcolo delle imposte) nei metodi di un oggetto dell'interfaccia utente. Gli oggetti UI dovrebbero solo inizializzare gli elementi dell'interfaccia utente, ricevere eventi UI e delegare le richieste di logica applicativa agli oggetti non UI.

Il **modello** è sinonimo per lo strato degli oggetti del dominio.

La **vista** è sinonimo per gli oggetti dell'interfaccia utente, come finestre, pagine web, applet, report.

Gli oggetti del **modello** (dominio) non devono avere una conoscenza diretta degli oggetti della **vista** (UI).

Le classi di dominio **incapsulano** le informazioni e il comportamento relativi alla logica applicativa.

Le classi della vista sono relativamente leggere, 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.

## Vantaggi

- Favorisce la definizione coesa dei modelli
- Consente lo sviluppo separato degli strati del modello e dell'interfaccia utente
- Minimizza l'impatto sullo strato del dominio dei cambiamenti dei requisiti relativi all'interfaccia
- Consente di connettere facilmente nuove viste a uno strato del dominio esistente
- Consente viste multiple simultanee sugli stessi oggetti modello
- Consente l'esecuzione dello strato modello indipendente da quella dello strato dell'interfaccia utente (batch o messaggi)
- Consente un porting facile dello strato del modello a un altro framework per l'interfaccia utente

## SSD, operazioni di sistema e strati

Gli SSD mostrano le operazioni di sistema ma nascondono gli oggetti specifici della UI. Gli oggetti dello strato UI inoltreranno (o delegheranno) le richieste da parte dello strato UI allo strato del dominio.

I messaggi inviati dallo strato UI allo strato del dominio saranno i messaggi mostrati negli SSD.

## Codifica degli oggetti

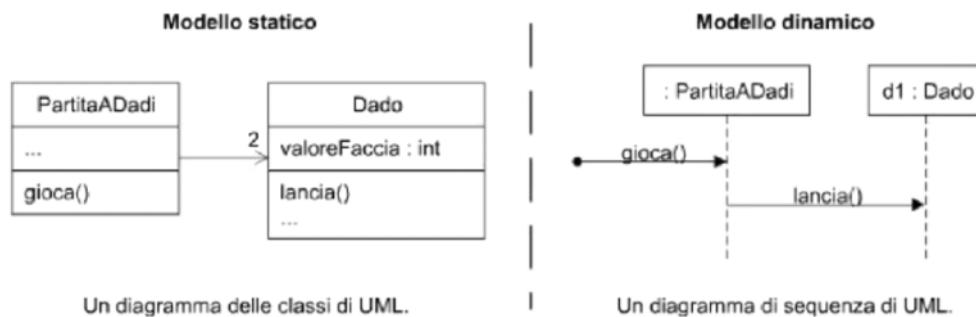
Come si progettano gli oggetti?

- **Codifica:** progettare mentre si codifica
- **Disegno, poi codifica:** disegnare alcuni diagrammi UML, poi si passa alla codifica.  
Il costo aggiuntivo dovuto al disegno dovrebbe ripagare lo sforzo impiegato
- **Solo disegno:** lo strumento genera ogni cosa dai diagrammi

Ci sono due tipi di modelli per gli oggetti:

- **Dinamici:** rappresentano il comportamento del sistema, la collaborazione tra oggetti software per realizzare (uno o più scenari di) un caso d'uso. Sono i metodi di classi software. La modellazione ad oggetti dinamica più comune è quella con i diagrammi di sequenza di UML.
- **Statici:** servono per definire i package, i nomi delle classi, gli attributi, le firme di operazioni. La modellazione a oggetti statica più comune è quella con i diagrammi delle classi di UML.

Conviene creare i due modelli in parallelo, visto che sono co-relazionati.



I messaggi nel diagramma di sequenza indicano operazioni nelle classi che ricevono il messaggio del diagramma delle classi.

Le linee di vita nel diagramma di sequenza rappresentano oggetti di classi del diagramma delle classi.

La maggior parte del lavoro di progettazione difficile, interessante e utile avviene mentre si disegnano i diagrammi di interazione, che rappresentano una vista dinamica.

Durante la modellazione a oggetti dinamica si pensa in modo dettagliato e preciso a quali oggetti devono esistere e come questi collaborano attraverso messaggi e metodi.

È durante la modellazione dinamica che si applica la progettazione guidata dalle **responsabilità** e i principi **GRASP** (principi di assegnazione di responsabilità).

# Modellazione UML

martedì 14 aprile 2020 14:42

Questi appunti sono condivisi da Stefano Vittorio Porta (Stefa168) con licenza <https://creativecommons.org/licenses/by-nc-nd/4.0/>.

UML comprende i **diagrammi di interazione** per illustrare il modo in cui gli oggetti interagiscono attraverso lo scambio di messaggi.

I diagrammi di interazione sono utilizzati per la modellazione **dinamica** degli oggetti.

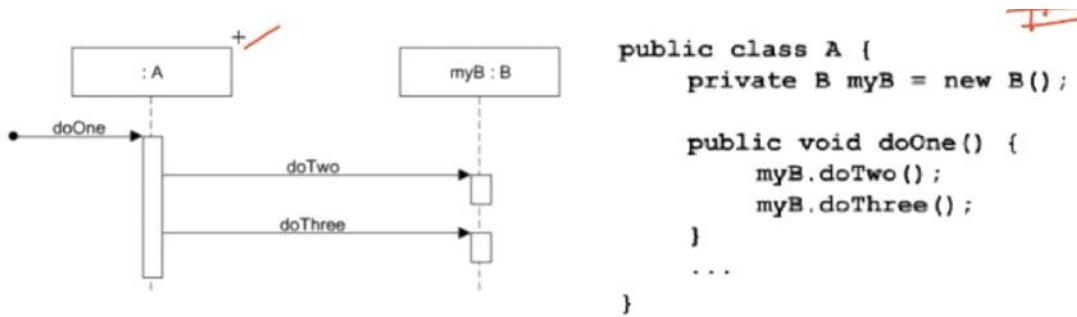
**Interazione:** è una specifica di come alcuni oggetti si scambiano messaggi nel tempo per eseguire un compito nell'ambito di un certo contesto.

Il termine **diagramma di interazione** è una generalizzazione dei tipi più specifici di diagrammi UML di **sequenza** e di **comunicazione**.

1. **Un'interazione** è motivata dalla necessità di eseguire un determinato **compito**.
2. Un compito è rappresentato da un **messaggio** che dà inizio all'interazione (messaggio trovato)
3. Il **messaggio** è inviato a un oggetto designato come **responsabile** per questo compito.
4. L'**oggetto responsabile** collabora/interagisce con altri oggetti (**partecipanti**) per svolgere il compito
5. Ciascun **partecipante** svolge un proprio **ruolo** nell'ambito della **collaborazione**.
6. La **collaborazione** avviene mediante **scambio di messaggi (interazione)**
7. Ciascun **messaggio** è una richiesta che un oggetto fa a un altro oggetto di eseguire un'**operazione**.

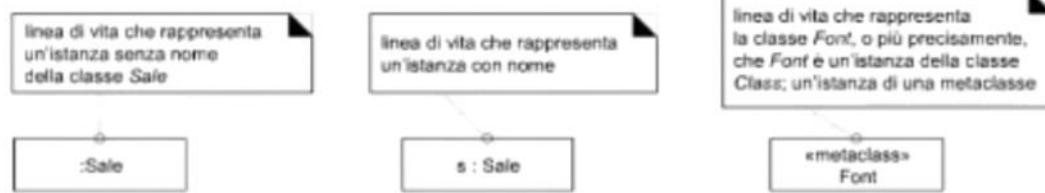
I diagrammi di sequenza mostrano le interazioni in una specie di formato a steccato, in cui gli oggetti che partecipano all'interazione sono mostrati in alto, a fianco.

- Vantaggi: mostrano chiaramente la sequenza dell'ordinamento temporale dei messaggi
- Svantaggi: costringono a estendersi verso destra quando si aggiungono nuovi oggetti



## Design Sequence Diagram

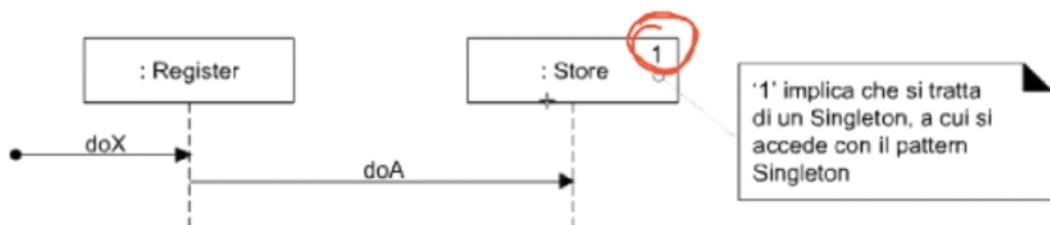
Il DSD di progetto è un diagramma di sequenza utilizzato da un punto di vista software o di progetto. In UP, l'insieme di tutti i DSD fa parte del **Modello di Progetto** che comprende anche i diagrammi delle classi.



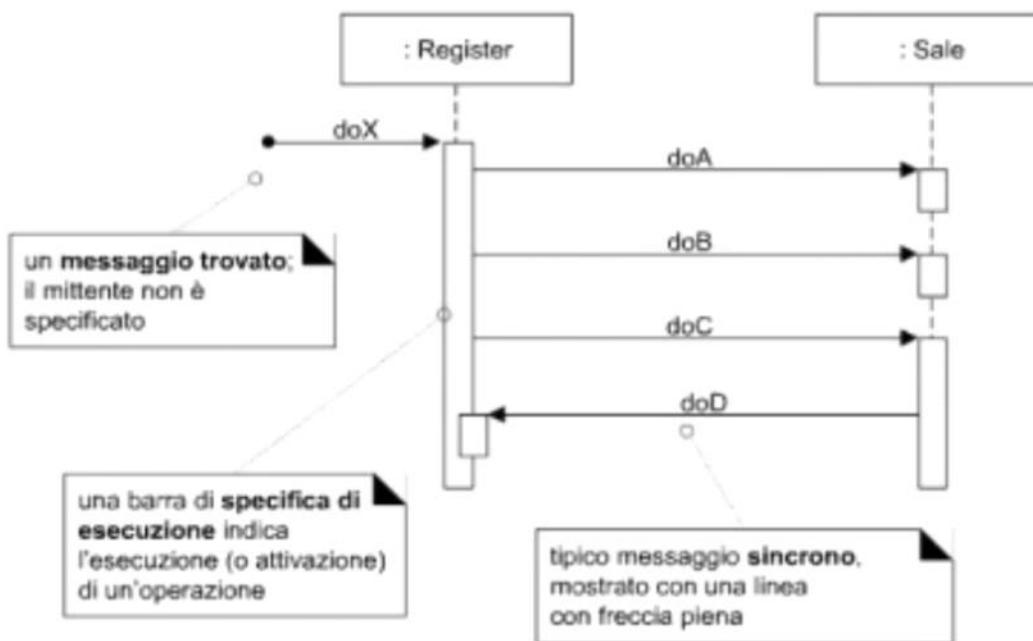
I rettangoli sono chiamati **linee di vita** (lifeline). Rappresentano i **partecipanti** all'interazione, ovvero le parti correlate definite nel contesto di un qualche diagramma strutturale.

## Singleton

Quando vogliamo rappresentare l'istanza **unica** di una certa classe, si usa l'1.



## Linee di vita e messaggi

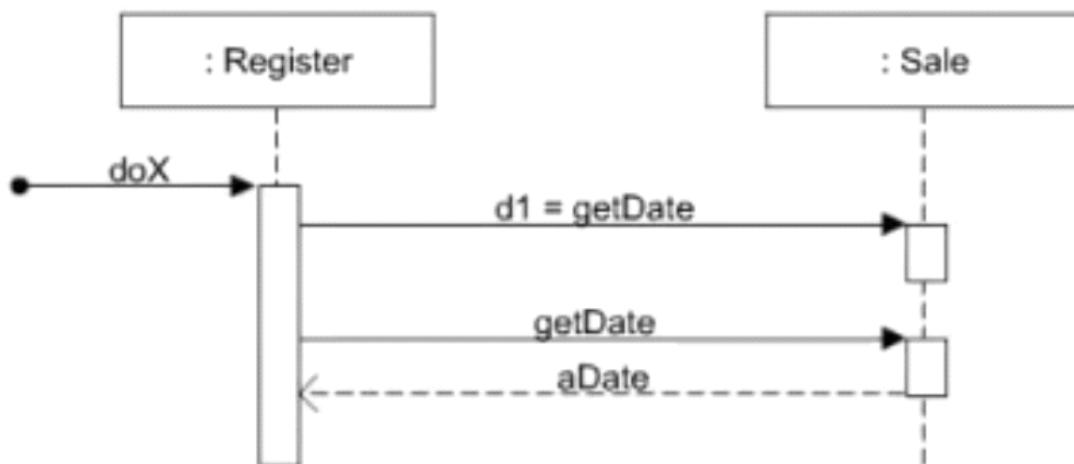


- Una **linea di vita** comprende sia un rettangolo che una linea verticale che si estende sotto di esso
- Ogni messaggio (di solito **sincrono**) tra gli oggetti è rappresentato da un'espressione messaggio mostrata su una linea continua con una freccia piena tra le linee di vita verticali
- Il messaggio iniziale è chiamato **messaggio trovato**
- Una barra di **specifiche di esecuzione** (chiamata anche **barra di attivazione** o **attivazione**) mostra l'esecuzione di un'operazione da parte di un oggetto

## Risposte o ritorni

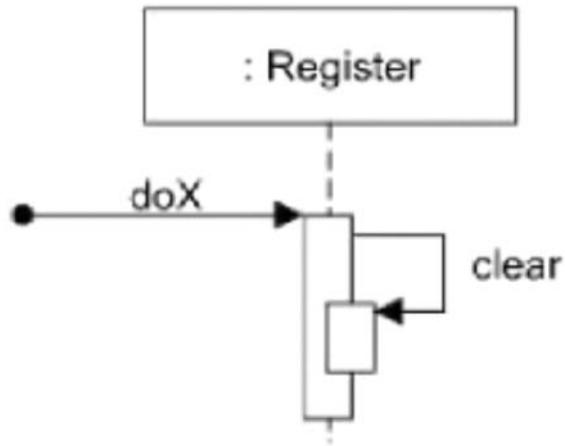
Ci sono due modi per mostrare il risultato di ritorno:

- Utilizzando la sintassi `returnVar = message(parametri)`
- Utilizzando una linea di messaggio di risposta (o ritorno) alla fine della barra di specifica di esecuzione



## Self o this

È possibile mostrare un messaggio inviato da un oggetto a sé stesso utilizzando una barra di specifica di esecuzione annidata

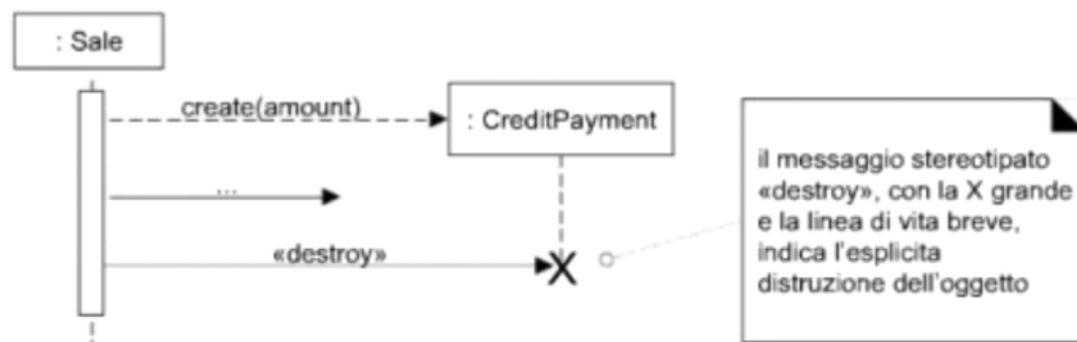


### Creazione di istanze

È possibile mostrare un messaggio inviato da un oggetto a sé stesso utilizzando una barra di specifica di esecuzione annidata. Si noti che i nuovi oggetti sono posizionati all'"altezza" della loro creazione.

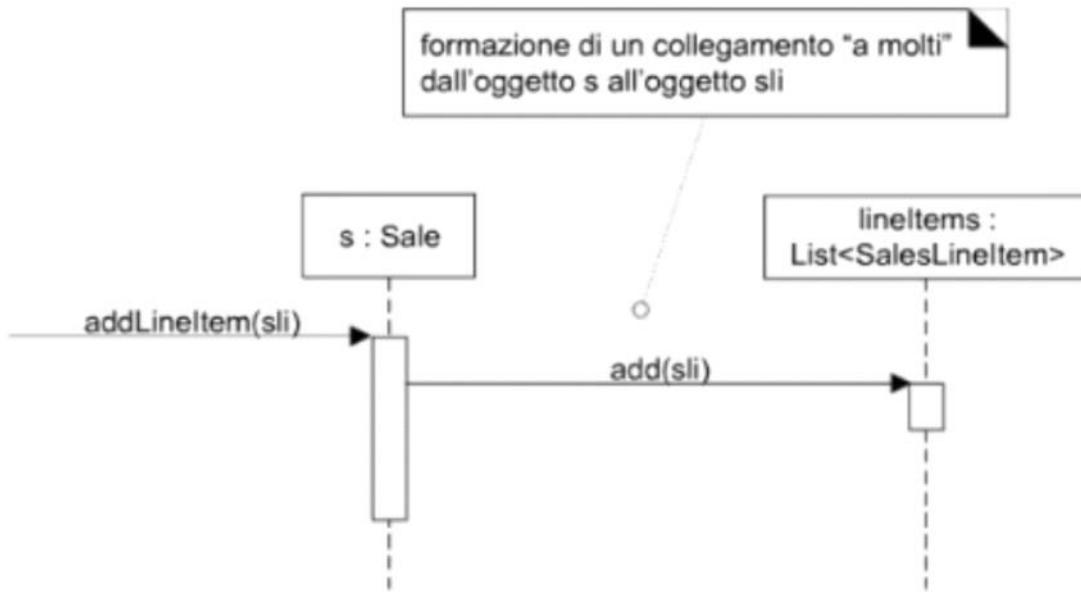


### Distruzione esplicita di oggetti



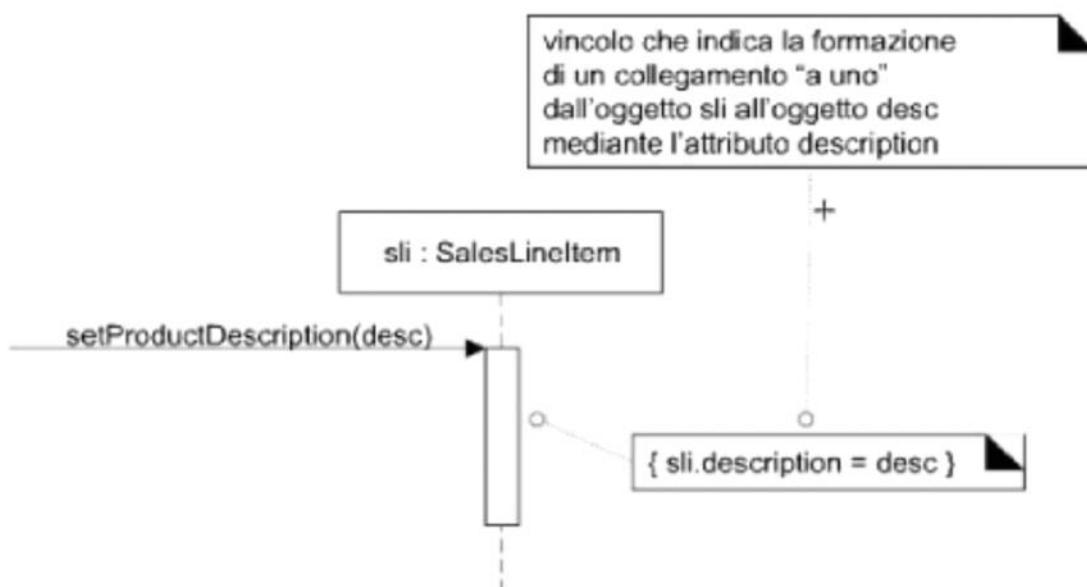
### Formazione di collegamenti

A molti



### A uno

Il vincolo va interpretato come una post-condizione dell'operazione, ovvero una condizione che deve risultare vera al termine della sua esecuzione

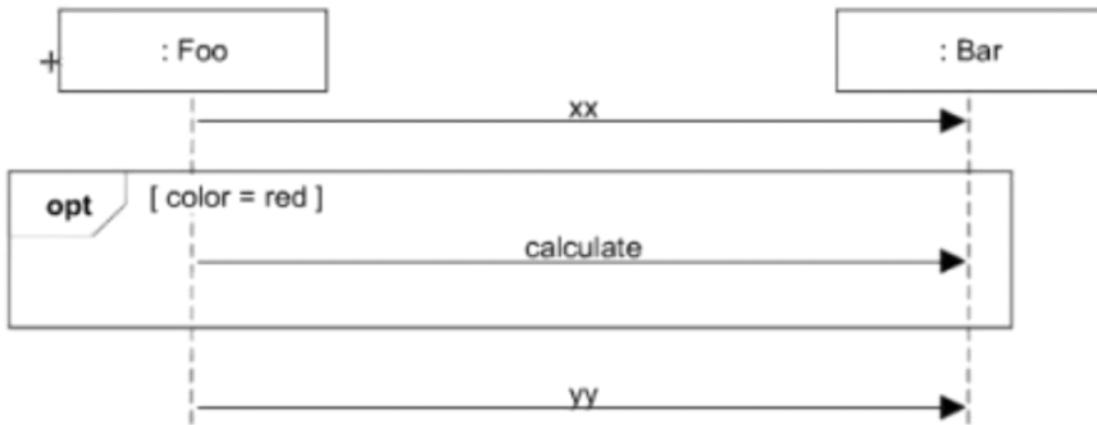


### Frame

Come supporto alle "istruzioni" di controllo condizionali e di ciclo si utilizzano i **frame**, regioni o frammenti dei diagrammi che hanno un operatore (etichetta) e una guardia (condizione o test booleano) che va posizionata sopra la linea di vita che deve valutare tale condizione.

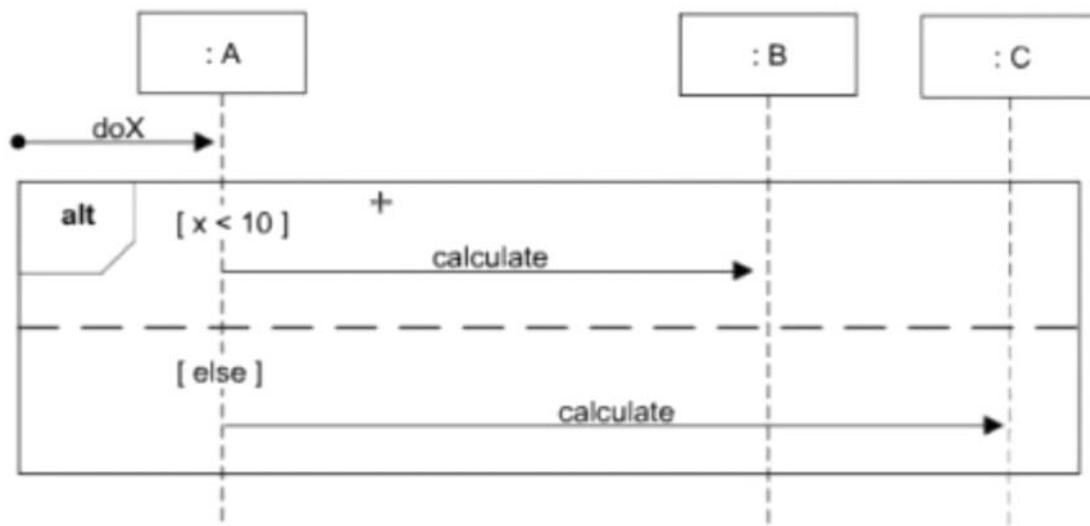
### Messaggi condizionali

Un frame **OPT** è posizionato attorno a uno o più messaggi



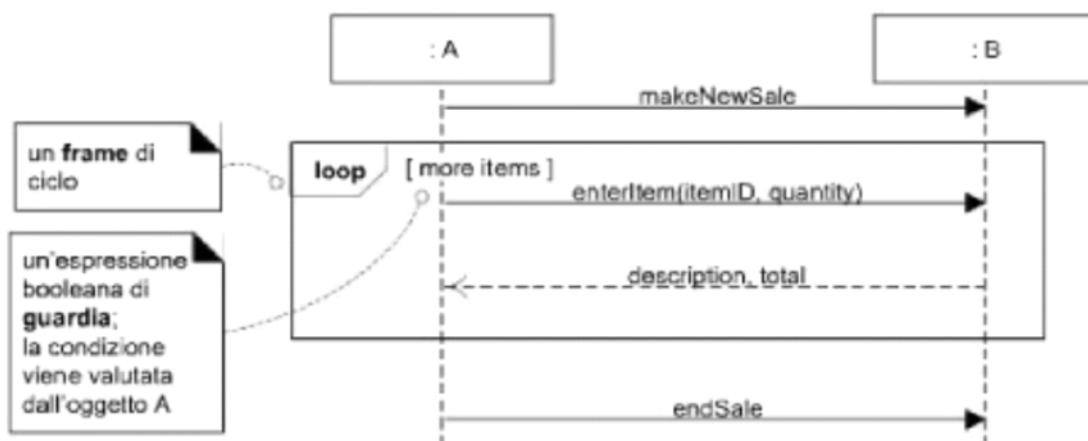
### Messaggi condizionali mutualmente esclusivi

Un frame **ALT** è posizionato attorno alle alternative mutuamente esclusive



### Iterazione

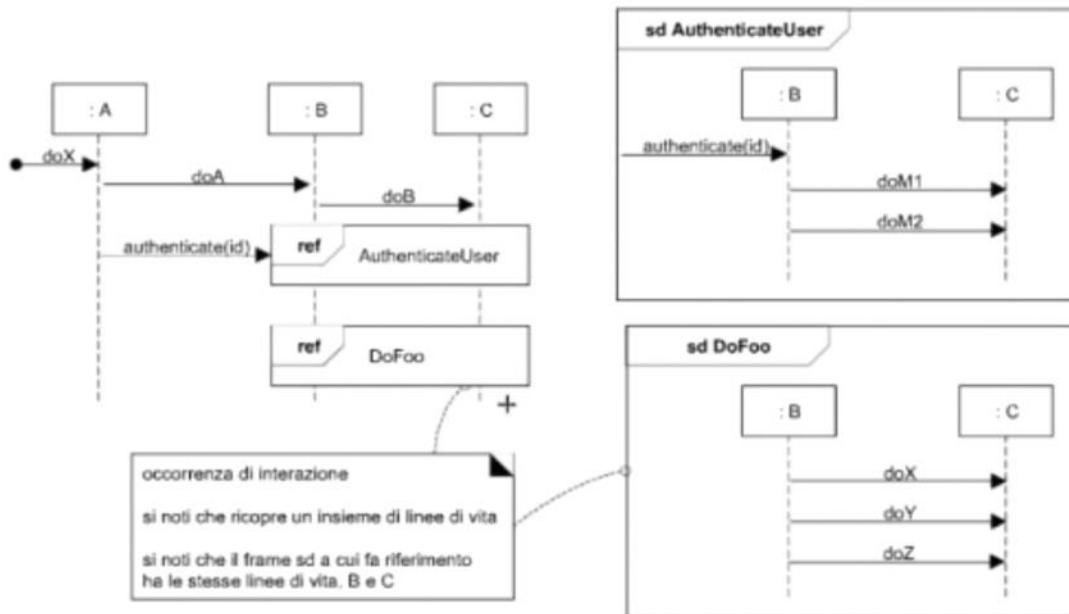
Un frame **LOOP** è posizionato attorno a uno o più messaggi da iterare



Si ricorda che i frame possono essere annidati

### Correlare diagrammi di interazione

Una occorrenza di interazione (o uso di interazione) è un riferimento a un'interazione all'interno di un'altra interazione che permette di correlare e collegare i relativi diagrammi

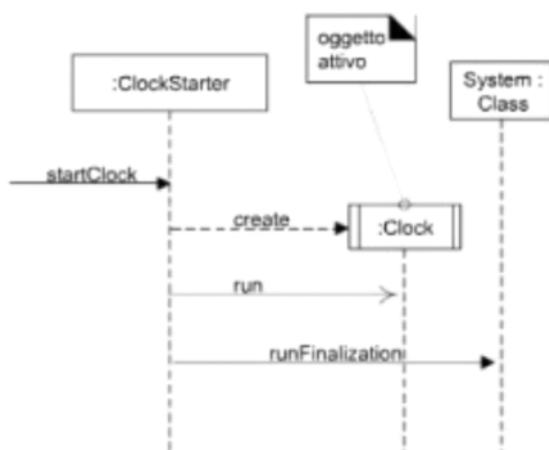
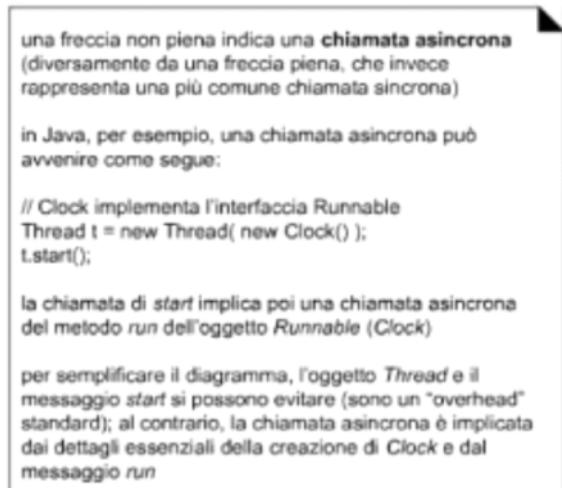


### Invocare metodi statici

L'oggetto ricevente è una classe o, più precisamente, un'istanza di una **meta-classe**.



### Chiamate sincrone e asincrone



## Diagrammi delle classi

UML comprende i **diagrammi delle classi** per illustrare classi, interfacce e le relative associazioni

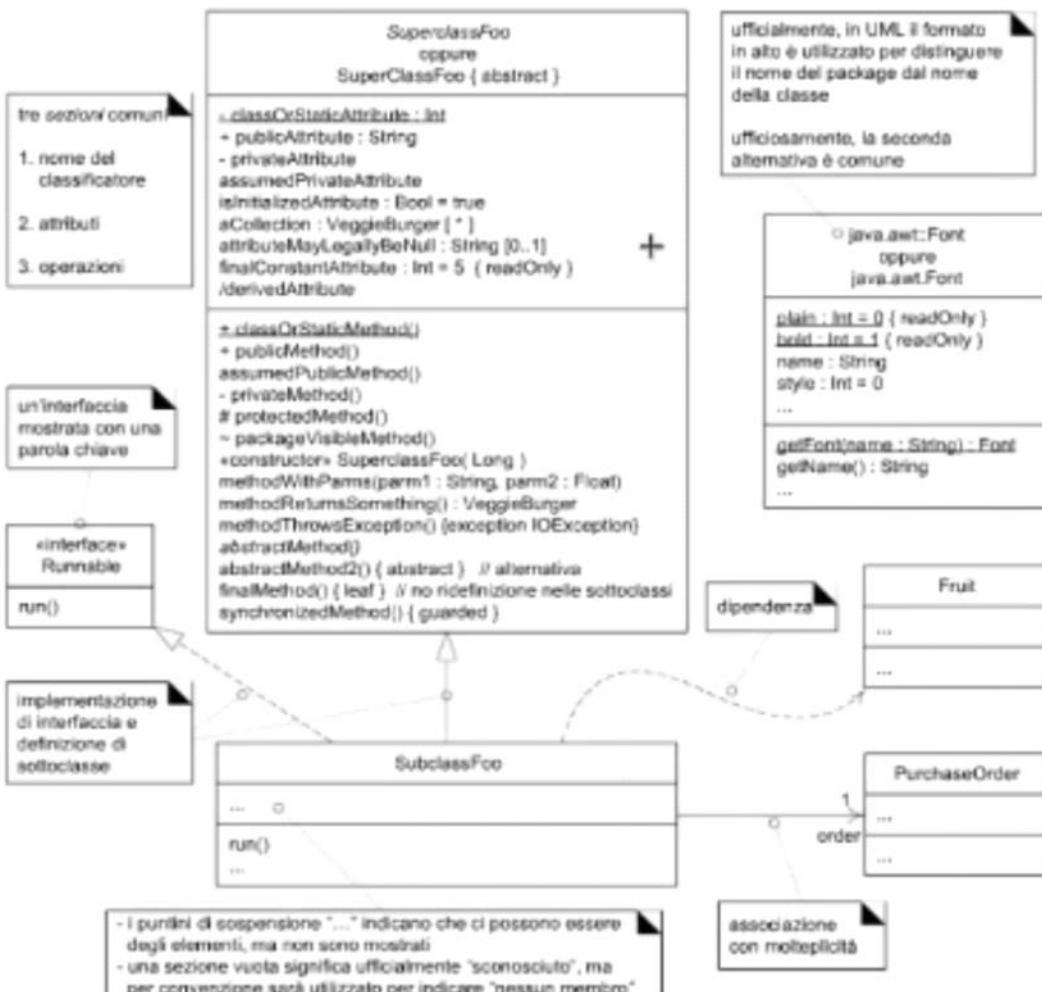
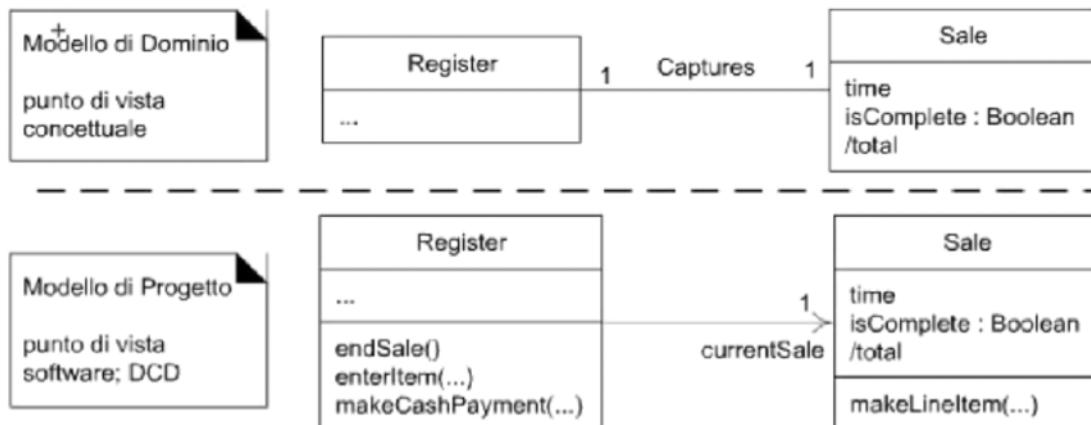
I diagrammi delle classi sono utilizzati per la **modellazione statica degli oggetti**.

I diagrammi delle classi sono stati utilizzati, da un punto di vista concettuale, per visualizzare un

modello di dominio.

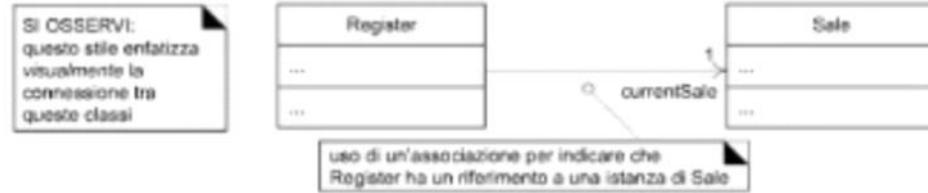
**Design Class Diagram:** il DCD è un diagramma delle classi utilizzato da un punto di vista software o di progetto.

In UP, l'insieme di tutti i DCD fa parte del **Modello di Progetto** che comprende anche i diagrammi di interazione.



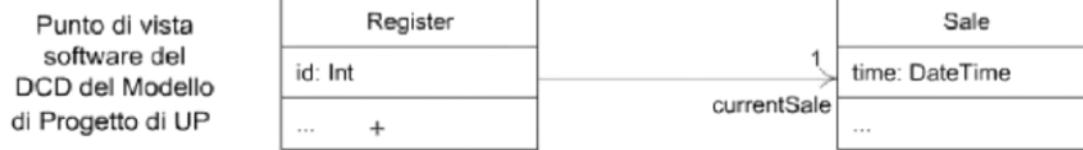
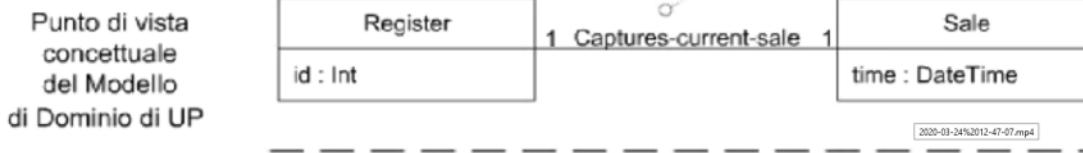
## Notazioni per attributi come associazioni

- Notazione testuale per un attributo (in alto)
- Notazione con una linea di associazione (in mezzo)
- Entrambe le annotazioni, insieme (in basso). **Non consigliata**



- Se non viene indicata **alcuna visibilità**, solitamente si ipotizza che gli attributi siano privati
- Una freccia di navigabilità rivolta dalla classe sorgente alla classe destinazione dell'associazione indica che un oggetto della classe sorgente ha un attributo di tipo della classe destinazione. È presente una molteplicità all'estremità vicina alla destinazione, ma non all'estremità vicina alla sorgente (in genere).
- È presente un nome di ruolo solo all'estremità vicina alla destinazione, per indicare il nome dell'attributo
- Non si ha alcun nome associato all'associazione.

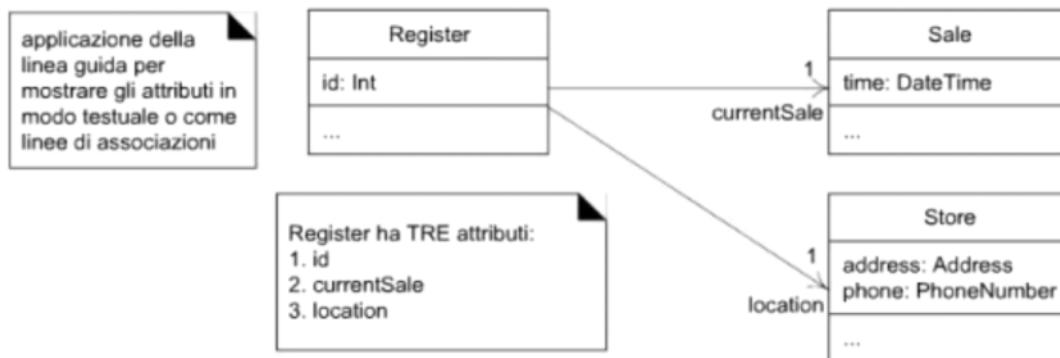
il *nome* dell'associazione, normale quando si disegna un modello di dominio, è spesso escluso (anche se legale) quando si utilizzano i diagrammi delle classi da un punto di vista software in un DCD



```

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

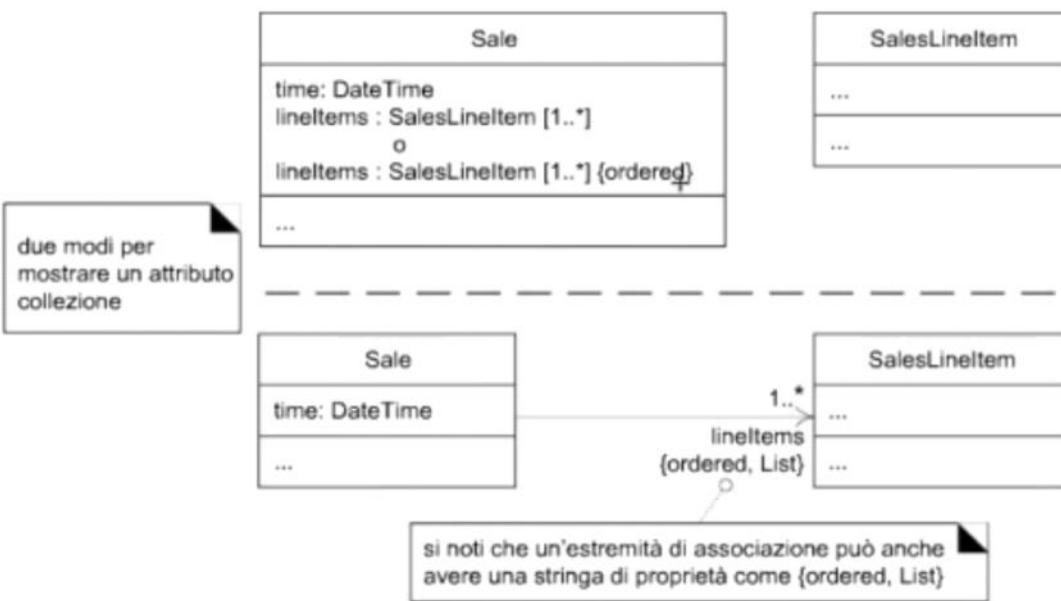
```



```

public class Sale {
    private List<SalesLineItem> lineItems = new ArrayList<>();
    ...
}

```



(preferibile la seconda notazione perché più leggibile).

## Operazioni e metodi

- Un'operazione è una dichiarazione di un metodo:  
`visibility name (parameter-list) : return-type { property-string }`
- Per default le operazioni hanno visibilità pubblica
- Nei diagrammi di classe vengono solitamente indicate le operazioni (signature - nomi e parametri)
- Nei diagrammi di interazione vengono modellati i metodi, come sequenze di messaggi

```

«method»
// vanno bene sia un linguaggio specifico che pseudo-codice
public void enterItem( id, qty ) {
    ProductDescription desc = catalog.getProductDescription(id);
    currentSale.makeLineItem(desc, qty);
}

```

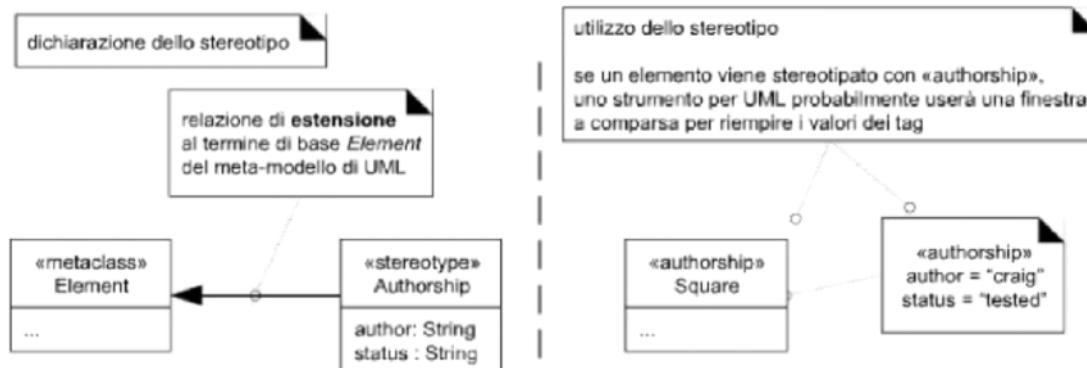
Register
...
endSale()
enterItem(id, qty)
makeNewSale()
makeCashPayment(cashTendered)

## Parole chiave

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

## Stereotipi

Gli stereotipi rappresentano un raffinamento di un concetto di modellazione esistente, ed è definito all'interno di un profilo UML (un profilo è una collezione di stereotipi)



## Generalizzazioni

- Si tratta di **relazioni tassonomiche** 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.
- La generalizzazione implica l'**ereditarietà** nei linguaggi OO
- Uso del tag {abstract} per le **classi astratte**.

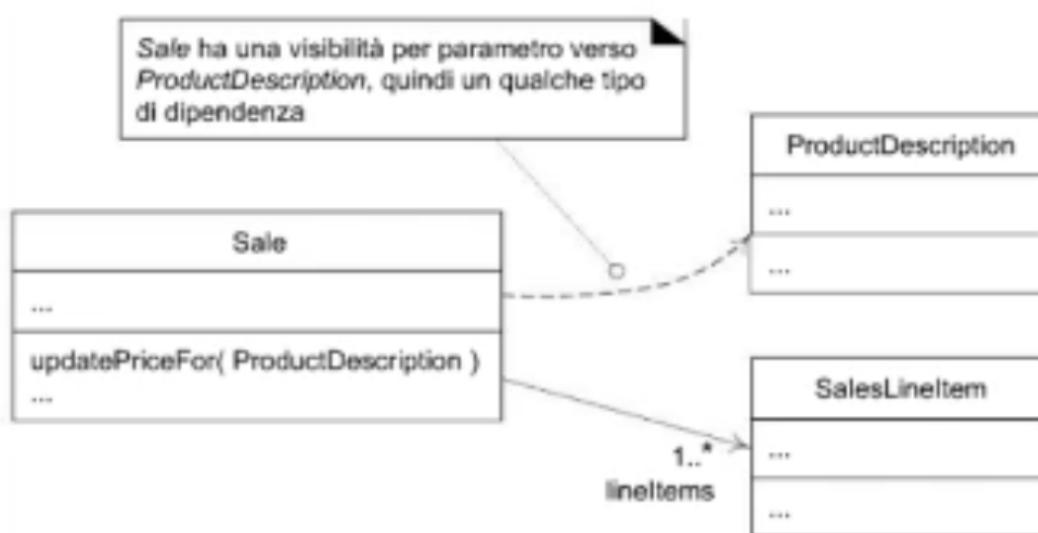
## Dipendenze

- Le linee di dipendenza sono comuni nei diagrammi delle classi e dei package.
- Una relazione di dipendenza indica che un elemento **cliente** è a conoscenza di un altro elemento **fornitore** e che un cambiamento nel fornitore potrebbe influire sul cliente (è l'**accoppiamento**)

```

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

```



Esistono molte tipologie di dipendenze:

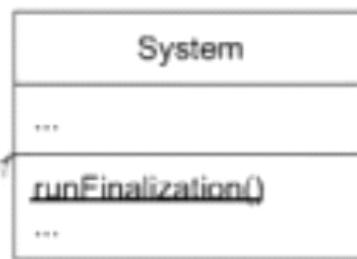
- Avere un attributo del tipo fornitore
- Inviare un messaggio a un fornitore; la visibilità verso il fornitore potrebbe essere data da:
  - Un attributo
  - Una variabile parametro
  - Una variabile locale
  - Una variabile globale
  - Una visibilità di classe (chiamata di metodi statici o di classe)
- Ricevere un parametro del tipo del fornitore
- Il fornitore è una superclasse o un'interfaccia implementata

```

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

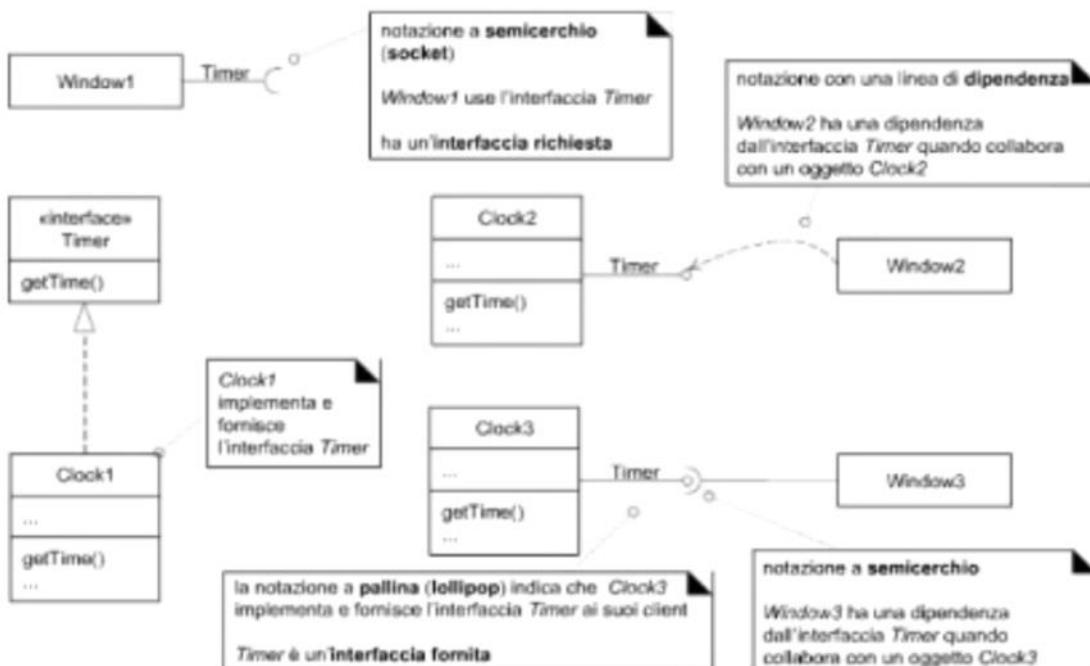
```

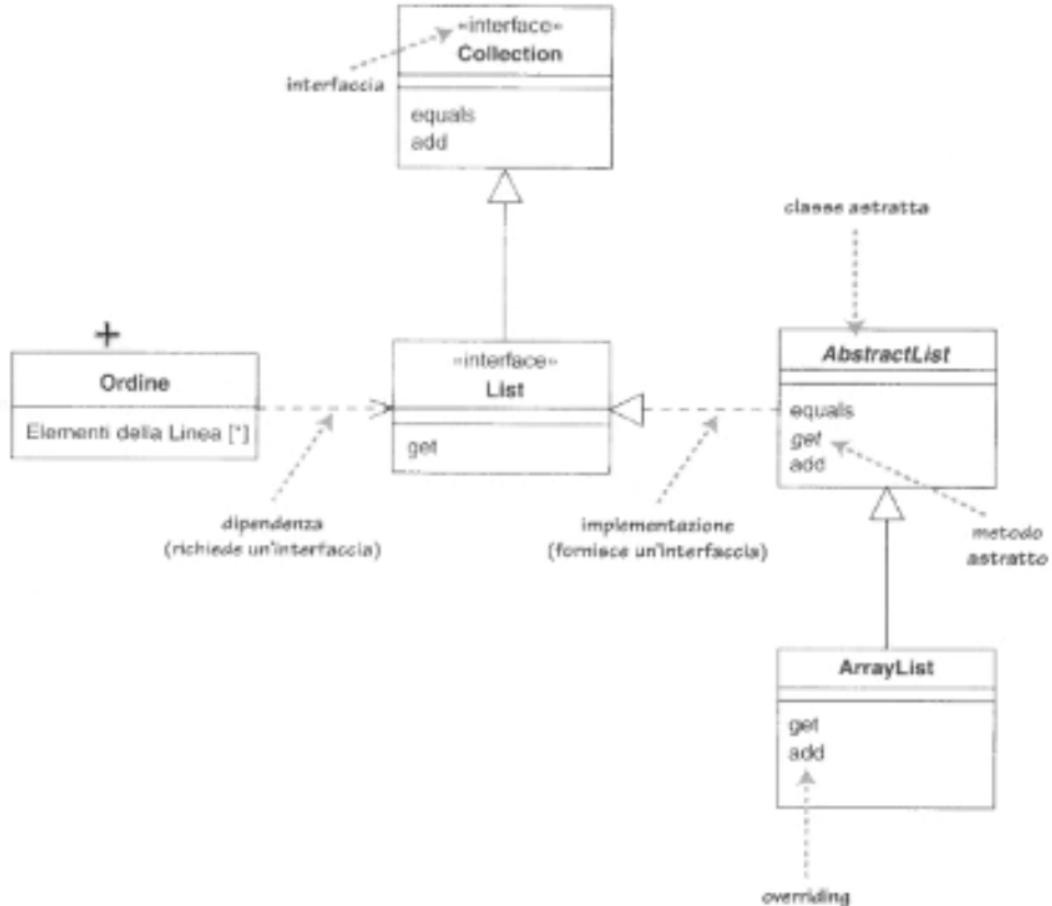
il metodo `doX` richiama il metodo statico `runFinalization`, quindi ha una dipendenza dalla classe `System`



### Dipendenza: Interfacce

- L'implementazione di un'interfaccia viene chiamata una **realizzazione di interfaccia**
- La notazione a pallina (**lollipop**) indica che una classe X implementa (fornisce) un'interfaccia Y, senza disegnare il rettangolo per l'interfaccia Y
- La notazione a semicerchio (**socket**) indica che una classe X richiede (usa) un'interfaccia Y, senza disegnare una linea che punta all'interfaccia Y

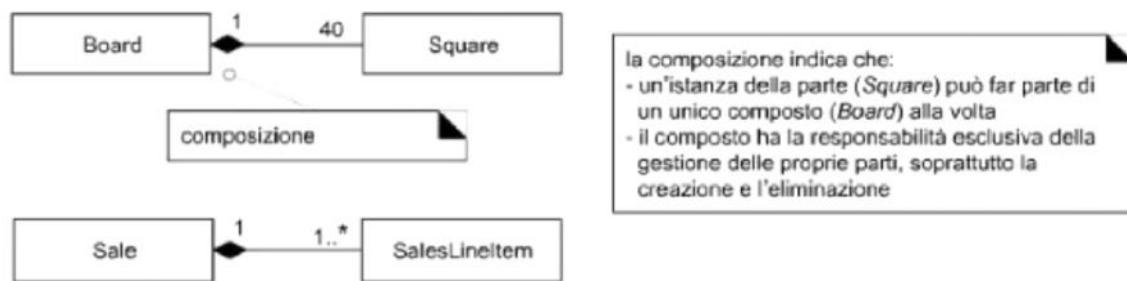




## Composizione

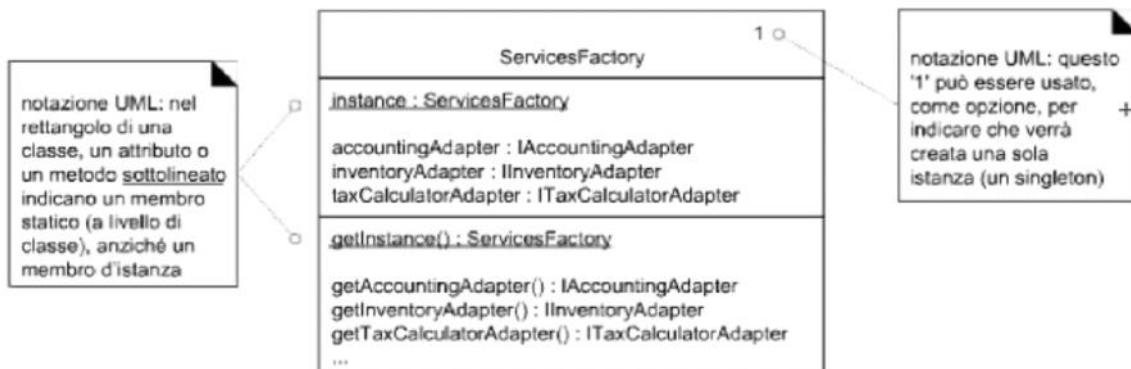
Una possibile interpretazione (dal punto di vista software) di una composizione tra le classi A e B è la seguente:

- Gli oggetti B non possono esistere indipendentemente da un oggetto A
- L'oggetto A è responsabile della creazione e distruzione dei suoi oggetti B



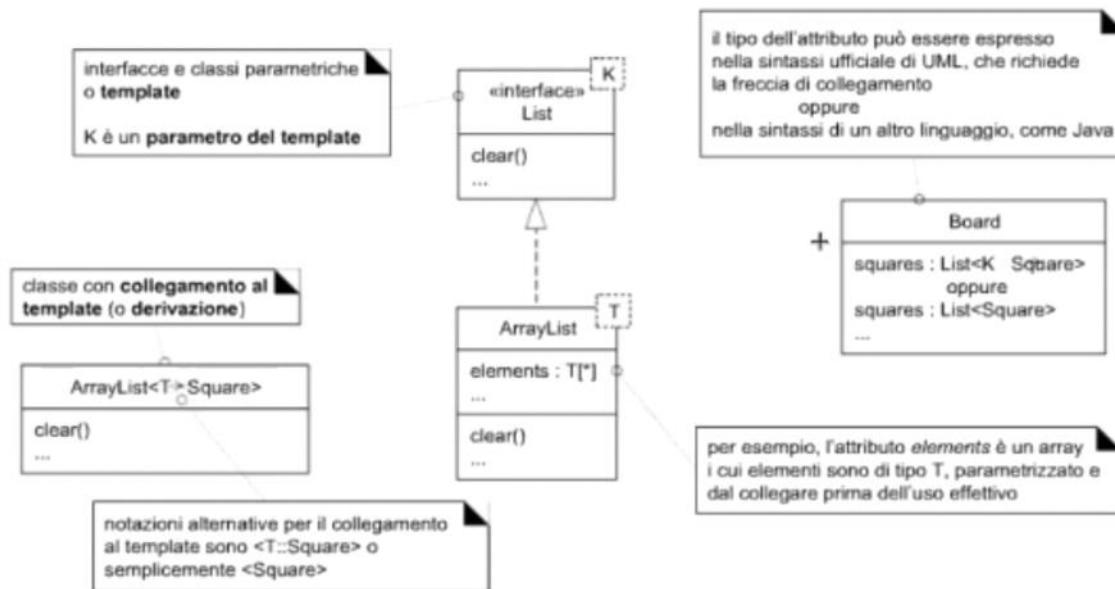
## Singleton

Esiste una sola istanza di una classe, mai due



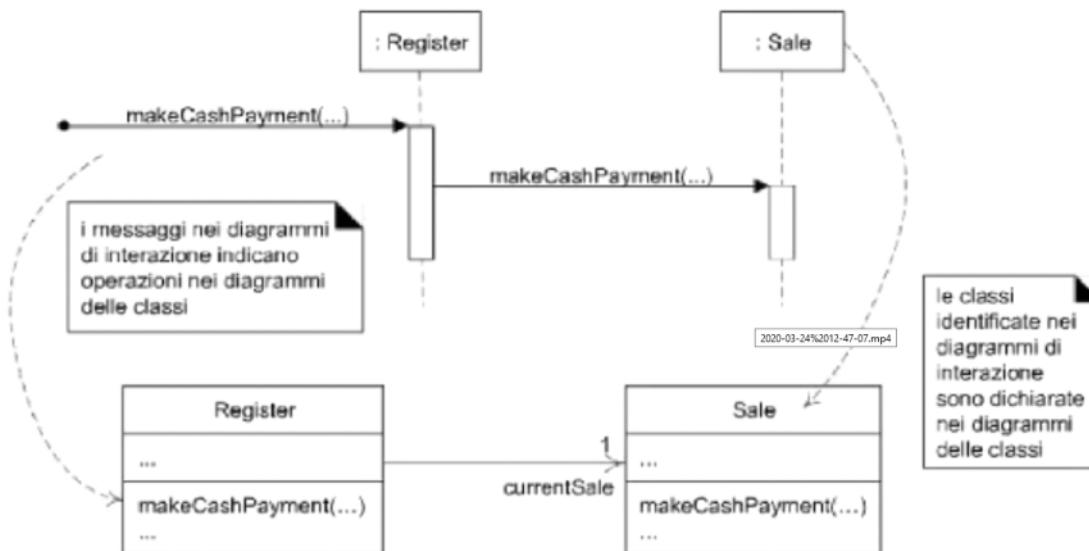
## Template

Molti linguaggi supportano **tipi a template**, noti anche come **template**, **tipi parametrizzati** e **generici**.



## Diagrammi delle classi e diagrammi di interazione/sequenza

L'influenza dei diagrammi di interazione sui diagrammi delle classi:



# GRASP

martedì 21 aprile 2020 15:46

## General Responsibility Assignment Software Patterns

### Responsabilità

Finora abbiamo

- Identificato i requisiti
- Creato il modello di dominio

Ora bisogna

- Aggiungere i metodi alle classi appropriate
- Definire i messaggi fra gli oggetti per soddisfare i requisiti

Capire le responsabilità è fondamentale per una buona programmazione ad oggetti.

Siamo ancora nella fase di Progettazione

Il decidere quali metodi appartengono a chi e come devono interagire gli oggetti ha delle conseguenze e pertanto queste scelte devono essere fatte con attenzione.

La padronanza dell'OOD coinvolge un insieme ampio di principi flessibili (e pattern), con molti gradi di libertà.

UML è semplicemente un linguaggio di modellazione visuale standard, la conoscenza dei dettagli non insegna come "pensare a oggetti"

La cosa più importante è che, durante le attività di disegno e codifica, vengano applicati 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 ad assegnare le responsabilità degli oggetti che collaborano.

Durante il disegno UML va adottato l'atteggiamento realistico (modellazione agile) secondo cui si disegnano i modelli soprattutto allo scopo di comprendere e comunicare, non di documentare.

**Ricordiamo che siamo in UP:** presto si deve interrompere la modellazione e si deve passare a programmare, per evitare una mentalità a cascata che porterebbe a un'eccessiva modellazione prima della programmazione (iterazioni timeboxed).

I risultati della progettazione ad oggetti saranno (in particolare dopo la prima iterazione):

- Diagrammi UML di interazione, delle classi e dei package, per le parti più difficili che è opportuno esaminare prima della codifica
- Abbozzi e prototipi dell'interfaccia utente
- Modelli della base di dati

Un modo comune di pensare alla progettazione di oggetti software è in termini di **responsabilità, ruoli e collaborazioni: progettazione guidata dalle responsabilità o RDD**:

Gli oggetti software sono considerati come dotati di responsabilità. Per responsabilità si intende un'astrazione di ciò che fa o rappresenta un oggetto o una componente software.

### RDD

In UML la responsabilità è un **contratto** o un **obbligo** di un classificatore.

Le responsabilità sono correlate agli obblighi o al **comportamento** di un oggetto in relazione al suo

ruolo. Sono fondamentalmente di due tipi:

- **Di fare:**
  - Fare qualcosa esso stesso, come creare un oggetto o eseguire un calcolo
  - Chiedere ad altri oggetti di eseguire azioni
  - Controllare e coordinare le attività di altri oggetti
- **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 ad oggetti. La traduzione delle responsabilità in classi e metodi è influenzata dalla *granularità* delle responsabilità. Le responsabilità più grandi coinvolgono centinaia di classi e metodi, mentre le responsabilità minori possono coinvolgere un solo metodo.

Nel software non c'è niente che corrisponde direttamente ad una responsabilità. Tuttavia, nel software vengono definite classi, metodi e variabili con lo scopo di **soddisfare** responsabilità. Le responsabilità sono implementate per mezzo degli oggetti e metodi che agiscono da soli oppure che **collaborano** con altri oggetti e metodi.

## Passi della RDD

La progettazione guidata dalle responsabilità viene fatta iterativamente:

1. Identifica le responsabilità e considerale una alla volta
2. Chiediti a quale oggetto software assegnare questa responsabilità. Potrebbe essere un oggetto tra quelli già identificati, oppure un nuovo oggetto
3. Chiediti come fa l'oggetto scelto a soddisfare questa responsabilità. Potrebbe fare tutto da solo, oppure potrebbe collaborare con altri oggetti (l'identificazione di una collaborazione spesso porta ad identificare nuove responsabilità da assegnare)

Il procedimento va basato su opportuni criteri per l'assegnazione di responsabilità, come i pattern GRASP.

## GRASP

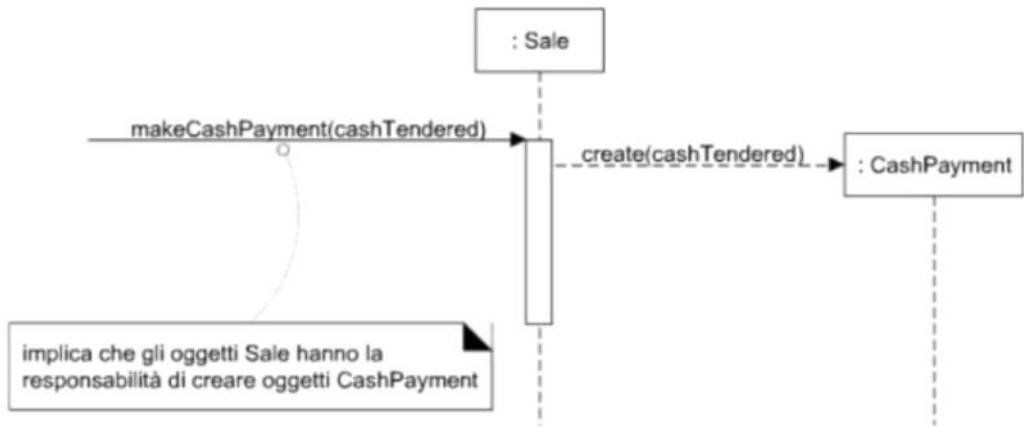
### Come aiuto alla comprensione

I pattern GRASP, per l'**assegnamento di responsabilità**, rappresentano solo un aiuto per apprendere la struttura e dare un nome ai principi

In altre parole, i principi/pattern **GRASP** sono un aiuto per l'apprendimento degli aspetti essenziali della progettazione ad oggetti e per l'applicazione dei ragionamenti di progettazione in un modo metodico, razionale e spiegabile.

Le decisioni sull'assegnazione delle responsabilità agli oggetti possono essere prese mentre si esegue la **codifica** oppure durante la **modellazione**

Il disegnare i diagrammi di interazione diventa l'occasione per considerare tali responsabilità (realizzate come metodi): le responsabilità e i metodi sono correlati.



## Cosa sono i pattern?

Un repertorio contenente sia i principi generali che soluzioni idiomatische che guidino gli sviluppatori nella creazione del software.

I principi e gli idomi 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 è una descrizione con nome di un problema di progettazione ricorrente e di una sua soluzione ben provata e che può essere applicata a nuovi contesti. Sono anche forniti consigli su come applicarlo in situazioni nuove e con una discussione sui relativi *compromessi, implementazioni, variazioni* e così via.

## Riepilogando

- **RDD** come metafora per la progettazione degli oggetti: una comunità di oggetti con responsabilità che collaborano.
- **Pattern** come modo per dare un nome e spiegare le idee della progettazione OO.
  - **GRASP** per i pattern di base dell'assegnazione di responsabilità
  - **GoF** per idee di progettazione più avanzate.
- I pattern possono essere applicati sia durante la modellazione, sia durante la codifica
- **UML** per la **modellazione visuale** per la progettazione OO, nel corso della quale possono essere applicati sia i pattern GRASP sia quelli GoF.

# Pattern GRASP

martedì 21 aprile 2020 16:26

## Obiettivi di GRASP

Un sistema software ben progettato è facile da **comprendere**, da **mantenere** e da **estendere**. Inoltre le scelte fatte consentono delle buone opportunità di **riusare** i suoi componenti software in applicazioni future.

## Obiettivi di GRASP e UP

Comprendere, manutenzione, estensione e riuso sono **qualità fondamentali** in un **contesto di sviluppo iterativo**, in cui il **software viene continuamente modificato**, estendendolo con nuove funzionalità (relative a nuove operazioni di sistema e a nuovi casi d'uso) oppure manutenendo le funzionalità implementate (per esempio, a fronte di cambiamenti nei requisiti). Comprensibilità e semplicità facilitano queste attività evolutive.

## Progettazione modulare

Comprensibilità, modificabilità, impatto nei cambiamenti basso, flessibilità, riuso, semplicità sono sostenute dal principio classico della **progettazione modulare**, secondo cui il software deve essere decomposto in un insieme di elementi software (**moduli**) **coesivi** e **debolmente accoppiati**.

In GRASP i principi della progettazione modulare sono rappresentati dai pattern **High Cohesion** e **Low Coupling**.

(Nota: questi due pattern sarebbero sufficienti per la corretta assegnazione di responsabilità nella maggior parte dei casi sostenendo le qualità indicate. In pratica però, questi due pattern sono difficili da applicare direttamente perché sono *pattern valutativi*)

## Pattern GRASP

### Creator

Nome	Creator (Creatore)
Problema	Chi crea un oggetto A? Ovvero, chi deve essere responsabile della creazione di una nuova istanza di una classe?
Soluzione	Assegna alla classe B la responsabilità di creare un'istanza della classe A se una delle seguenti condizioni è vera (e più sono vere, meglio è): <ul style="list-style-type: none"><li>• B "contiene" o aggrega con una composizione oggetti di tipo A</li><li>• B registra A (Dove per "registra" si intende il salvataggio di una reference di un oggetto di una classe ad un altro)</li><li>• B utilizza strettamente A</li><li>• 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)</li></ul>

### Osservazioni

- Trovare creatore che abbia veramente bisogno di essere collegato all'oggetto creato (low coupling)
- Si devono usare classi di supporto (ovvero pattern non GRASP come le Factory) se la creazione può essere in alternativa a "riciclo" o se una proprietà esterna condiziona la scelta della classe creatrice tra un insieme di classi simili
- Creator correlato a Low Coupling, Creator favorisce un accoppiamento basso, minori dipendenze di manutenzione e maggiori opportunità di riuso. La classe creata deve probabilmente essere già visibile alla classe creatore.

## Information Expert (o Expert)

Nome	Information Expert (esperto delle informazioni)
Problema	Qual è un principio di base, generale, per l'assegnazione di responsabilità agli oggetti?
Soluzione	Assegna una responsabilità alla classe che possiede le informazioni necessarie per soddisfarla all'esperto delle informazioni, ovvero alla classe che possiede le informazioni necessarie per soddisfare la responsabilità

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...

#### Osservazioni

- Si individuano informazioni parziali di cui classi diverse sono "esperte": queste classi **collaborano** insieme per realizzare l'obiettivo
  - Informazioni distribuite, classi più leggere, senza perderne l'incapsulamento
- "*Do it myself*": gli oggetti software, a differenza di quelli reali, hanno la responsabilità di compiere delle "azioni" sulle cose che conoscono.

#### Low Coupling

Nome	Low Coupling (Accoppiamento Basso)
Problema	Come ridurre l'impatto dei cambiamenti? Come sostenere una dipendenza bassa, un impatto dei cambiamenti basso e una maggiore opportunità di riuso?
Soluzione	Assegna le responsabilità in modo tale che l'accoppiamento (non necessario) rimanga basso. Usa questo principio per valutare le alternative.

Una classe con un accoppiamento alto (forte) dipende da molte altre classi.

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

- I cambiamenti nelle classi correlate, da cui queste classi dipendono, obbligano a cambiamento locali anche in queste classi
- Queste classi sono più difficili da comprendere in isolamento, ovvero senza comprendere anche le classi da cui dipendono
- Sono più difficili da riusare, poiché il loro uso richiede la presenza aggiuntiva delle classi da cui dipendono

#### Osservazioni

Le forme più comuni di accoppiamento da un tipo X a un tipo Y comprendono:

- La classe X ha un **attributo** (una variabile d'istanza o un dato membro) di tipo Y o **referenzia** un'istanza di tipo Y o una collezione di oggetti Y
- Un oggetto di tipo X **richiama operazioni o servizi** di un oggetto di tipo Y
- Un oggetto di tipo X **crea** un oggetto di tipo Y
- Il tipo X ha un metodo che **contiene** un elemento (parametro, variabile locale oppure tipo di ritorno) di tipo Y che **referenzia** un'istanza di tipo Y
- La classe X è una **sottoclasse** (diretta o indiretta) della classe Y
- Y è un'interfaccia, e la classe X implementa questa interfaccia

Le classi che sono per natura **generiche** e che hanno un'alta probabilità di **riuso** devono avere un **accoppiamento particolarmente basso**

Un certo grado moderato di accoppiamento tra le classi è normale, anzi è necessario per la creazione di un sistema orientato agli oggetti in cui i compiti vengono svolti grazie a una **collaborazione** tra oggetti connessi

Una sottoclass è fortemente accoppiata alla sua **superclasse**. Si consideri attentamente ogni decisione di **estendere** una superclasse, poiché è una forma di **accoppiamento forte**.

Porzioni di **codice duplicato** sono **fortemente accoppiate** tra di loro; infatti la modifica di una copia spesso implica la necessità di modificare anche le altre copie.

Un accoppiamento alto con elementi *stabili* o con elementi *pervasivi* costituisce raramente un problema che infatti non è l'accoppiamento alto di per sé, ma **l'accoppiamento alto con elementi per certi aspetti instabili**, per esempio nell'interfaccia, nell'implementazione o per loro pura e semplice presenza.

### Vantaggi

- Una classe o componente con un accoppiamento basso non è influenzata dai cambiamenti nelle altre classi e componenti
- È semplice da capire separatamente dalle altre classi e componenti
- È conveniente da riusare

## High Cohesion

Nome	High Cohesion (Coesione Alta)
Problema	Come mantenere gli oggetti focalizzati, comprensibili e gestibili e, come effetto collaterale, sostenere Low Coupling?
Soluzione	Assegna le responsabilità in modo tale che la coesione rimanga alta. Usa questo principio per valutare alternative.

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*.

Una classe con coesione bassa fa molte cose non correlate tra loro o svolge troppo lavoro.

Tali classi presentano i seguenti problemi:

- Sono difficili da comprendere
- Sono difficili da mantenere
- Sono difficili da riusare
- 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.

### Osservazioni

High Cohesion è un principio da tenere presente durante tutte le decisioni di progetto: è un obiettivo basilare da tenere continuamente in considerazione. È un **principio di valutazione** per scegliere tra diverse alternative.

La coesione è una misura di quanto sono correlate le responsabilità di un elemento software.

- **Coesione di dati**: una classe implementa un tipo di dati (molto buona)
- **Coesione funzionale**: gli elementi di una classe svolgono una singola funzione (buona o molto buona)
- **Coesione temporale**: gli elementi sono raggruppati perché usati circa nello stesso tempo (es controller, a volte buona, a volte meno)
- **Coesione per pura coincidenza**: es. una classe usata per raggruppare tutti i metodi il cui nome inizia per una certa lettera dell'alfabeto (molto cattiva)

La coesione è una misura di quanto sono correlate le responsabilità di un elemento software.

High Cohesion è la coesione funzionale, una misura di quanto sono correlate le responsabilità e le operazioni di un elemento software, una misura di quanto lavoro esegue un elemento software:

- Un elemento ha coesione alta se ha responsabilità (funzionali) altamente correlate e se "non svolge troppo lavoro"
- Un elemento ha coesione bassa se fa molte cose non correlate o se svolge troppo lavoro

La coesione è una misura di quanto sono correlate le responsabilità di un elemento software:

- **Coesione molto bassa:** una classe è la sola responsabile di molte cose in aree funzionali molto diverse
- **Coesione Bassa:** una classe ha da sola la responsabilità di un compito complesso in una sola area funzionale
- **Coesione Alta:** una classe ha responsabilità moderate in un'area funzionale e collabora con altre classi per svolgere i suoi compiti
- **Coesione Moderata:** una classe ha, da sola, responsabilità leggere in poche aree diverse, che sono logicamente correlate al concetto rappresentato dalla classe, ma non l'una dall'altra

**Regola pratica:** una classe con coesione alta ha un numero di metodi relativamente basso, con delle funzionalità altamente correlate e focalizzate. Inoltre, non fa troppo lavoro.

Essa collabora con altri oggetti per condividere lo sforzo, se il compito è grande.

In questo modo si sostiene la manutenzione, l'evoluzione e il riutilizzo.

In alcuni casi una coesione bassa è accettabile:

- Si vuole mantenere in una sola classe l'insieme delle query sql di un sistema (un esperto)
- Per gli oggetti lato server per avere maggiori prestazioni

### Vantaggi

- Maggiore chiarezza e facilità di comprensione del progetto
- Spesso sostiene low coupling
- La manutenzione e i miglioramenti risultano semplificati
- Maggiore riuso di funzionalità a grana fine e altamente correlate, poiché una classe se coesa può essere usata per uno scopo molto specifico

### Controller

Nome	Controller (Controllore)
Problema	Qual è il primo oggetto oltre lo strato UI che riceve e coordina ("controlla") un'operazione di sistema?
Soluzione	Assegna la responsabilità a un oggetto che rappresenta una delle seguenti scelte: <ul style="list-style-type: none"><li>• Rappresenta il "sistema" complessivo, un "oggetto radice", un dispositivo all'interno del quale viene eseguito il software, un punto di accesso al software o un sottosistema principale (variante del facade controller)</li><li>• Rappresenta uno scenario di un caso d'uso all'interno del quale si verifica l'operazione di sistema (un controller di caso d'uso o controller di sessione)</li></ul>

Nel caso di controller di caso d'uso o controller di sessione:

- Si utilizzi la stessa classe controller per tutti gli eventi di sistema nello stesso scenario di caso d'uso
- Una sessione è un'istanza di una conversazione con un attore.  
Le sessioni possono avere una lunghezza qualsiasi, ma spesso una sessione corrisponde a un'esecuzione di un caso d'uso

Note:

1. Le classi dello strato UI non sono Controller, non devono svolgere i compiti associati agli eventi di sistema, ma normalmente ricevono questi eventi e li delegano ad un controller
2. I controller sono classi che ricevono/gestiscono i messaggi legati alle operazioni di sistema (coordinano, ma non fanno molto di più)
3. Posso controllare che gli eventi avvengano in un ordine prestabilito

### Osservazioni

Controller è semplicemente un pattern di **delega**:

- Gli oggetti dello strato UI catturano gli eventi di sistema generati dagli attori
- Gli oggetti dello strato UI devono delegare le richieste di lavoro ad oggetti di un altro strato.

Il pattern Controller riassume le scelte fatte comunemente dagli sviluppatori OO quando questo "altro strato" è lo strato del dominio, in merito all'oggetto di dominio **delegato** che riceve le richieste di lavoro, che viene chiamato un oggetto **controller**.

Il controller è una sorta di "facciata" dello strato del dominio dallo strato UI. Consente di progettare gli oggetti di dominio in modo indipendente dagli oggetti dell'interfaccia utente che potrebbero interagire con essi.

**Importante:** normalmente un controller deve **delegare** ad altri oggetti il lavoro da eseguire durante l'operazione di sistema: il controller **coordina o controlla** le attività, ma **non esegue** di per sé molto lavoro.

Un problema comune deriva da un'**eccessiva** assegnazione di responsabilità. Un controller soffre di una coesione bassa, violando il principio di **High Cohesion**.

Usare la stessa classe controller può servire per **conservare le informazioni sullo stato del caso d'uso** e per identificare degli eventi fuori sequenza.

# Pattern GoF

venerdì 8 maggio 2020 18:31

Sono degli schemi di progettazione avanzata (a differenza dei principi GRASP).

Ciascun pattern descrive una soluzione progettuale comune a un problema di progettazione ricorrente.

Sono classificati in base allo scopo:

- Creazionale
- Strutturale
- Comportamentale

## Creazionali

Risolvono problematiche inerenti l'instanziazione degli oggetti:

- Abstract factory
- Builder
- Factory method
- Lazy initialization
- Prototype pattern
- Singleton
- Double-check locking

## Strutturali

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

## Comportamentali

- Chain of Responsibility
- Command
- Event Listener
- Hierarchical Visitor
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

Meglio la composizione rispetto all'ereditarietà

- Ereditarietà di classi:
  - Definiamo un oggetto in termini di un altro
  - Riuso *white-box*: la visibilità della superclasse è la visibilità della sottoclasse (la sottoclasse può accedere ai dettagli implementativi della superclasse)
- Composizione di oggetti
  - La funzionalità sono ottenute assemblando o componendo gli oggetti per avere funzionalità più complesse
  - Riuso *black-box*: i dettagli interni non sono conosciuti

Preferire la composizione rispetto all'ereditarietà tra classi perché aiuta a **mantenere le classi encapsulate e coese**.

La delegazione permette di rendere la composizione tanto potente quanto l'ereditarietà.

- Ereditarietà di classi:
  - Definita staticamente, non è possibile cambiarla a tempo di esecuzione: se una classe estende un'altra, questa relazione è definita nel codice sorgente, non può cambiare a runtime
  - Una modifica alla sopraclasse potrebbe avere ripercussioni indesiderate sul funzionamento di una classe che la estende (non rispetta l'incapsulamento)
- Composizione di oggetti :
  - Se una classe usa un'altra classe, questa potrebbe essere referenziata attraverso una interfaccia, a runtime potrebbe esserci una qualsiasi altra classe che implementa l'interfaccia
  - La composizione attraverso un'interfaccia rispetta l'incapsulamento, solo una modifica all'interfaccia comporterebbe ripercussioni

# Creazionali

venerdì 8 maggio 2020 18:49

## Abstract Factory

**Problema:** come creare famiglie di classi correlate che implementano un'interfaccia comune?

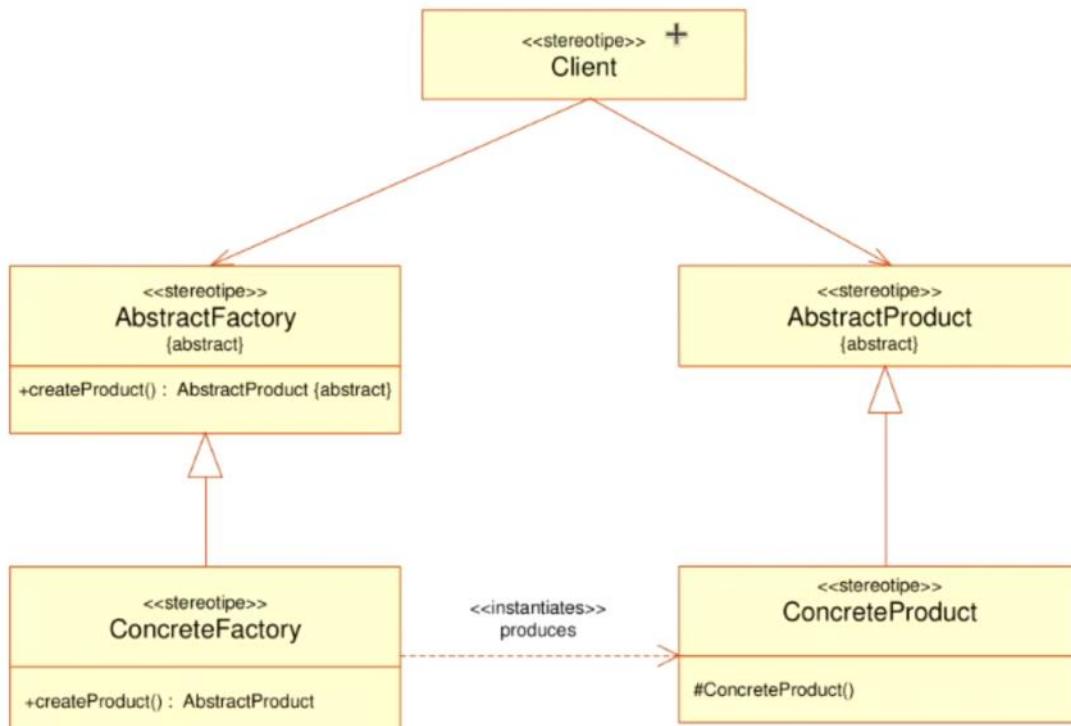
**Soluzione:** definire un'interfaccia factory (la factory astratta). Definire una classe factory concreta per ciascuna famiglia di elementi da creare. Opzionalmente, definire una vera classe astratta che implementa l'interfaccia factory e fornisce servizi comuni alle factory concrete che la estendono.

Presenta un'interfaccia per la creazione di famiglie di prodotti, in modo tale che il cliente che li utilizza non abbia conoscenza delle loro concrete classi. Questo consente:

- Assicurarsi che il cliente crei soltanto prodotti vincolati fra di loro
- L'utilizzo di diverse famiglie di prodotti da parte dello stesso cliente

Una variante comune di AF consiste nel creare una classe astratta Factory a cui si accede utilizzando il pattern Singleton.

È usata nelle librerie Java per la creazione di famiglie di elementi GUI per diversi sistemi operativi e sottosistemi GUI



(stereotipe = meta modello. I nomi delle classi sono solo indicativi, "AbstractFactory" non esiste)

## Singleton

**Problema:** è consentita (o richiesta) esattamente una sola istanza di una classe, ovvero un "singleton". Gli altri oggetti hanno bisogno di un punto di accesso globale e singolo a questo oggetto.

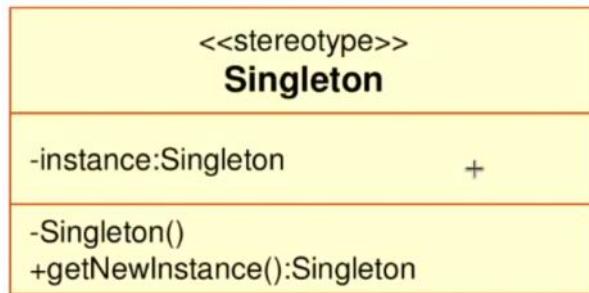
**Soluzione:** definisci un metodo statico (di classe) della classe che restituisce l'oggetto singleton.

Il "Singleton pattern definisce una classe della quale è possibile la istanziazione di un unico oggetto, tramite l'invocazione a un metodo della classe, incaricato della produzione degli oggetti.

Le diverse richieste di istanziazione comportano la restituzione di un riferimento allo stesso oggetto.

In UML un singleton viene illustrato con un "1" nella sezione del nome, in alto a destra

Singleton



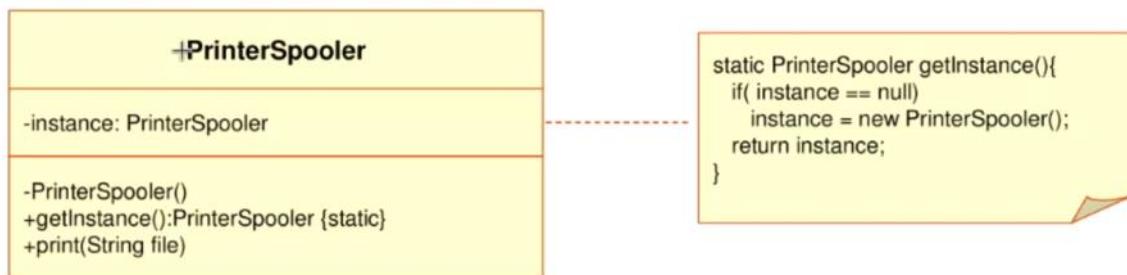
In Java esistono tre implementazioni:

- Singleton come classe statica: non è un vero Singleton, si lavora con la classe statica, non un oggetto. Questa classe statica ha metodi statici che offrono i servizi richiesti.
- Singleton creato da un metodo statico: una classe che ha un metodo statico che deve essere chiamato per restituire l'istanza del Singleton. L'oggetto verrà istanziato solo la prima volta. Le successive sarà restituito un riferimento allo stesso oggetto (inizializzazione pigra)
- Singleton multithread: versione multithread della versione precedente.

Nota: inizializzazione golosa: l'oggetto è istanziato quando la classe è caricata. È preferibile quella pigra perché se non si accede mai all'istanza viene evitato il lavoro della creazione. L'inizializzazione potrebbe essere complessa.

Il Singleton creato da un *metodo statico* è preferibile:

- I metodi d'istanza consentono la ridefinizione nelle sottoclassi e il raffinamento della classe singleton in sottoclassi
- La maggior parte dei meccanismi di comunicazione remota orientati agli oggetti supporta l'accesso remoto solo a metodi d'istanza
- Una classe non è sempre un singleton in tutti i contesti applicativi



# Strutturali

venerdì 8 maggio 2020 19:24

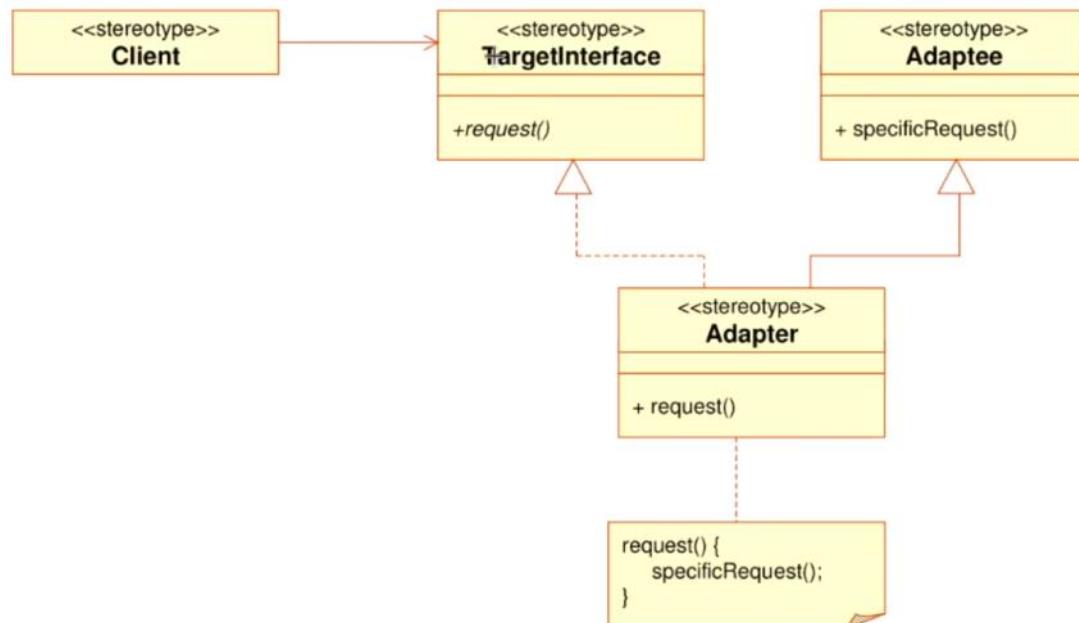
## Adapter

**Problema:** Come gestire interfacce incompatibili o fornire un'interfaccia stabile a comportamenti simili ma con interfacce diverse?

**Soluzione:** converti l'interfaccia originale di un componente in un'altra interfaccia, attraverso un oggetto adattatore intermedio

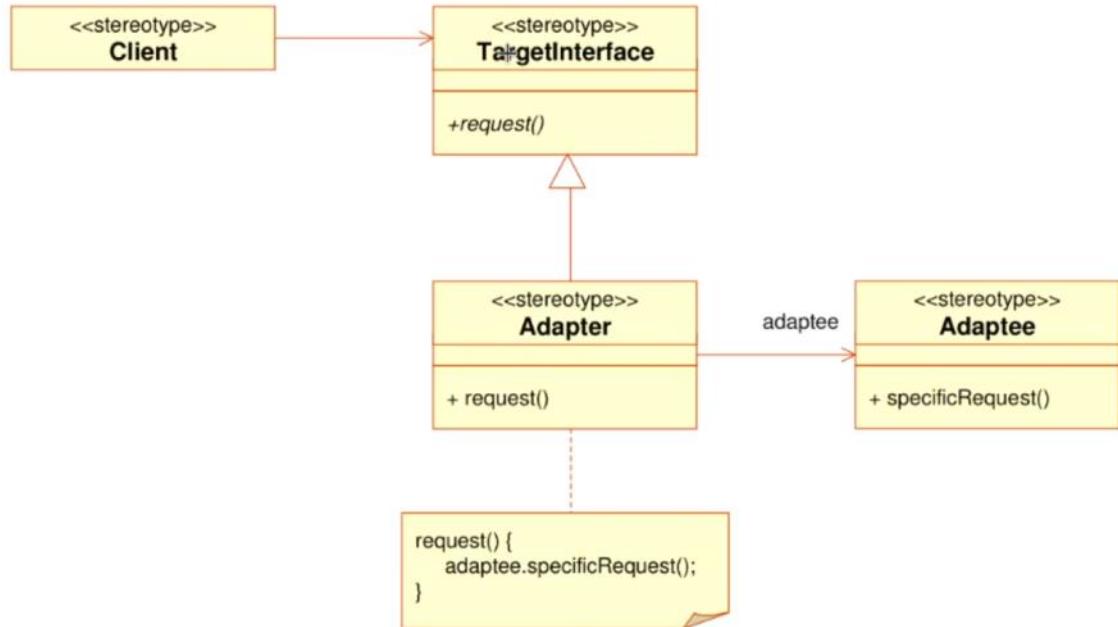
- Si consideri una coppia di oggetti software in una relazione client-server. Si parla di interfacce incompatibili quando l'oggetto server offre servizi di interesse per l'oggetto client ma l'oggetto client vuole fruire di questi servizi in una modalità diversa da quella prevista dall'oggetto server (**interfacce incompatibili**)
- Ci sono più oggetti server che offrono servizi simili; questi oggetti hanno interfacce simili ma diverse tra loro. Un oggetto client vuole fruire dei servizi offerti da uno tra questi oggetti server (**componenti simili con interfacce diverse**)
  - In generale un adattatore riceve richieste dai suoi client, per esempio da un oggetto dello strato di dominio, nel formato client dell'adattatore
  - L'adattatore adatta, trasforma, una richiesta ricevuta in una richiesta nel formato del server
  - L'adattatore invia la richiesta al server
  - Se il server fornisce una risposta, lo fa nel formato del server
  - L'adattatore adatta, trasforma, la risposta ricevuta dal server in una risposta nel formato del client e poi la restituisce al suo client.

Class Adapter



L'adapter **implementa** request e usa nell'implementazione specificRequest().

Object Adapter



L'adapter implementa l'interfaccia desiderata implementando request. Adapter ha una reference ad Adaptee e **delega** a lui.

## Composite

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

**Soluzione:** Definisci le classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia

Consente la costruzione di gerarchie di oggetti composti. Gli oggetti composti possono essere formati da oggetti singoli, oppure da altri oggetti composti.

Questo pattern è utile nei casi in cui si vuole:

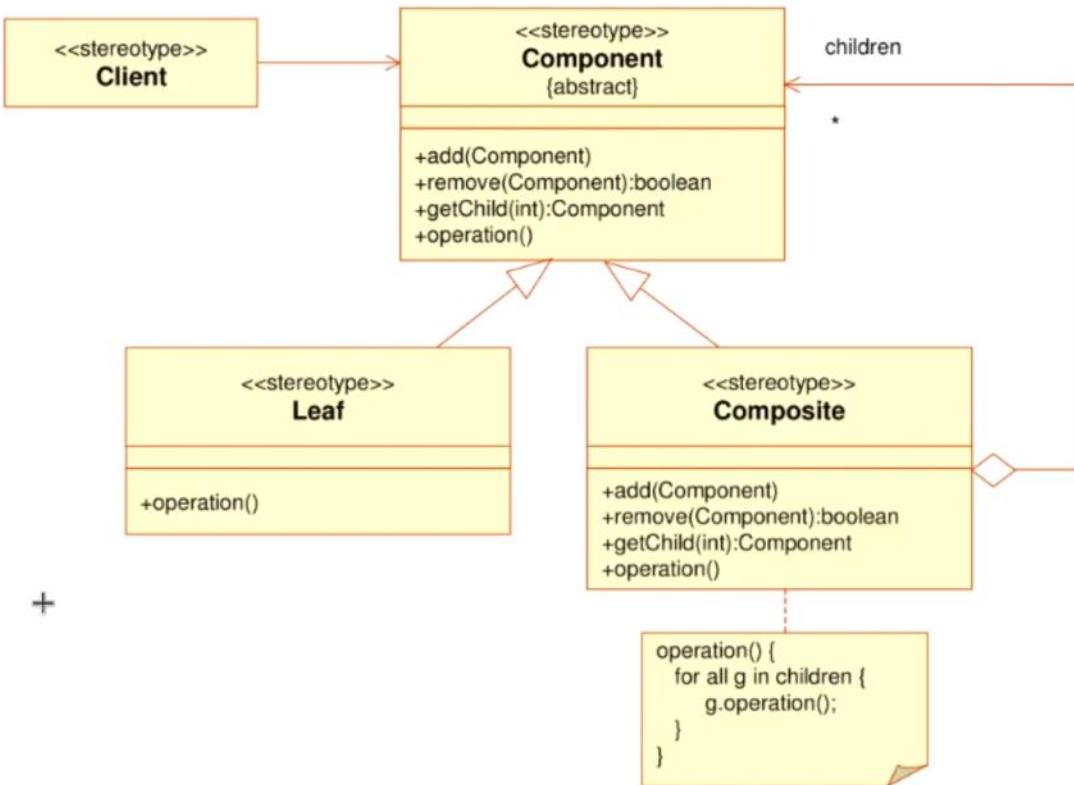
+

- Rappresentare gerarchie di oggetti tutto-parte
- Essere in grado di ignorare le differenze tra oggetti singoli e oggetti composti
- è nota anche come *struttura ad albero*, *composizione ricorsiva*, *struttura induttiva*: foglie e nodi hanno la stessa funzionalità
- implementa la stessa interfaccia per tutti gli elementi contenuti

Il pattern definisce la classe astratta componente che deve essere estesa in due sottoclassi:

- una che rappresenta i singoli componenti (foglia)
- l'altra che rappresenta i componenti composti e che si implementa come contenitore di componenti

Nota: i componenti composti possono immagazzinare sia componenti singoli sia altri contenitori.



Si usa l'ereditarietà per offrire una vista generalizzata su tutto. Il client vuole vedere tutto in una maniera indistinguibile.

L'ereditarietà è per generalizzare e dare un accesso disaccoppiato (posso cambiare le sotto-parti senza variare la classe astratta)

## Decorator

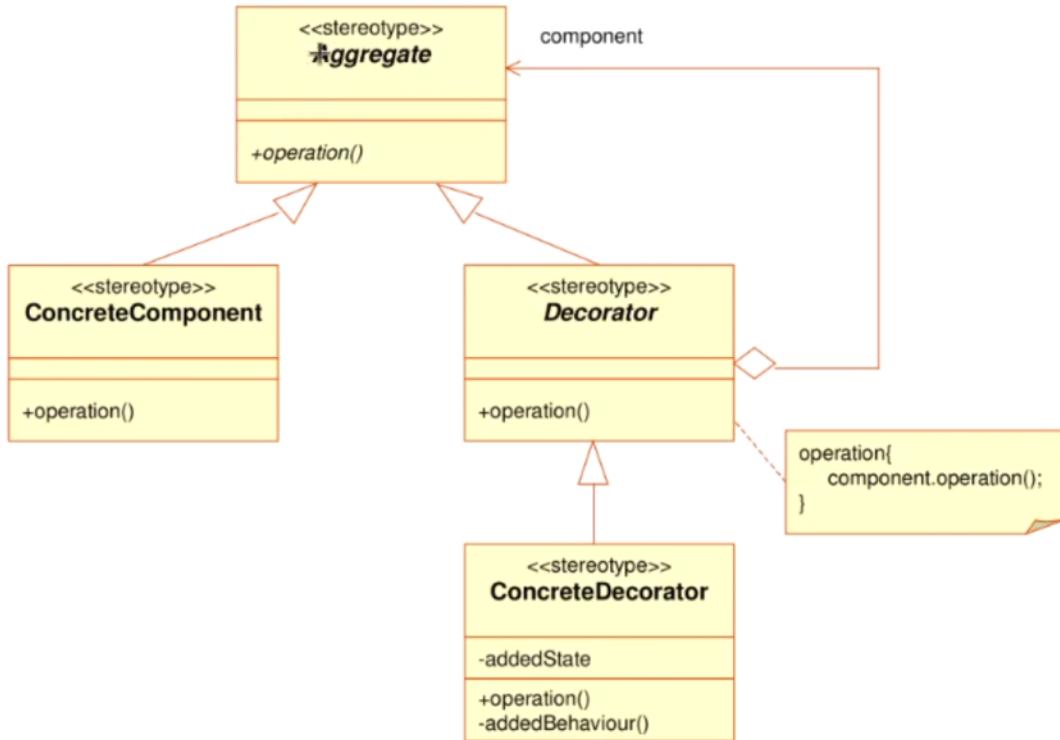
(Noto anche come Wrapper: inglobare l'oggetto all'interno di un altro che aggiunge le nuove funzionalità)

- **Problema:** come permettere di assegnare una o più responsabilità addizionali ad un oggetto in maniera dinamica ed evitare il problema della relazione statica?

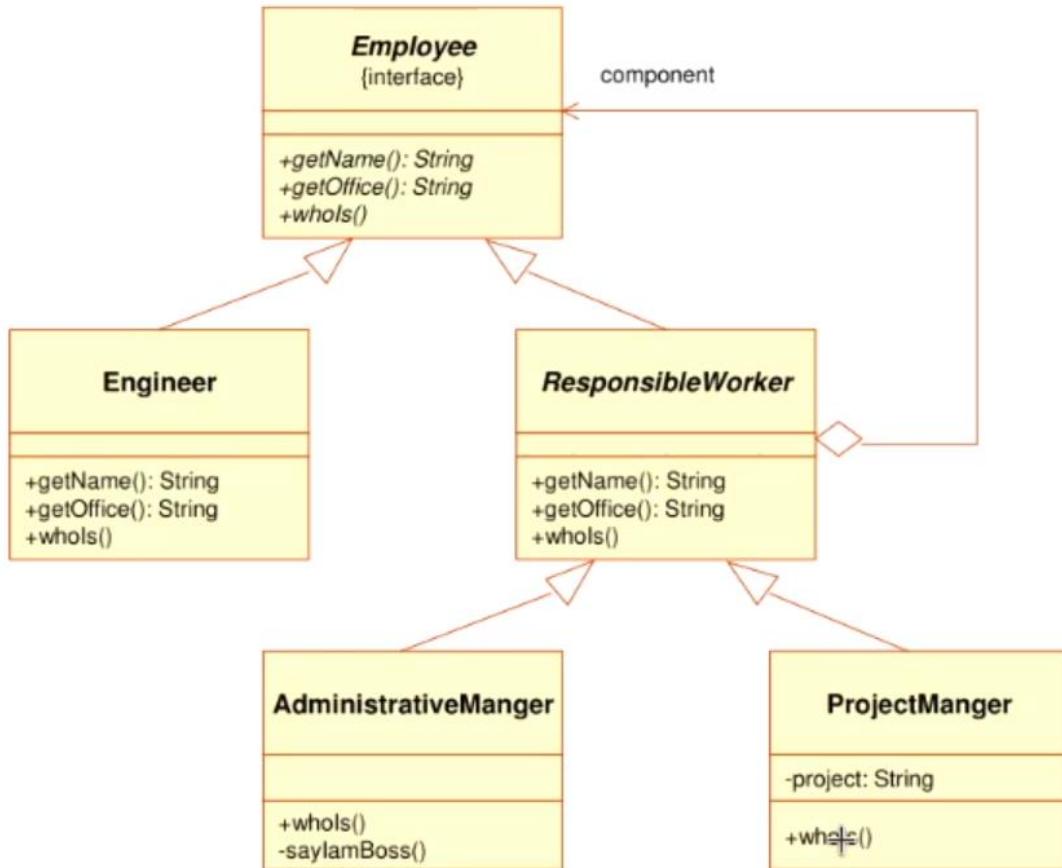
Come provvedere una alternativa più flessibile al meccanismo di sottoclasse ed evitare il problema di avere una gerarchia di classi complessa?

- **Soluzione:**

- Permette di aggiungere responsabilità ad oggetti individualmente, dinamicamente e in modo trasparente, ossia senza impatti sugli altri oggetti
- Le responsabilità possono essere ritirate
- Permette di evitare l'esplosione delle sotoclassi per supportare un ampio numero di estensioni e combinazioni di esse oppure quando le definizioni sono ~~scattered~~ sparse e non disponibili alle sottoclassi



Aggregate estende ConcreteComponent  
 Operation() di Decorator delega il lavoro.



# Comportamentali

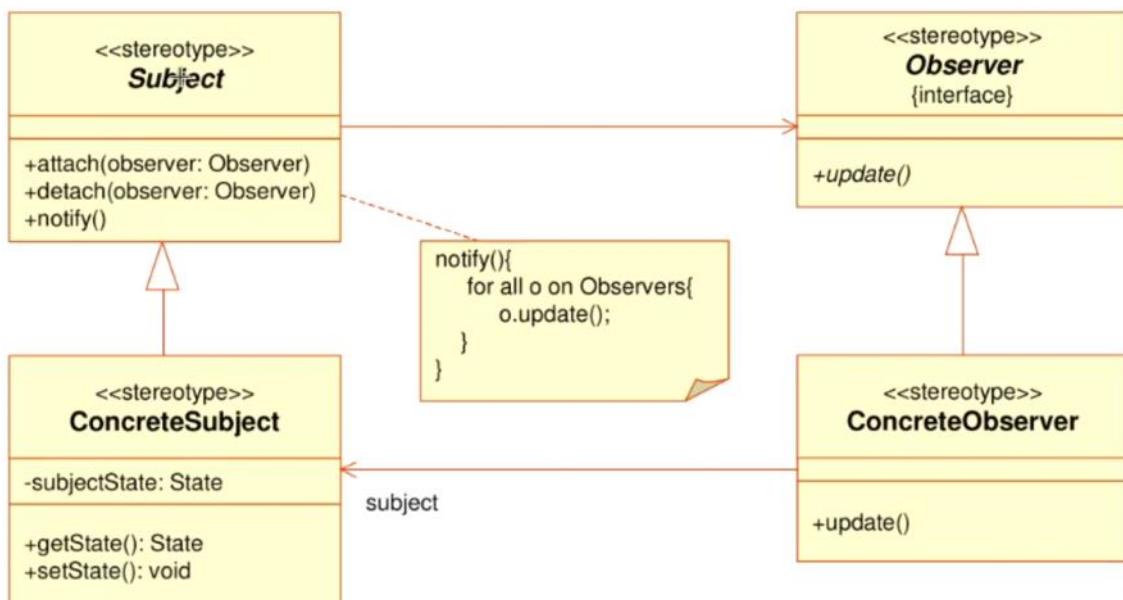
lunedì 11 maggio 2020 12:25

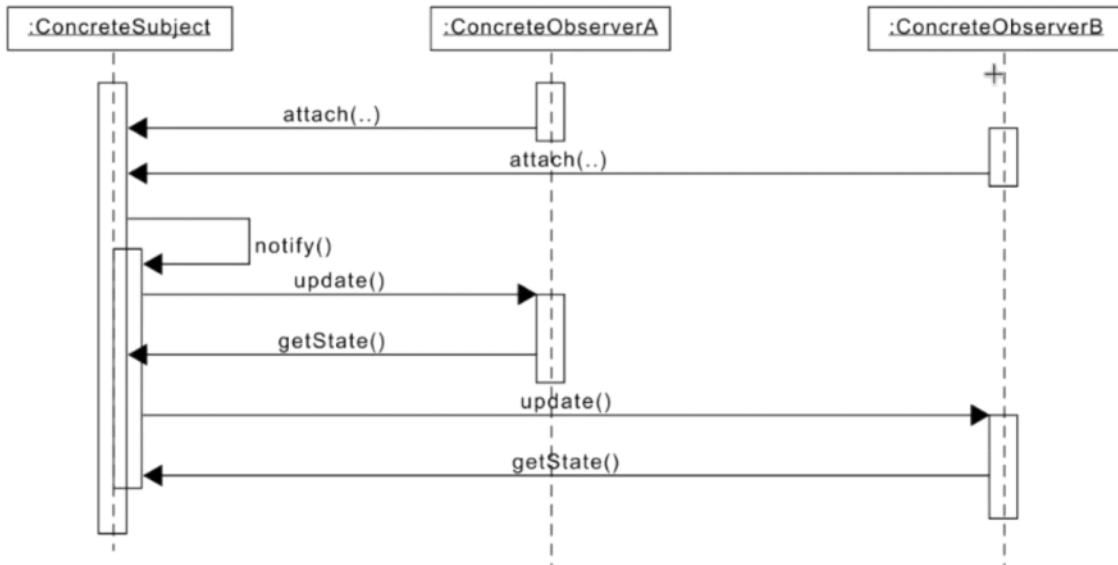
## Observer

**Problema:** Diversi tipi di oggetti subscriber sono interessati ai cambiamenti di stato o agli eventi di un oggetto publisher. Ciascun subscriber vuole reagire in un suo modo proprio quando il publisher genera un evento. Inoltre, il publisher vuole mantenere un accoppiamento basso verso i subscriber.  
**Soluzione:** definisci un'interfaccia subscriber o listener. Gli oggetti subscriber implementano questa interfaccia. Il publisher registra dinamicamente i subscriber che sono interessati ai suoi eventi, e li avvisa quando si verifica un evento.

- Definisce una dipendenza tra oggetti di tipo *uno-a-molti*: quando lo stato di un oggetto cambia, tale evento viene notificato a tutti gli oggetti dipendenti, essi vengono automaticamente aggiornati
- L'oggetto che notifica il cambiamento di stato non fa alcuna assunzione sulla natura degli oggetti notificati: le due tipologie di oggetti sono disaccoppiati
- il numero degli oggetti affetti dal cambiamento di stato di un oggetto non è noto a priori
- Fornisce un modo per accoppiare in modo debole degli oggetti che devono comunicare (eventi). I publisher conoscono i subscriber sono attraverso un'intefaccia, e i subscriber possono registrarsi (o cancellare la propria registrazione) dinamicamente con il publisher

Spesso associato al pattern architettonale Model-View-Controlle (MVC): le modifiche al modello sono notificate agli osservatori che sono le viste



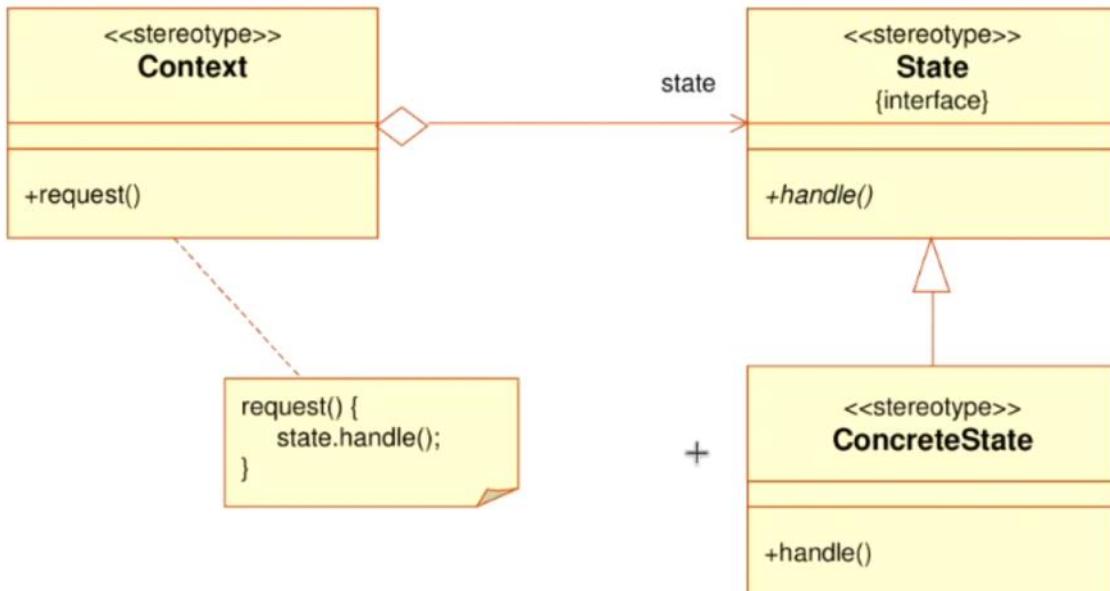


## State

**Problema:** Il comportamento di un oggetto dipende dal suo stato e i suoi metodi contengono logica condizionale per casi che riflette le azioni condizionali che dipendono dallo stato. C'è un'alternativa alla logica condizionale?

**Soluzione:** Crea delle classi stato per ciascuno stato, che implementano un'interfaccia comune. Delega le operazioni che dipendono dallo stato dall'oggetto contesto all'oggetto stato corrente corrispondente. Assicura che l'oggetto contesto referenzi sempre un oggetto stato che riflette il suo stato corrente.

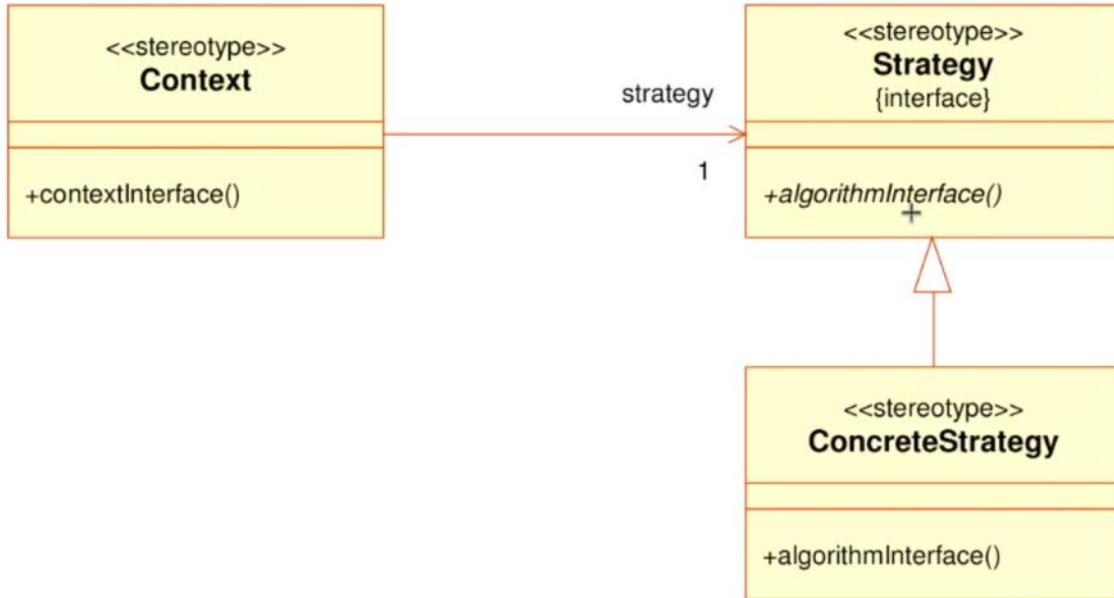
- Permette ad un oggetto di modificare il suo comportamento quando cambia il suo stato interno
- Può sembrare che l'oggetto modifichi la sua classe



## Strategy / Policy

**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.



- L'oggetto contesto è l'oggetto a cui va applicato l'algoritmo
- L'oggetto contesto è associato a un oggetto strategia, che è l'oggetto che implementa un algoritmo
- Consente la definizione di una famiglia di algoritmi, incapsula ognuno e li rende intercambiabili tra di loro
- Permette di modificare gli algoritmi in modo indipendente dai clienti che fanno uso di essi
- Disaccoppia gli algoritmi dai clienti che vogliono usarli dinamicamente
- Permette che un oggetto client possa usare indifferentemente uno o l'altro algoritmo
- È utile dove è necessario modificare il comportamento a runtime di una classe
- Usa la composizione invece dell'ereditarietà: i comportamenti di una classe non dovrebbero essere ereditati ma piuttosto incapsulati usando la dichiarazione di inferenza.

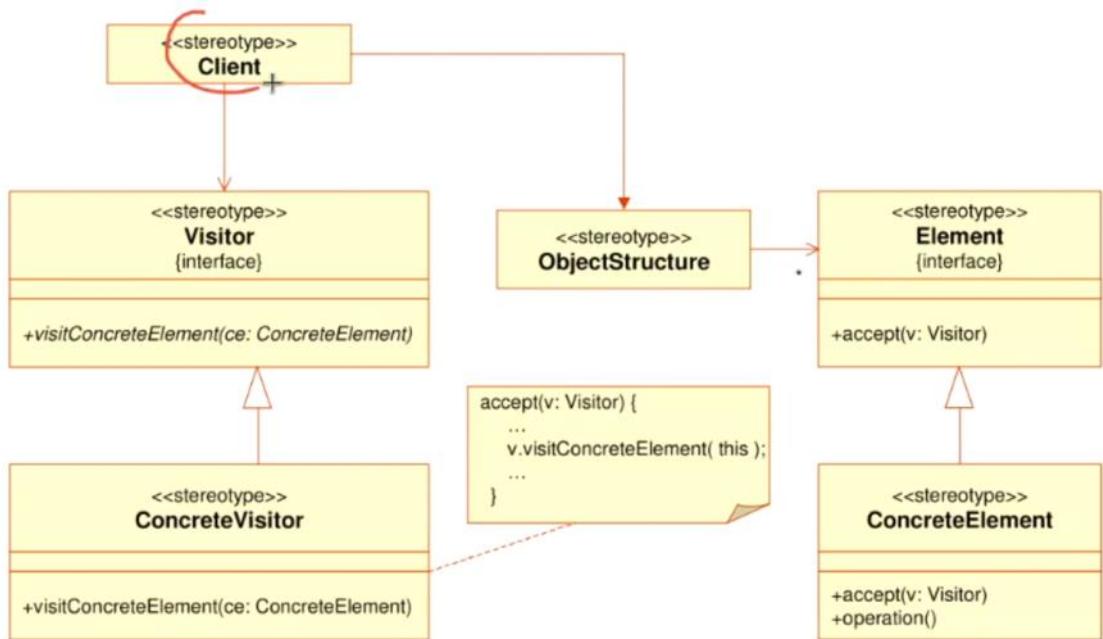
## Visitor

**Problema:** Come separare l'operazione applicata su un contenitore complesso dalla struttura dati cui è applicata? Come poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa? Come attraversare il contenitore complesso i cui elementi sono eterogenei applicando azioni dipendenti dal tipo degli elementi?

**Soluzione:** Creare un oggetto (`ConcreteVisitor`) che è in grado di percorrere la collezione, e di applicare un metodo proprio su ogni oggetto (`Element`) visitato nella collezione (avendo un riferimento a questi ultimi come parametro).

Ogni oggetto della collezione aderisce ad un'interfaccia (`Visitable`) che consente al `ConcreteVisitor` di essere accettato da parte di ogni `Element`. Il visitor analizza il tipo di oggetto ricevuto, fa l'invocazione alla particolare operazione che deve eseguire.

- Flessibilità delle operazioni
- Organizzazione logica
- Visita di vari tipi di classe
- Mantenimento di uno stato aggiornabile ad ogni visita
- Le diverse modalità di visita della struttura possono essere definite come sottoclassi del Visitor.



# Dal progetto al codice

lunedì 18 maggio 2020 18:16

Anche la prima iterazione si implementa!

Ricordare che ogni parte di codice prodotto è codice da produzione!

- Identificare i requisiti ✓
- Creare il modello di dominio ✓
- Aggiungere i metodi alle classi appropriate ✓
- Definire i messaggi fra gli oggetti per soddisfare i requisiti ✓

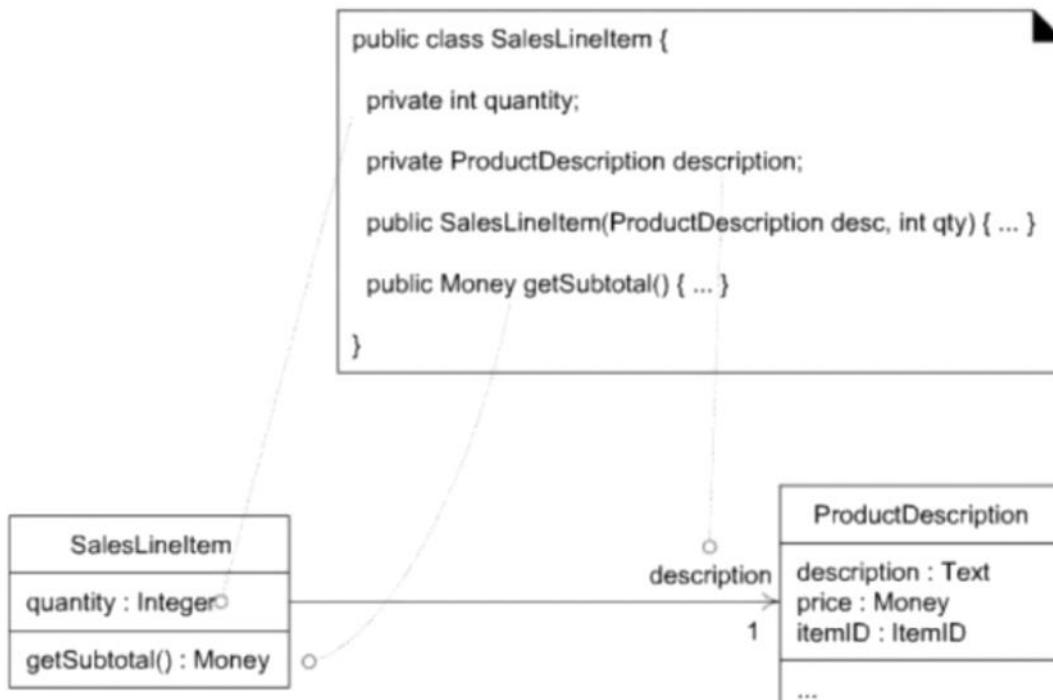
2020\_05\_08\_12\_52\_14.mp4

Siamo pronti per la realizzazione del **Modello di Implementazione**, costituito da tutti gli elaborati dell'implementazione, come il **codice sorgente**, la definizione delle **basi di dati**, le pagine JSP/XML/HTML, ecc.

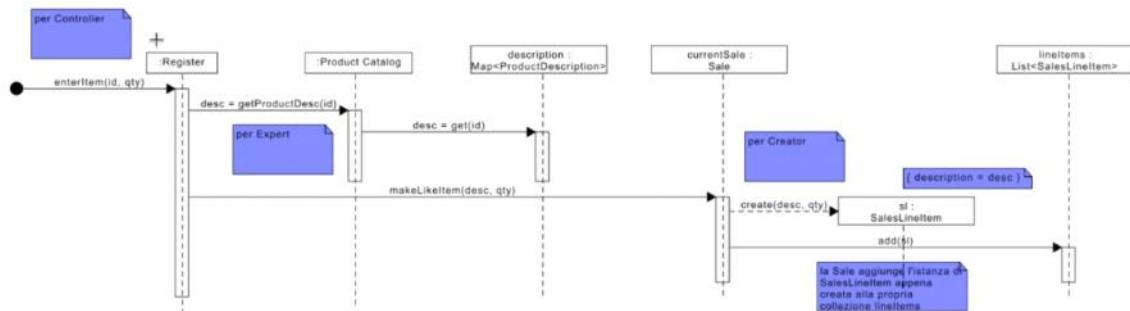
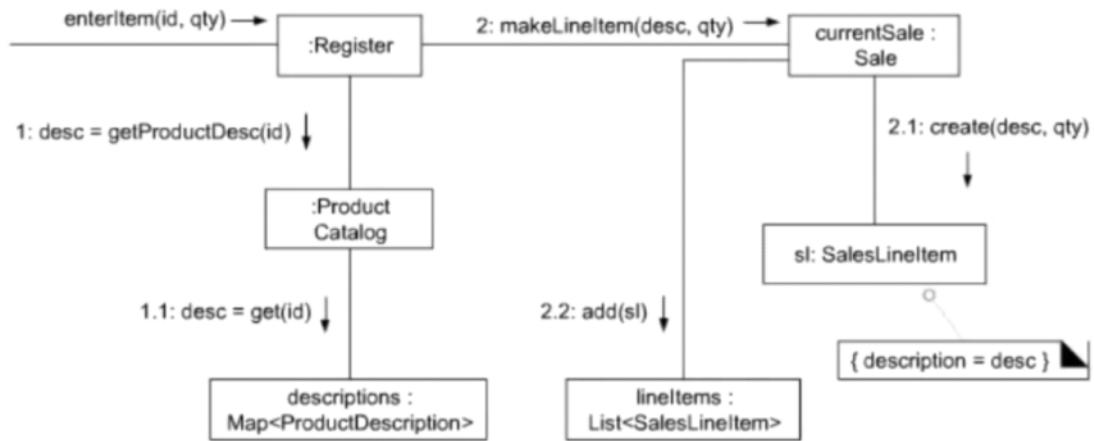
Abbiamo visto Java come esempio, molti altri linguaggi OO possono essere usati.

La creatività è stata spostata nelle fasi precedenti; quest'ultima fase è praticamente solo meccanica. Ci si deve aspettare cambiamenti e deviazioni durante la programmazione.

La traduzione in termini di definizione di attributi e di firme di metodi è spesso immediata.



La sequenza dei messaggi in un diagramma di interazione si traduce in una serie di istruzioni nelle definizioni di metodi e costruttori.

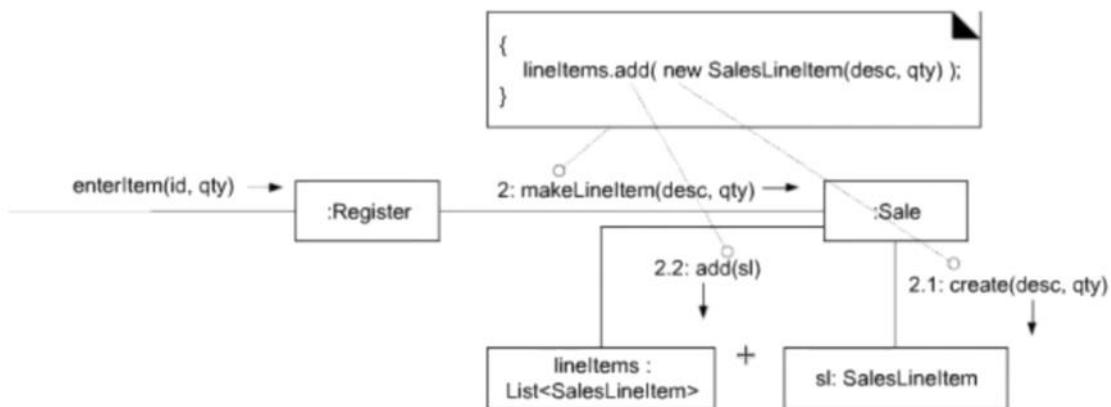


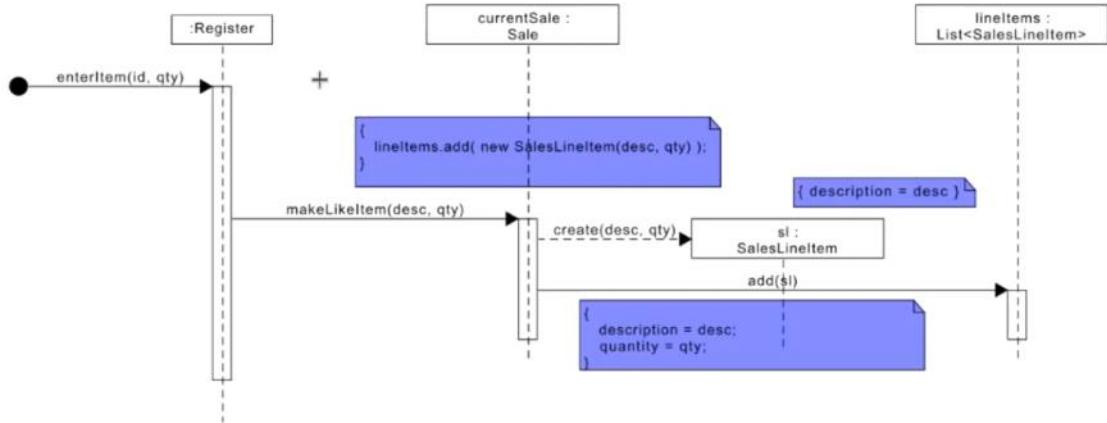
Le relazioni uno a molti sono implementate di solito con l'introduzione di un oggetto collezione.



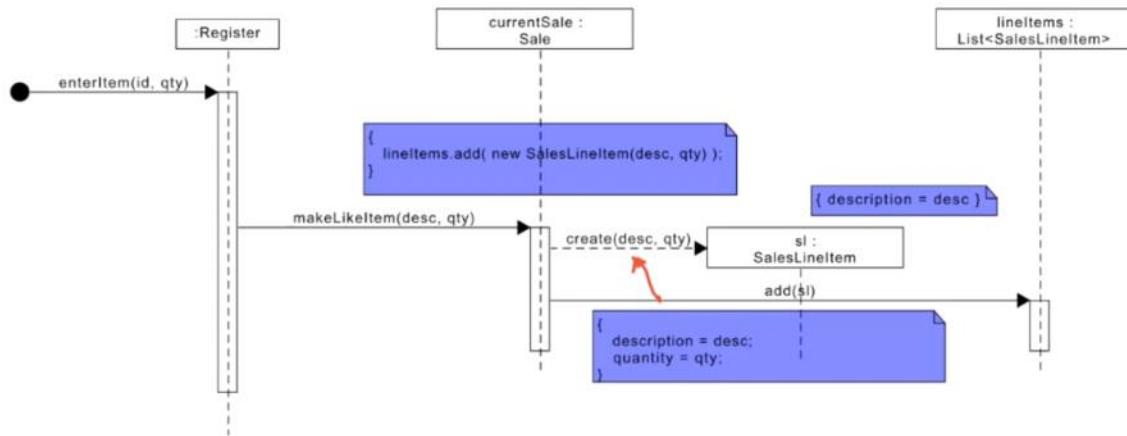
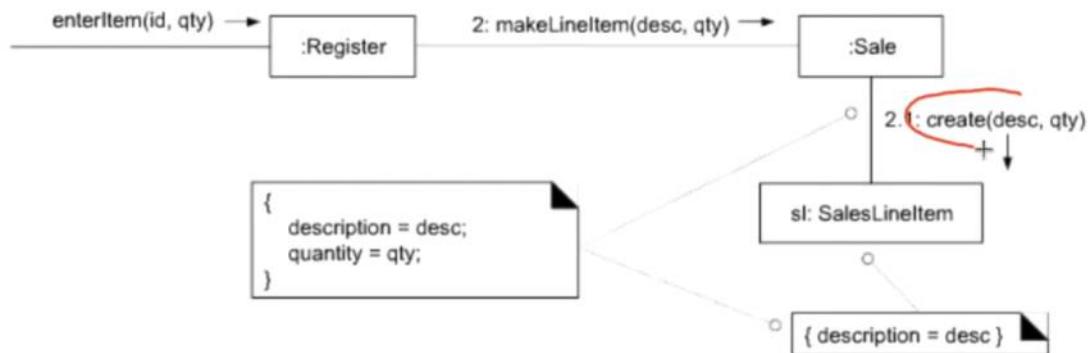
Private List<SalesLineItem> lineItems = new ArrayList();

Se un oggetto implementa un'interfaccia, si dichiara la variabile in termini dell'interfaccia, non della classe concreta!



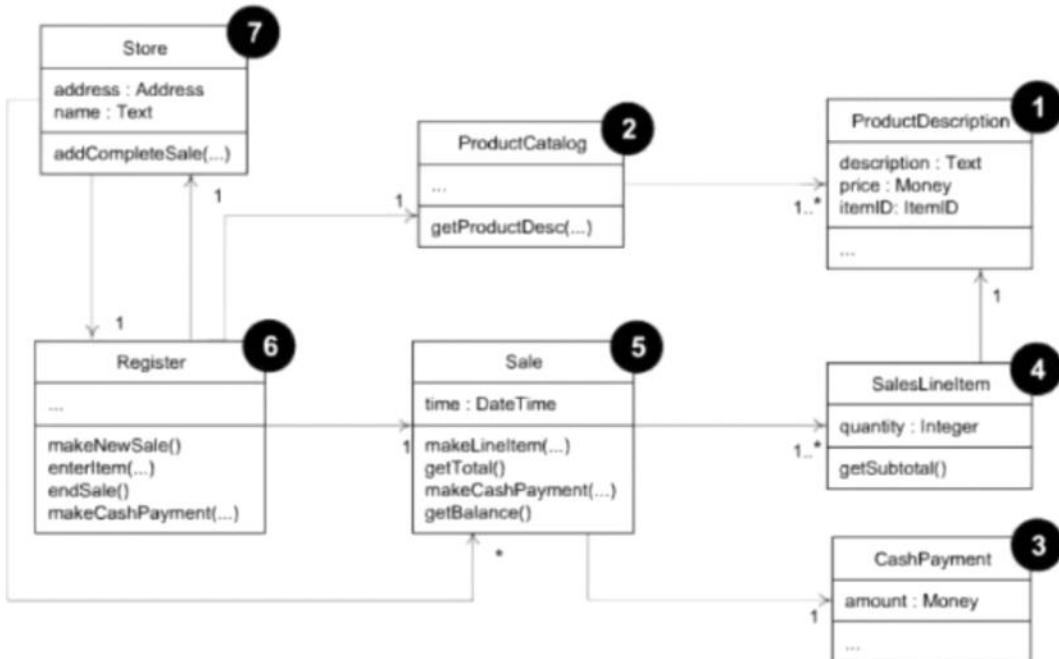


Il metodo `MakeLikelyItem` della classe `Sale` può essere scritto per ispezione del diagramma di collaborazione per `enterItem`.



Il costruttore della classe `SalesLineItem` è generato per ispezione da una versione parziale del diagramma di interazione per `enterItem`

Le classi possono essere implementate in modo e in ordine diverso, per esempio dalla meno accoppiata alla più accoppiata



## Sviluppo guidato dai test e refactoring

Extreme Programming ha promosso la pratica dei test: scriverli per primi

Promuove anche il refactoring continuo per migliorare la qualità: meno duplicazioni e maggiore chiarezza.

### TDD

Una pratica promossa dal metodo iterativo e agile XP è lo sviluppo guidato dai test.

Il codice dei test è scritto **Prima** del codice da verificare, immaginando che il codice da testare sia scritto.

- I test unitari vengono scritti
- La soddisfazione del programmatore porta a una scrittura più coerente dei test
- Chiarimento dell'interfaccia e del comportamento dettagliati
- Verifica dimostrabile, ripetibile e automatica
- Fiducia nei cambiamenti

In generale il TDD prevede l'utilizzo di diversi tipi di test:

- **Test unitari**: hanno lo scopo di verificare il funzionamento delle piccole parti (unità) del sistema ma non di verificare il sistema nel suo complesso
- **Test di integrazione**: per verificare la comunicazione tra specifiche parti (elementi strutturali) del sistema
- **Test end-to-end**: per verificare il collegamento complessivo tra tutti gli elementi del sistema
- **Test di accettazione**: hanno lo scopo di verificare il funzionamento complessivo del sistema, considerato a scatola nera e dal punto di vista dell'utente, ovvero con riferimento a scenari di casi d'uso del sistema.

### Unit Testing

Vi sono quattro parti:

- **Preparazione**: crea l'oggetto (o il gruppo di oggetti) da verificare (chiamato anche la **fixture**) e prepara altri oggetti e/o risorse necessari per l'esecuzione del test
- **Esecuzione**: fa fare qualcosa alla fixture (per esempio, eseguire delle operazioni), viene richiesto lo specifico comportamento da verificare
- **Verifica**: valuta che i risultati ottenuti corrispondano a quelli previsti
- **Rilascio**: optionalmente rilascia o ripulisce gli oggetti e/o le risorse utilizzate nel test (per evitare che altri test vengano corrotti)

## Refactoring

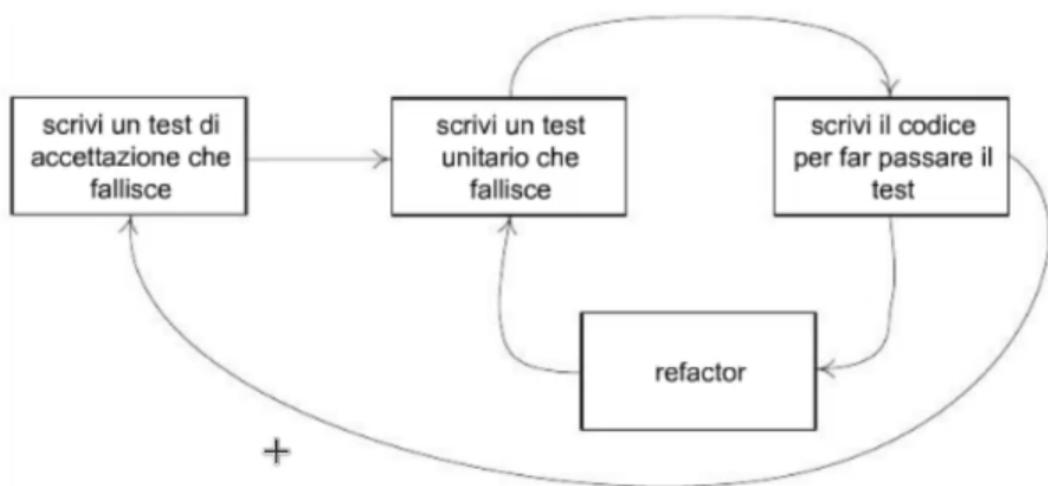
È un metodo strutturato e disciplinato per scrivere o ristrutturare del codice esistente senza però modificare il comportamento esterno, applicando piccoli passi di trasformazione in combinazione con la ripetizione dei test ad ogni passo.

Dopo ciascuna trasformazione i test unitari vengono eseguiti nuovamente per dimostrare che il refactoring non abbia provocato una regressione (un fallimento)

C'è una relazione tra il refactoring e il TDD: tutti i test unitari sostengono il processo di refactoring.



Il TDD è estendibile per effettuare anche i test di accettazione, per un'intera esecuzione di uno specifico caso d'uso.



Gli obiettivi del refactoring sono gli obiettivi e le attività di una buona programmazione:

- Eliminare il codice duplicato
- Migliorare la chiarezza
- Abbreviare i metodi lunghi
- Eliminare l'uso dei letterali costanti hard-coded
- ...

<b>Refactoring</b>	<b>Descrizione</b>
Rename	Per cambiare il nome di una classe, un metodo o un campo, per rendere più evidente il suo scopo. Semplice ma estremamente utile.
Extract Method	Trasforma un metodo lungo in uno più breve, estraendone una parte in un metodo di supporto.
Extract Class	Crea una nuova classe e vi muove alcuni campi e metodi da un'altra classe.
Extract Constant	Sostituisce un letterale costante con una variabile costante.
Move Method	Crea un nuovo metodo, con un corpo simile, nella classe che lo usa di più. +
Introduce Explaining Variable	Mette il risultato dell'espressione, o di una parte dell'espressione, in una variabile temporanea con un nome che ne spiega lo scopo.
Replace Constructor Call with Factory Method	In Java, per esempio, sostituisce l'uso dell'operazione <code>new</code> e la chiamata di un costruttore con l'invocazione di un metodo di supporto che crea l'oggetto (nascondendo i dettagli).