

# Sviluppo delle Applicazioni Software

Si noti che questi lucidi sono basati: sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016”; e parzialmente sul materiale fornito da Matteo Baldoni, Viviana Bono, Claudia Picardi e Gianluca Torta dell’Università degli Studi di Torino.

Appunti scritti e redatti da Alberto Marino, in riferimento all’anno accademico 2022/2023.



©2023 Copyright for this notes by Alberto Marino. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

<https://creativecommons.org/licenses/by/4.0/>

## Sommario

Capitolo 1 - Introduzione allo Unified Process (UP) .....	4
Fasi di UP .....	4
Discipline di UP .....	4
Requisiti evolutivi.....	5
Capitolo 2 - Ideazione.....	5
Capitolo 3 - Casi d'uso.....	6
Lavorare con gli UC in UP .....	7
Capitolo 4 - Elaborazione.....	7
Capitolo 5 - Modello di dominio.....	7
Capitolo 6 - Diagrammi di Sequenza di Sistema (SSD) .....	11
Capitolo 7 - Contratti .....	12
Capitolo 8 - Architettura logica e organizzazione in layer .....	13
Principio di separazione Modello-Vista.....	15
Capitolo 9 - Verso la progettazione ad oggetti: modellazione.....	16
Diagrammi di sequenza .....	17
Diagrammi di sequenza in UP: DSD (Design Sequence Diagram).....	17
Componenti dei DSD: partecipanti .....	17
Componenti DSD: singleton .....	18
Componenti DSD: linee di vita e messaggi .....	18
Componenti DSD: risposte o ritorni .....	18

Componenti DSD: self o this .....	19
Componenti DSD: creazione di istanze .....	19
Componenti DSD: distruzione di oggetti.....	19
Componenti DSD: formazione di collegamenti .....	19
Componenti DSD: frame .....	20
Componenti DSD: correlare diagrammi di interazione.....	20
Componenti DSD: invocare metodi statici.....	21
Componenti DSD: chiamate sincrone e asincrone.....	21
Diagrammi di sequenza in UP: DCD (Design Class Diagram) .....	21
Diagrammi delle classi: notazione per attributi come associazioni .....	22
Diagrammi delle classi: modello di dominio VS DCD.....	23
Diagrammi delle classi: associazioni e attributi .....	23
Diagrammi delle classi: attributi collezione e note .....	23
Diagrammi delle classi: operazioni e metodi .....	24
Diagrammi delle classi: parole chiave .....	24
Diagrammi delle classi: generalizzazione .....	24
Diagrammi delle classi: dipendenze.....	25
Diagrammi delle classi: interfacce .....	25
Diagrammi delle classi: composizione .....	26
Diagrammi delle classi: singleton .....	26
Diagrammi delle classi: template .....	26
Diagrammi delle classi e diagrammi di interazione/sequenza .....	27
Capitolo 10 - GRASP (General Responsibility Assignment Software Patterns) .....	27
RDD (Responsability-Driven Development).....	27
Passi della RDD .....	28
GRASP .....	28
GRASP Patterns .....	29
GRASP Pattern Creator .....	29
GRASP Pattern Information Expert .....	29
GRASP Pattern Low Coupling .....	30
GRASP Pattern Controller.....	31
GRASP Pattern High Cohesion.....	31
Capitolo 11 - GoF (Gang-of-Four) .....	32
GoF Patterns.....	33
Notazione dei GoF Patterns.....	33
GoF Pattern Creazionali: Abstract Factory.....	33

GoF Pattern Creazionali: Singleton .....	34
GoF Pattern Strutturali: Adapter.....	34
GoF Pattern Strutturali: Composite .....	35
GoF Pattern Strutturali: Decorator .....	36
GoF Pattern Comportamentali: Observer .....	36
GoF Pattern Comportamentali: State .....	37
GoF Pattern Comportamentali: Strategy.....	37
GoF Pattern Comportamentali: Visitor .....	38
Capitolo 12 – Dal progetto al codice .....	38
Sviluppo guidato dai test e refactoring.....	38
XP (Extreme Programming) e TDD (Test-Driven Development).....	38
Refactoring .....	39
Ricordi? Pillole di Java.....	40

## Capitolo 1 - Introduzione allo Unified Process (UP)

Per studiare **OOA** (Object Oriented Analysis) e **OOD** (Object Oriented Design) utilizziamo **UP** (Unified Process), un **processo iterativo** per lo sviluppo del software orientato agli oggetti che utilizza **UML**, un **linguaggio visuale** per la specifica, la costruzione e la documentazione degli elaborati di un sistema software. Gli usi di UML possono essere i seguenti:

- Punto di vista **concettuale (modello di dominio)**;



- Punto di vista **software (diagramma delle classi di progetto)**.

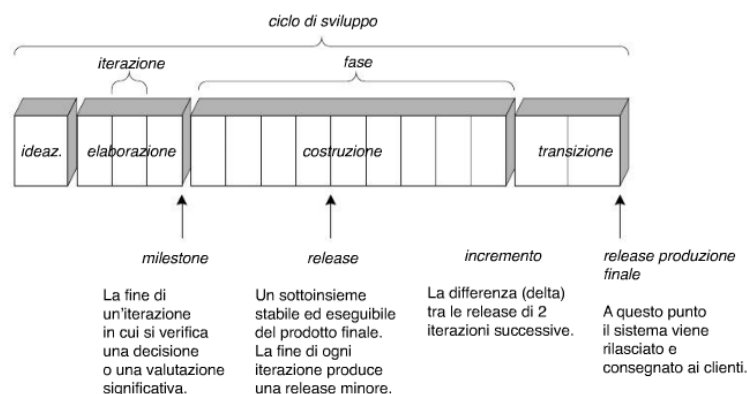


### Fasi di UP

UP è un processo iterativo ed evolutivo. È composto da quattro fasi:

- **Ideazione**: viene eseguita un'indagine per decidere di proseguire con il progetto o di interromperlo. Milestone\*: **obiettivi**;
- **Elaborazione**: implementazione iterativa del nucleo dell'architettura, risoluzione dei rischi maggiori e identificazione della maggior parte dei requisiti e della portata. Milestone\*: **architetturale**;
- **Costruzione**: implementazione iterativa degli elementi rimanenti, più facili e a rischio minore, preparazione al rilascio. Milestone\*: **capacità operativa**;
- **Transizione**: beta test e rilascio. Milestone\*: **rilascio prodotto**.

\* **Milestone**: traguardo importante da raggiungere



### Discipline di UP

- Disciplina di **Modellazione del business**: attività che modellano il dominio del problema ed il suo ambito. Fa parte di questa disciplina il **Modello di Dominio**;

- Disciplina dei **Requisiti**: attività di raccolta dei requisiti del sistema. Fanno parte di questa disciplina: **Modello dei Casi d'Uso (SSD)**, **Visione**, **Specifiche Supplementari**, **Glossario**, **Regole di business**;
- Disciplina di **Progettazione** (analysis and design): attività di analisi dei requisiti e progetto architeturale del sistema. Fanno parte di questa disciplina: **Modello di progetto**, **Documento dell'Architettura**, **Software**, **Modello dei Dati**;
- Disciplina di **Implementazione**: attività di progetto dettagliato e codifica del sistema, test su componenti;
- Disciplina di **Test**: attività di controllo di qualità, test di integrazione e di sistema;
- Disciplina di **Rilascio**: attività di consegna e messa in opera;
- Disciplina di **Gestione delle configurazioni e del cambiamento**: attività di manutenzione durante il progetto;
- Disciplina di **Gestione progetto**: attività di pianificazione e governo del progetto;
- Disciplina di **Infrastruttura** (environment): attività che supportano il team di progetto, riguardo ai processi e strumenti utilizzati.

**Come viene usato UML in UP?** UP usa UML come linguaggio di modellazione; i diagrammi si usano in UP seguendo le caratteristiche di iterazione ed incremento; UP dice quando usare un programma.

## Requisiti evolutivi

**Requisito**: capacità o condizione a cui il sistema (e il progetto) deve essere conforme. I requisiti possono essere:

- **Requisiti funzionali**: descrivono il comportamento del sistema, in termini di funzionalità fornite agli utenti.  
Un **esempio** di requisito funzionale riguardante un sistema di prenotazione di voli online potrebbe essere: *“Il sistema deve consentire agli utenti di cercare e prenotare voli in base alla destinazione e alla data desiderata”*;
- **Requisiti non funzionali**: le proprietà del sistema nel suo complesso (sicurezza, prestazioni, usabilità, ...).  
Un **esempio** di requisito non funzionale riguardante un sistema di prenotazione di voli online potrebbe essere: *“Il sistema deve garantire una risposta alle richieste degli utenti entro 2 secondi per fornire una risposta tempestiva”*.

UP ha diversi **elaborati**:

- **Modello dei Casi d'Uso**: scenari tipici dell'utilizzo di un sistema;
- **Specifiche Supplementari**: ciò che non rientra nei casi d'uso,
- **Glossario**: termini significativi, dizionario dei dati;
- **Visione**: un documento sintetico per apprendere rapidamente le idee principali del progetto;
- **Regole di Business**: regole di dominio, i requisiti o le politiche a cui il sistema deve conformarsi.

## Capitolo 2 - Ideazione

Permette di stabilire una visione comune e la portata del progetto (studio di fattibilità). Durante l'ideazione:

- Si analizzano **circa 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;
- **Durata**: normalmente breve.

Lo scopo della fase di ideazione non è quello di definire tutti i requisiti, né di generare una stima o un piano di progetto affidabili.

L'ideazione non rappresenta il momento per fare tutti i requisiti o creare stime o piani affidabili. **La maggior parte dei requisiti avviene durante la fase di elaborazione**, in parallelo alle prime attività di programmazione di qualità-produzione e di test.

**Durante l'ideazione si tratta di decidere se il progetto merita un'indagine più seria** (durante l'elaborazione), non di effettuare questa indagine.

In fase di ideazione:

- Le **specifiche supplementari**: raccolgono altri requisiti, informazioni e vincoli che non sono facilmente colti dai casi d'uso o nel glossario;
- Il **documento Visione**: riassume alcune delle informazioni contenute del modello dei casi d'uso e nelle specifiche supplementari. Descrive brevemente il progetto, come contesto per i partecipanti, al fine di stabilire una visione comune del progetto;
- Il **glossario** è un elenco dei termini significativi e delle relative definizioni. In UP svolge anche il ruolo di **dizionario dei dati** con le relative regole di validazione;
- Le **regole di dominio**: stabiliscono come può funzionare un dominio o un business.

## Capitolo 3 - Casi d'uso

Processo per scoprire cosa deve essere costruito ed orientare lo sviluppo verso il sistema corretto.

Passi principali:

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

Ogni requisito è caratterizzato:

- Breve **descrizione**;
- **Stato** (es. proposto, approvato, incorporato, validato);
- **Costi di implementazione** stimato;
- **Priorità**;
- **Rischio** associato per la sua implementazione.

UP è una metodologia "**use-case driven**":

- **Attori**: qualcosa o qualcuno dotato di un comportamento;
- **Scenario**: sequenza specifica di azioni ed interazioni tra il sistema e alcuni attori. Descrive una particolare storia nell'uso del sistema, ovvero un percorso attraverso il caso d'uso;

- **Caso d'uso** (o casi di utilizzo): una collezione di scenari correlati (di successo e di fallimento) che descrivono un attore che usa il sistema per raggiungere un obiettivo specifico.

Un **attore** è qualcosa o qualcuno dotato di comportamento. Esso può essere: **primario, di supporto o fuori scena**.

Come scrivere il caso d'uso?

- **Scenario principale di successo**: chiamato anche “**percorso felice**” e descrive un percorso di successo comune che soddisfa gli interessi delle parti interessate. È costituito da una sequenza di passi che possono ripetersi più volte. I passi possono essere di tre tipi:
  - Un'interazione tra attori;
  - Un cambiamento di stato da parte del sistema;
  - Una validazione.
- **Estensioni**: hanno lo scopo di descrivere **tutti gli altri scenari oltre a quello principale**, sia di successo che di fallimento. Solitamente le estensioni sono descritte per differenza dallo scenario principale.

## Lavorare con gli UC in UP

	Identificazione UC	Descrizione dettagliata UC	Realizzazione UC
Ideazione	50% - 70%	10%	5%
Elaborazione	Quasi 100%	40% - 80%	Meno del 10%
Costruzione	100%	100%	100%
Transizione			

## Capitolo 4 - Elaborazione

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.

Durante questa fase vengono prodotti **codice** e **progettazione** che sono parti di qualità-produzione del sistema finale.

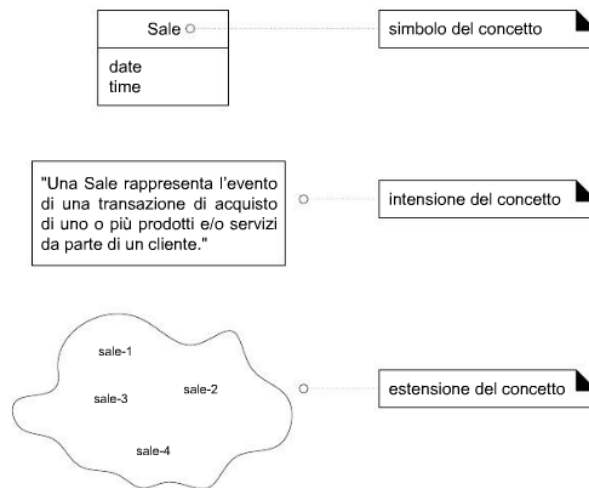
## Capitolo 5 - Modello di dominio

È una **rappresentazione visuale (e NON testuale!)** delle **classi concettuali (e NON software!)**. È un insieme di diagrammi di classi UML che includono:

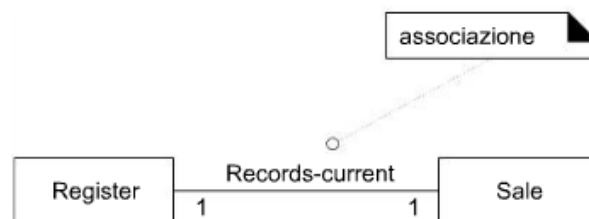
- **Oggetti** di dominio-classi concettuali;
- **Associazioni** tra classi concettuali;
- **Attributi** di classi concettuali;
- **Non appaiono operazioni**.

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

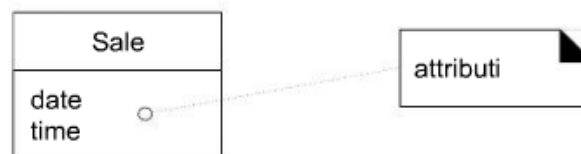
- **Simbolo**: parola o immagine che rappresenta la classe concettuale;
- **Intenzione**: definizione della classe concettuale;
- **Estensione**: insieme degli oggetti della classe concettuale.



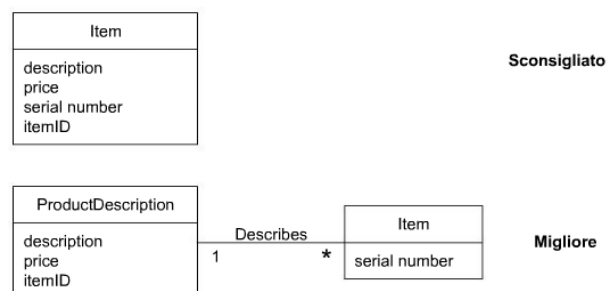
Una **associazione** è una relazione tra due o più classi che indica una connessione significativa tra le istanze di quelle classi.



Un **attributo** è un valore logico degli oggetti di una classe.

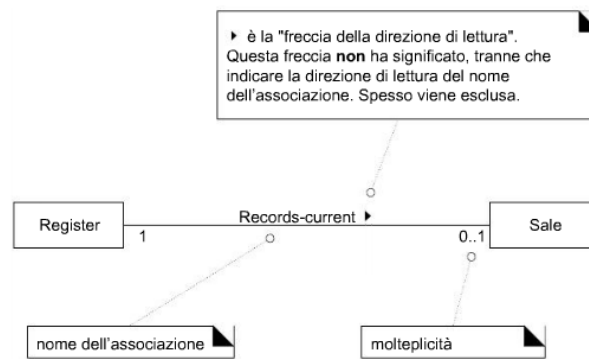


Le **classi "descrizione"** contengono informazioni che descrivono qualcos'altro. È utile avere un'associazione che collega la classe e la sua descrizione.

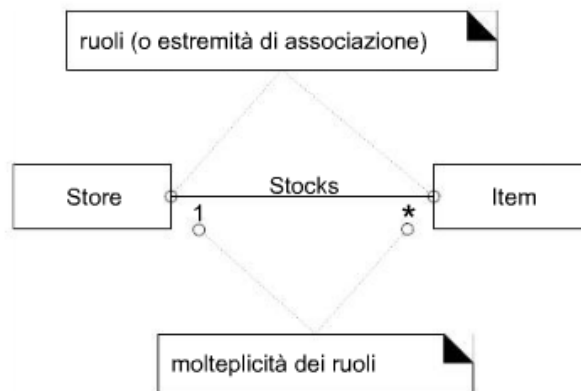


Un'associazione è per natura **bidirezionale**. È caratterizzabile con un nome significativo seguendo la traccia NomeClasse-FraseVerbale-NomeClasse. Occorre indicare molteplicità (cardinalità) e direzione di lettura (freccia).



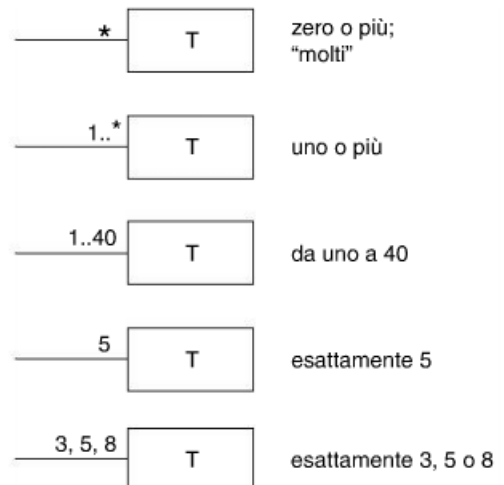


Ciascuna estremità di un'associazione è chiamata **ruolo**. I ruoli possono esprimere **molteplicità, nome e navigabilità**.

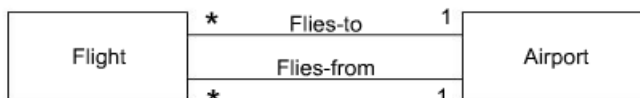


La **molteplicità delle associazioni** (o cardinalità) definisce quante istanze di una classe possono essere associate ad una istanza di un'altra. I casi di molteplicità possibili sono:

- uno-a-molti;
- molti-a-uno;
- molti-a-molti;
- uno-a-uno.



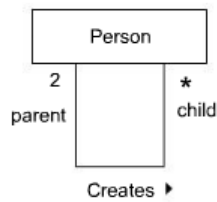
È possibile avere **associazioni multiple**, ovvero che due classi siano collegate da più di un'associazione.



È possibile indicare i **nomi di ruolo**.



Nelle **associazioni riflesse** una classe può avere un'associazione con sé stessa. I nomi di ruolo risultano utili per indicare la molteplicità.

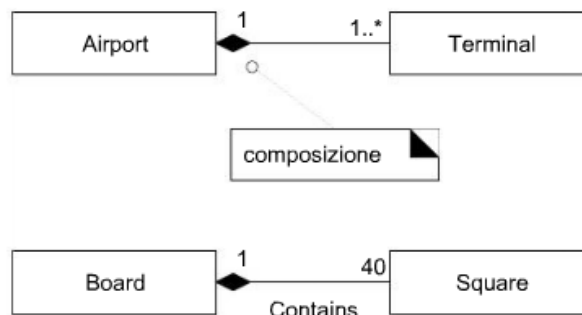


La **composizione** è un **tipo forte di aggregazione intero-parte**:

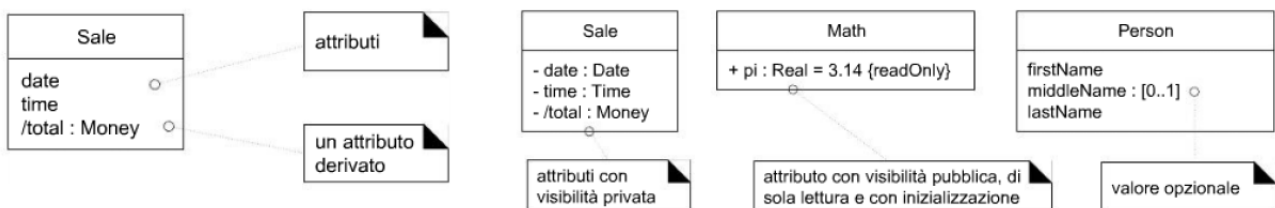
- Ciascuna istanza della parte appartiene ad **una sola** istanza del composto alla volta;
- Ciascuna parte deve sempre appartenere **a un** composto;
- La vita delle parti è limitata da quella del composto: le parti possono essere **create dopo il composto** (ma non prima) e possono essere **distrutte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

Negli esempi seguenti:

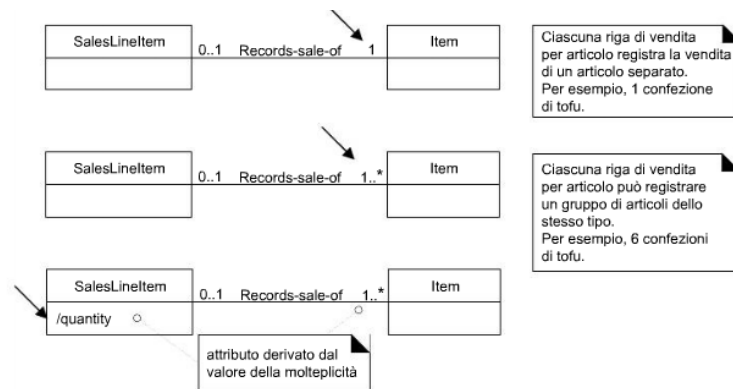
- *Airport* è l'entità "padre" e corrisponde al composto, *Terminal* è l'entità "figlia" e corrisponde alla parte;
- *Board* è l'entità "padre" e corrisponde al composto, *Square* è l'entità "figlia" e corrisponde alla parte;



Gli **attributi** sono dati primitivi oppure enumerativi. Le classi concettuali vanno correlate con un'associazione, non con un attributo.

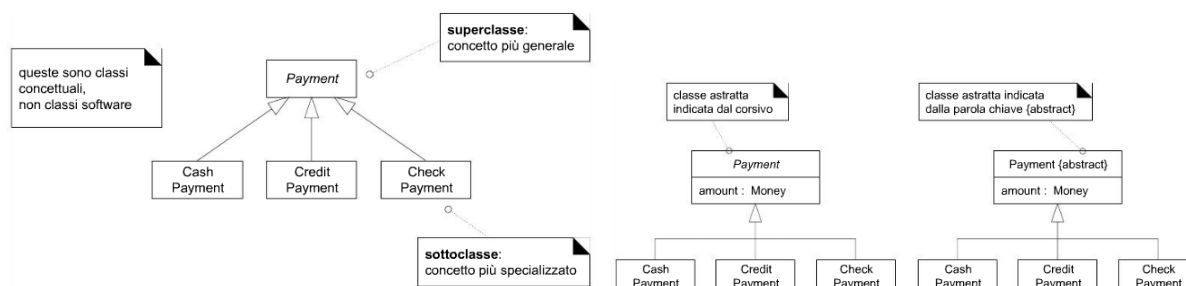


Un **attributo derivato** può essere calcolato o derivato dalla informazioni contenute negli oggetti associati. Può essere indicato dal simbolo "/" prima del nome dell'attributo.



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**, ovvero la capacità di sostituire un elemento specifico con un altro elemento equivalente all'interno del modello senza alterare le proprietà o le relazioni del sistema. Una classe concettuale è **astratta** se ciascun suo elemento è anche elemento di una delle sue sottoclassi.

**N.B.: NON è una specializzazione**, si fa dall'alto verso il basso.



Ad ogni sottoclasse possono essere associate associazioni diverse.

## Capitolo 6 - Diagrammi di Sequenza di Sistema (SSD)

È un elaborato della **disciplina dei requisiti** che illustra eventi di input e di output relativi ai sistemi in discussione. Si modella un SSD per ogni caso d'uso per lo scenario principale e per ogni scenario alternativo. SSD costituiscono un input per i **contratti** delle operazioni e per la progettazione degli oggetti.

Un SSD è 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:

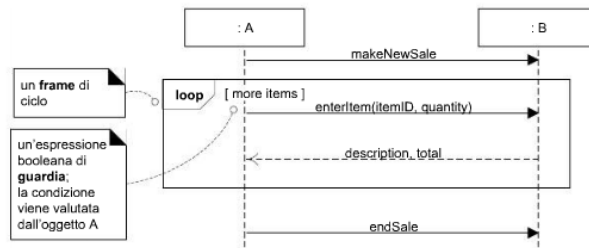
- **Eventi esterni** da parte di attori umani o sistemi informatici;
- **Eventi temporali**;
- Guasti o **eccezioni**.

Un SSD mostra:

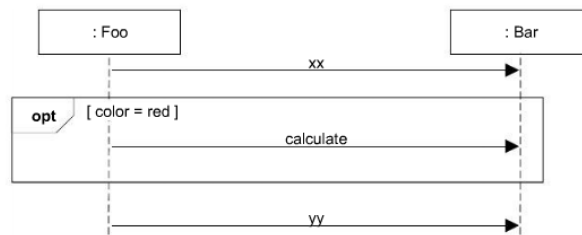
- L'**attore primario** del caso d'uso;
- Il **sistema** in discussione;

- I passi che rappresentano le **interazioni** tra il sistema e l'attore.

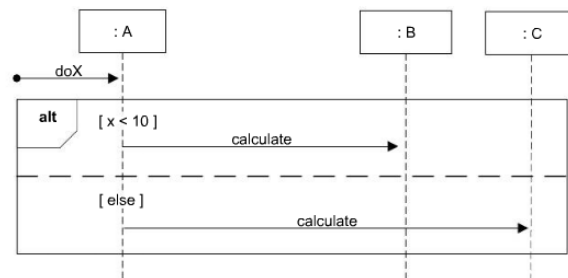
Un **esempio** di frame di UML è il seguente:



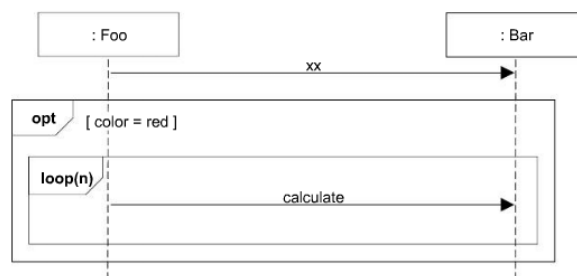
Un **messaggio condizionale** si effettua usando un frame **opt**.



Un **messaggio condizionale** può essere **mutuamente esclusivo**.



I **frame** possono essere **annidati** seguendo la logica della programmazione classica.



## Capitolo 7 - Contratti

Le **operazioni di sistema** sono operazioni che il sistema offre nella sua interfaccia pubblica. Le operazioni di sistema possono essere identificate mentre si abbozzano gli SSD. Gli SSD mostrano eventi di sistema.

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

I principali input dei contratti sono le **operazioni di sistema** identificate negli SSD e il **modello di dominio**.

Le sezioni di un contratto sono:

- **Operazione:** nome e parametri dell'operazione;
- **Riferimenti:** casi d'uso in cui può verificarsi questa operazione;
- **Pre-condizioni:** ipotesi significative sullo stato del sistema o degli oggetti nel modello di dominio prima dell'esecuzione dell'operazione. Si tratta di ipotesi non banali che dovrebbero essere comunicate al lettore;
- **Post-condizioni:** descrivono i cambiamenti di stato degli oggetti nel modello di dominio dopo il completamento dell'operazione.

Le **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.

Le **pre-condizioni** descrivono le ipotesi significative sullo stato del sistema prima dell'esecuzione dell'operazione. Una descrizione sintetica dello stato di avanzamento del caso d'uso.

Ogni operazione di sistema può avere una componente di **trasformazione** (il sistema cambia il proprio stato) e/o una componente di **interrogazione** (il sistema calcola e restituisce valori).

Un'operazione di sistema **ha post-condizioni** solo se implica una trasformazione, mentre **non ha post-condizioni** se si tratta semplicemente di un'interrogazione.

## Q&A

- Bisogna scrivere un contratto per ogni evento di sistema trovato nel SSD?  
*Non è necessario: consideriamo quelli più complessi*
- Se si scoprono nuove classi, attributi, si possono aggiungere nel modello di dominio?  
*Ovvio! UP è incrementale*
- Le post-condizioni devono essere in ogni momento le più complete possibili?  
*Non è necessario: UP iterativo ed incrementale*

## Capitolo 8 - Architettura logica e organizzazione in layer

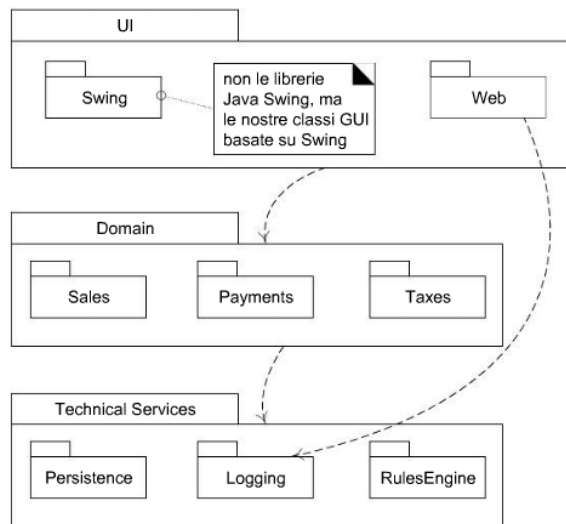
**Architettura:** la progettazione di un tipico sistema orientato agli oggetti è basata su diversi strati architetturali, come uno strato dell'interfaccia utente, uno strato della logica applicativa (o "del dominio") e così via.

L'architettura logica può essere illustrata sotto forma di diagrammi dei package di UML.

**Architettura logica:** è la macro-organizzazione su larga scala delle classi software in package (o namespace), sottoinsiemi e strati. È chiamata architettura logica poiché non vengono prese decisioni su come questi elementi siano distribuiti sui processi o sui diversi computer fisici di una rete.

**Layer o strato:** gruppo di classi software, packages, sottosistemi con responsabilità condivisa su un aspetto importante del sistema. Gli strati di un'applicazione software comprendono normalmente:

- **User interface** (interfaccia utente o presentazione): 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** (servizi tecnici ): oggetti e sottosistemi d'uso generale che forniscono servizi tecnici di supporto.



**Architettura a strati:** 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.

**Il concetto di “accoppiamento”:** nel contesto dell'architettura software, l'**accoppiamento** si riferisce al **grado di dipendenza** tra i componenti o le classi di un sistema. Indica quanto un componente dipenda da altri componenti per poter funzionare correttamente. Più alto è l'accoppiamento, più forte è la dipendenza tra i componenti e più difficile diventa modificarli o sostituirli in modo indipendente. **Un basso accoppiamento è un obiettivo desiderabile nell'architettura software** perché consente di ottenere una maggiore modularità, facilitando la manutenzione, il riuso del codice e la sostituibilità dei componenti. Inoltre, una bassa dipendenza tra i componenti aumenta la chiarezza e la comprensibilità del sistema.

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

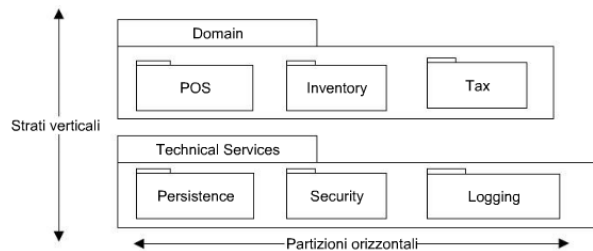
**Alta coesione:** in uno strato le responsabilità degli oggetti devono essere fortemente correlate, l'uno all'altro, e non devono essere mischiate con le responsabilità degli altri strati. Ad esempio, gli oggetti dell'interfaccia utente non devono implementare logica applicativa. Aumentare la coesione aumenta la possibilità di **riuso**, facilita la **manutenzione** e aumenta la **chiarezza**.

**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 a 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 (es. un oggetto Sale è in grado di calcolare il suo totale).

Lo strato di dominio fa riferimento al modello di dominio per trarre ispirazione per i nomi delle classi dello strato del dominio. Creando uno strato del dominio ispirandosi al modello di dominio, si ottiene un “salto rappresentazionale basso” tra dominio del mondo reale e il progetto software.

La nozione originaria di livello è di strato logico. Si dice che gli strati di un’architettura rappresentano le sezioni verticali, mentre le partizioni rappresentano una divisione orizzontale di sottosistemi relativamente paralleli di uno strato.



## Principio di separazione **Modello-Vista**

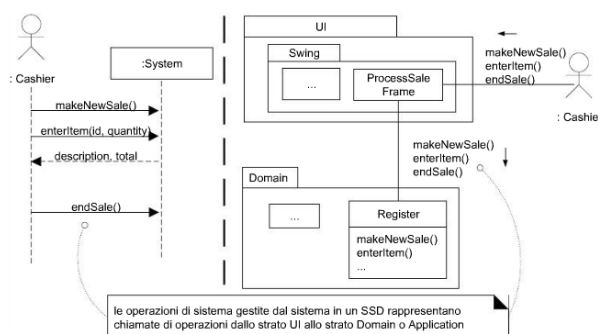
1. **Non relazionare o accoppiare oggetti non UI con oggetti UI:** gli oggetti non UI non devono essere connessi o accoppiati direttamente agli oggetti UI (es. non si deve permettere a un oggetto software di “dominio”, non UI, di avere un riferimento a un oggetto UI);
2. **Non incapsulare la logica dell’applicazione in metodi di UI:** non mettere logica applicativa (es. 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.

I concetti da considerare sono i seguenti:

- **Modello:** sinonimo per lo strato degli oggetti del dominio;
- **Vista:** sinonimo per gli oggetti dell’interfaccia utente (finestre, pagine web, ...).

Gli oggetti del **modello** (dominio) non devono avere una conoscenza diretta degli oggetti della vista (UI), almeno in quanto oggetti della vista. È un principio fondamentale del pattern **Model-View-Controller (MVC)**.

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.



## Capitolo 9 - Verso la progettazione ad oggetti: modellazione

- **Requisiti:** “fare la cosa giusta”. Consiste nel capire alcuni degli obiettivi preminenti per i caso di studio e le relative regole e vincoli;
- **Progettazione:** “fare la cosa bene”. Consiste nel progettare abilmente una soluzione che soddisfa i requisiti per l’iterazione corrente.

Come progettare a oggetti?

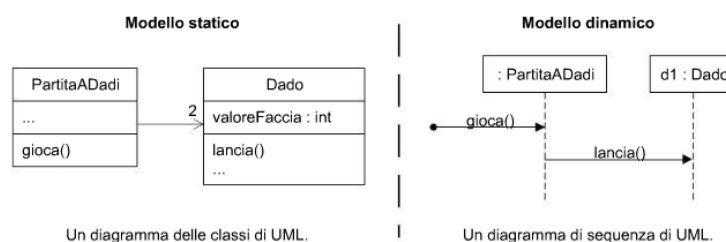
- **Codifica:** progettare mentre si codifica;
- **Disegno, poi codifica:** disegnare alcuni diagrammi UML, poi passare alla codifica;
- **Solo disegno:** lo strumento genera ogni cosa dai diagrammi.

**Disegno leggero:** il costo aggiuntivo dovuto al disegno dovrebbe ripagare lo sforzo impiegato.

**Modellazione agile:** ridurre il costo aggiuntivo del disegno e modellare per comprendere e comunicare, anziché per documentare.

Ci sono due tipi di modelli per gli oggetti:

- **Dinamici:** rappresentano il comportamento del sistema, la collaborazione tra oggetti software per realizzare (uno/più scenari di) un caso d’uso, i metodi di classi software;
- **Statici:** servono per definire i package, i nomi delle classi, gli attributi, le firme di operazioni.



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. Durante la modellazione dinamica si applica la progettazione guidata dalle **responsabilità** e i principi **GRASP**.

La **progettazione a oggetti** richiede soprattutto la conoscenza di:

- principi di **assegnazione di responsabilità**;
- **design pattern**.

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

- Un'interazione è motivata dalla necessità di eseguire un determinato **compito**;
- Un **compito** è rappresentato da un **messaggio** che dà inizio all'interazione (messaggio trovato);
- Il **messaggio** è inviato a un oggetto designato come **responsabile** per questo compito.

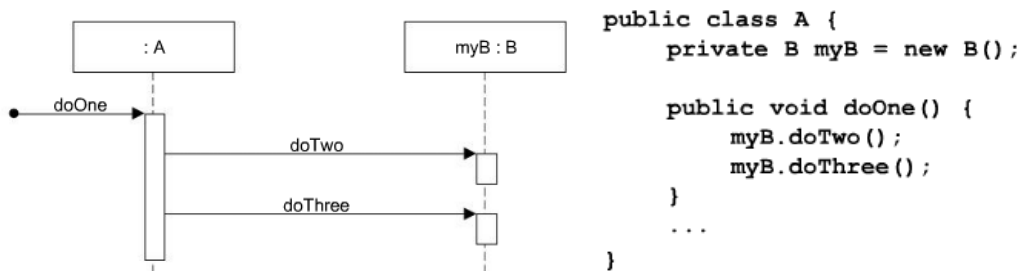


- L'**oggetto responsabile** collabora/interagisce con altri oggetti (**partecipanti**) per svolgere il compito;
- Ciascun **partecipante** svolge un proprio **ruolo** nell'ambito della **collaborazione**;
- La **collaborazione** avviene mediante **scambio di messaggi (interazione)**;
- Ciascun **messaggio** è una richiesta che un oggetto fa a un altro oggetto di eseguire un'**operazione**.

## Diagrammi di sequenza

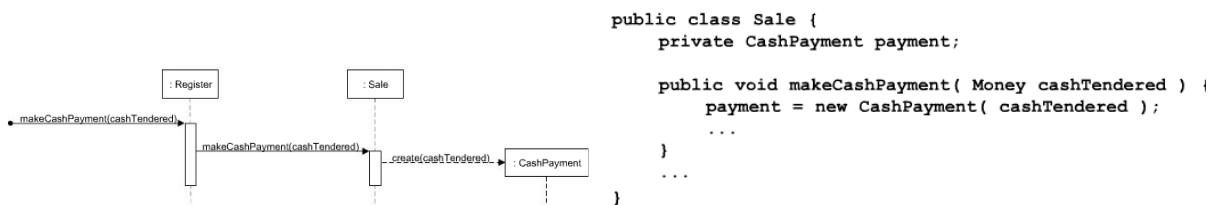
Mostrano le interazioni in una specie di formato a steccato, in cui gli oggetti che partecipano all'interazione sono mostrati in alto, uno a fianco all'altro.

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



Un **esempio** è il seguente:

1. Il messaggio `makeCashPayment` viene inviato a un'istanza di `Register`. Il mittente non è identificato;
2. L'istanza di `Register` invia il messaggio `makeCashPayment` a un'istanza di `Sale`;
3. L'istanza di `Sale` crea un'istanza di `CashPayment`.



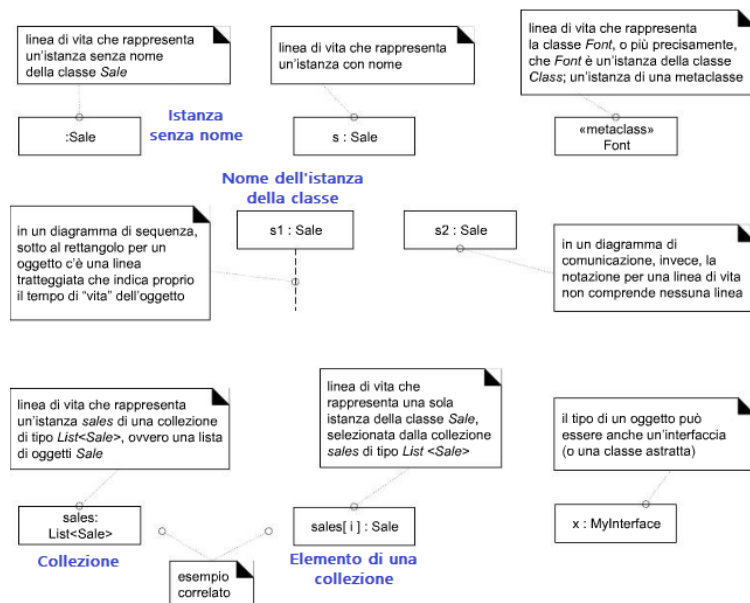
## Diagrammi di sequenza in UP: **DSD (Design Sequence Diagram)**

Il diagramma di sequenza 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.

### Componenti dei DSD: **partecipanti**

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



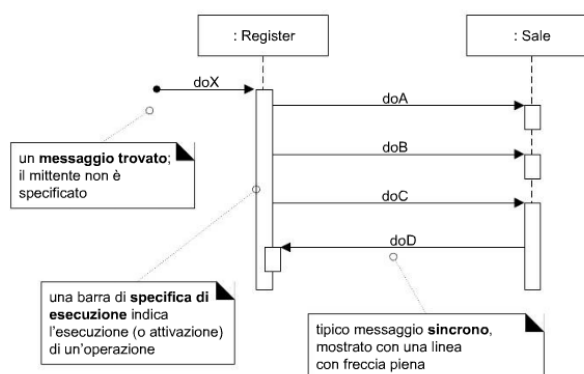
## Componenti DSD: **singleton**

Pattern nel quale da una classe viene istanziata una sola istanza, mai due o più.



## Componenti DSD: **linee di vita e messaggi**

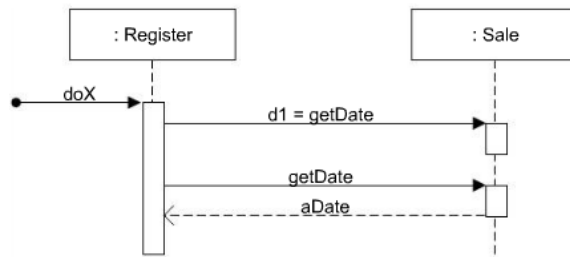
- **Linee di vita:** rettangolo + linea verticale sottostante;
- **Messaggio sincrono:** linea continua con freccia piena tra le linee di vita verticali;
- **Messaggio asincrono:** freccia non piena;
- Messaggio iniziale: **messaggio trovato**;
- Barra di **specifica di esecuzione:** esecuzione di un'operazione da parte di un oggetto.



## Componenti DSD: **risposte o ritorni**

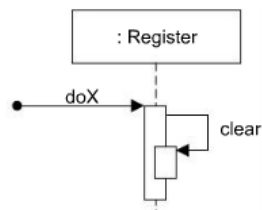
Ci sono due modi per mostrare il risultato di ritorno:

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



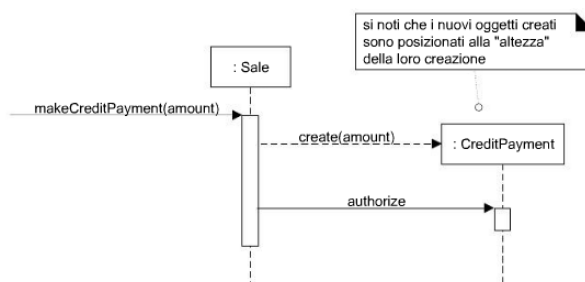
### Componenti DSD: **self o this**

Messaggio inviato da un oggetto a sé stesso utilizzando una barra di specifica di esecuzione annidata.



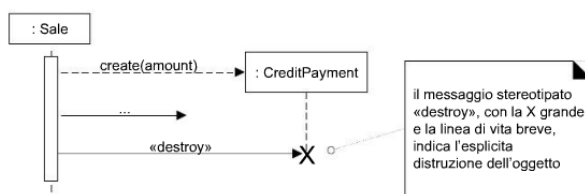
### Componenti DSD: **creazione di istanze**

I nuovi oggetti creati sono posizionati all'altezza della loro creazione.



### Componenti DSD: **distruzione di oggetti**

Una distruzione implicita di oggetti.

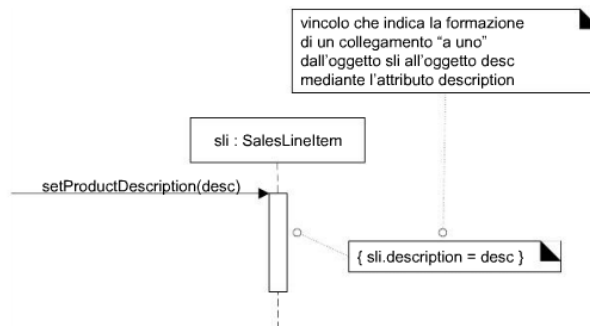


### Componenti DSD: **formazione di collegamenti**

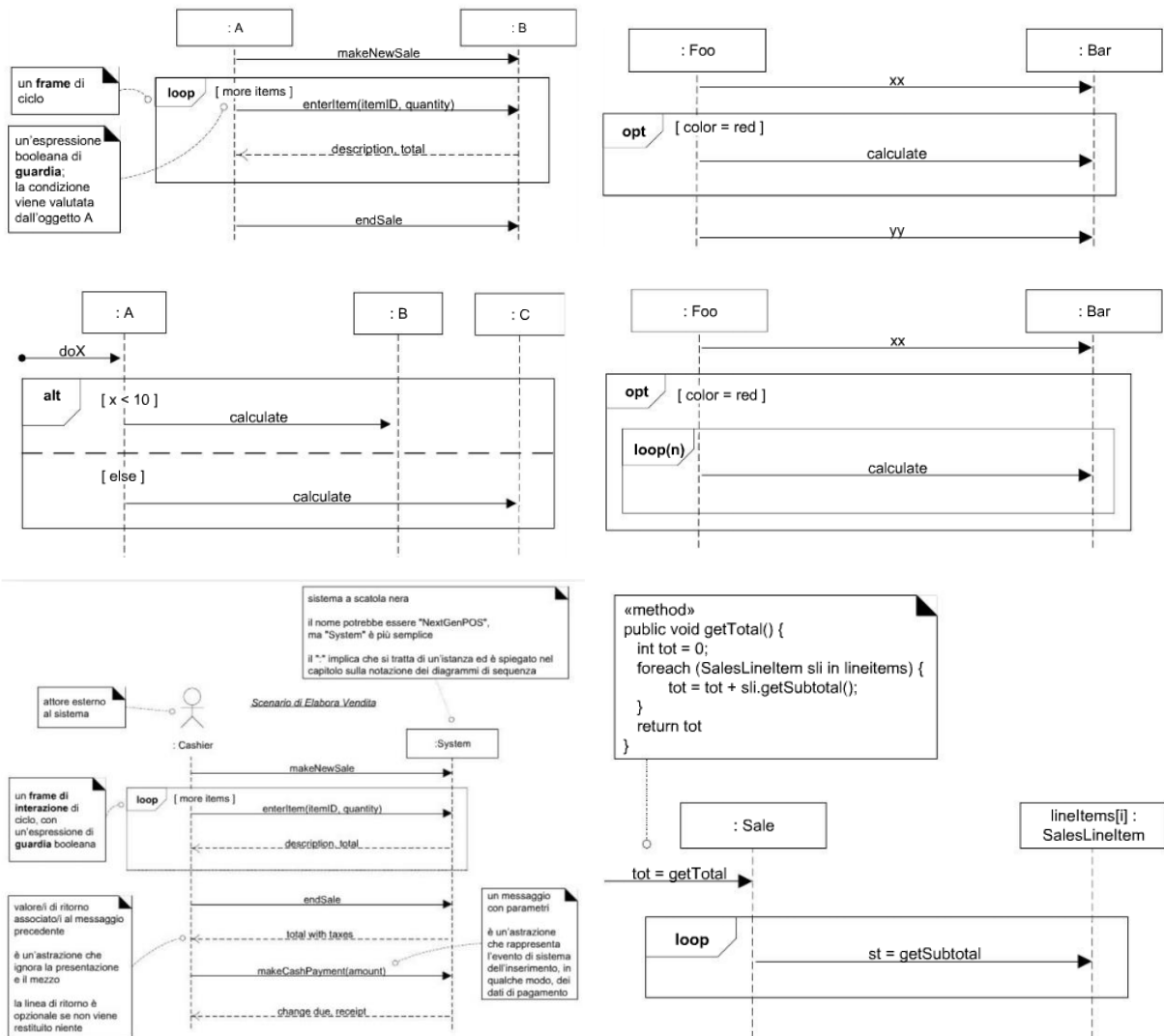
Formazione di un collegamento di un'associazione "a molti".



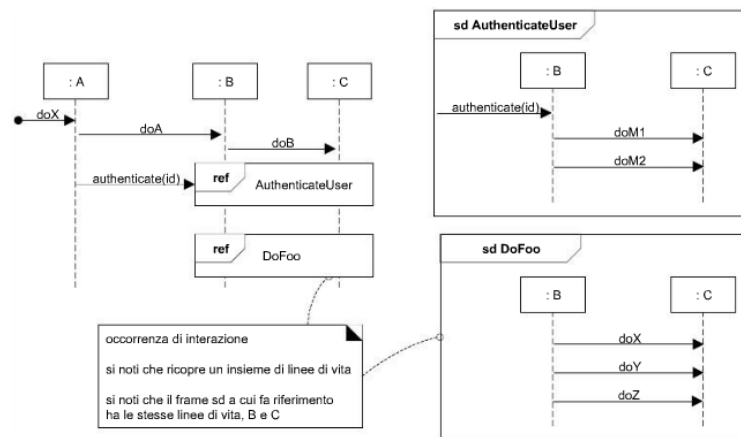
Formazione di un collegamento di un'associazione "a uno".



## Componenti DSD: frame

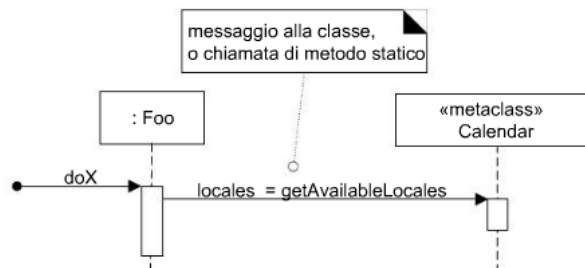


## Componenti DSD: correlare diagrammi di interazione



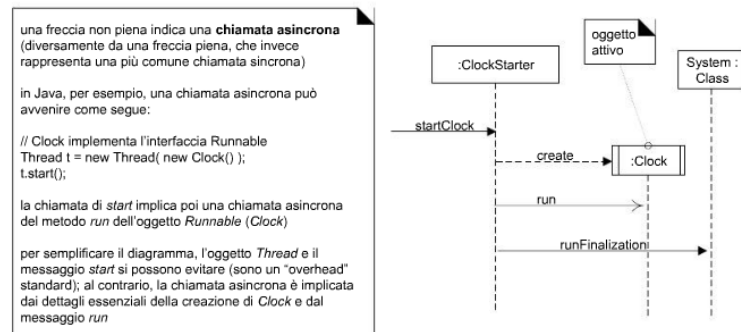
## Componenti DSD: **invocare metodi statici**

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



## Componenti DSD: **chiamate sincrone e asincrone**

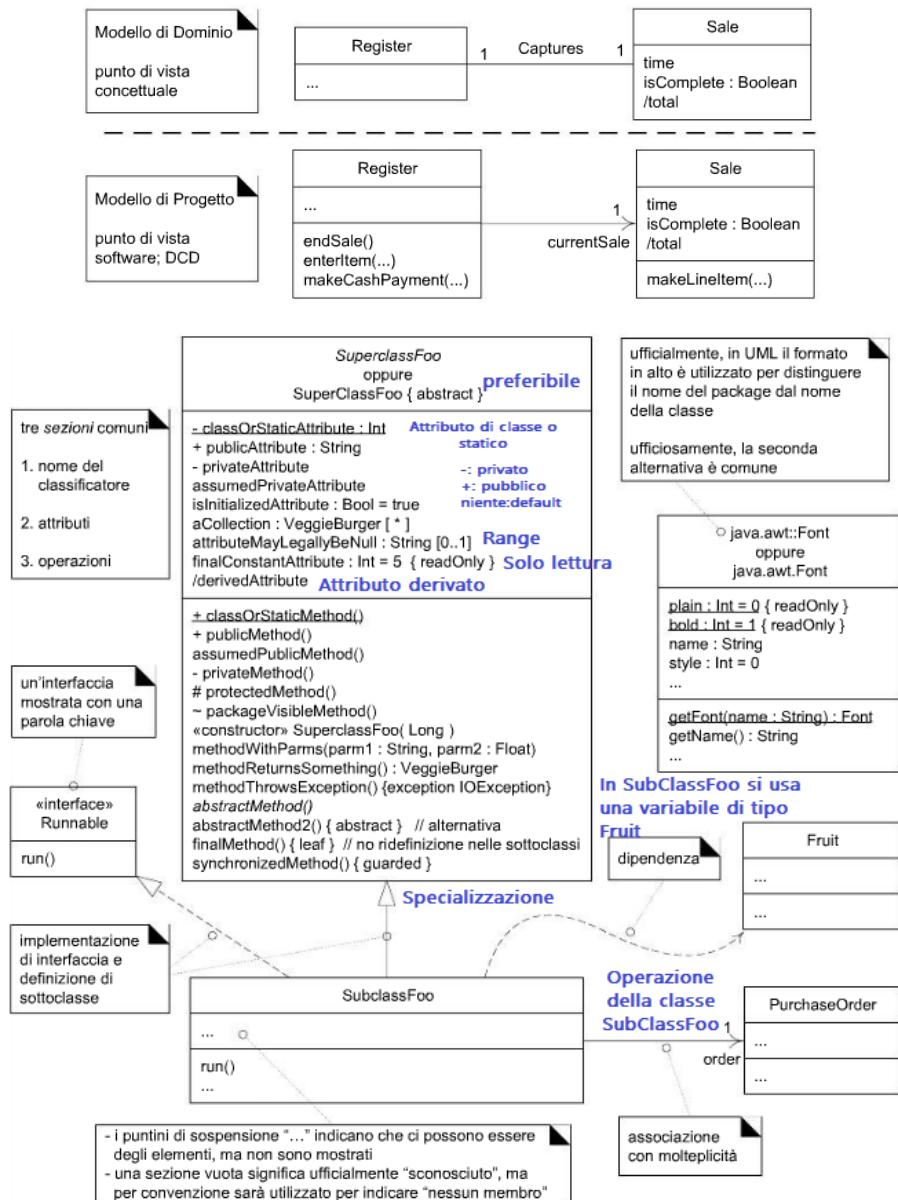
Oggetto attivo: ciascuna istanza è eseguita nel proprio thread di esecuzione e lo controlla.



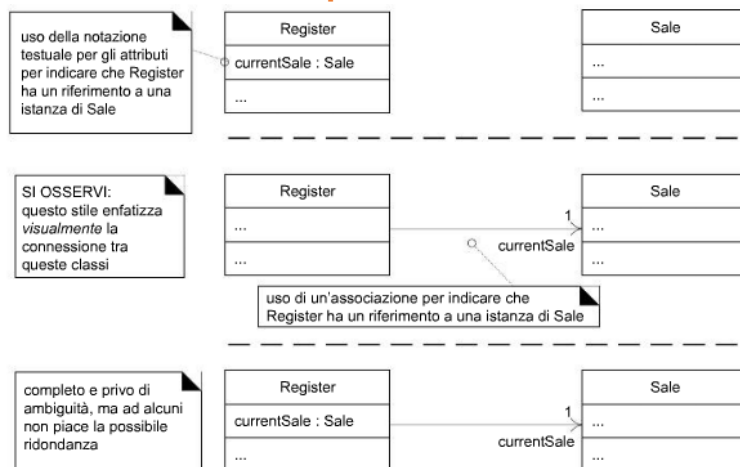
## Diagrammi di sequenza in UP: **DCD (Design Class Diagram)**

Il diagramma delle classi di progetto 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.



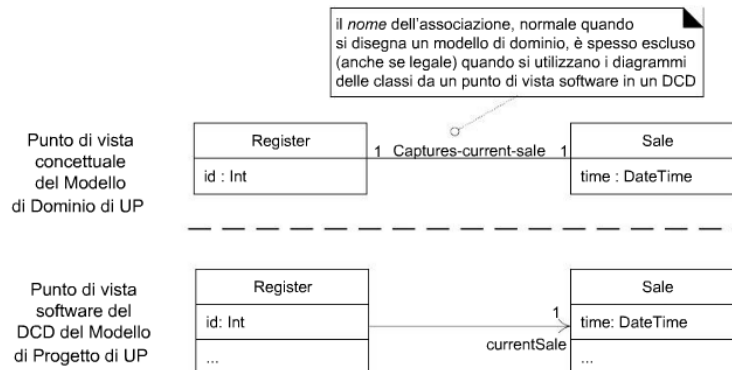
## Diagrammi delle classi: notazione per attributi come associazioni



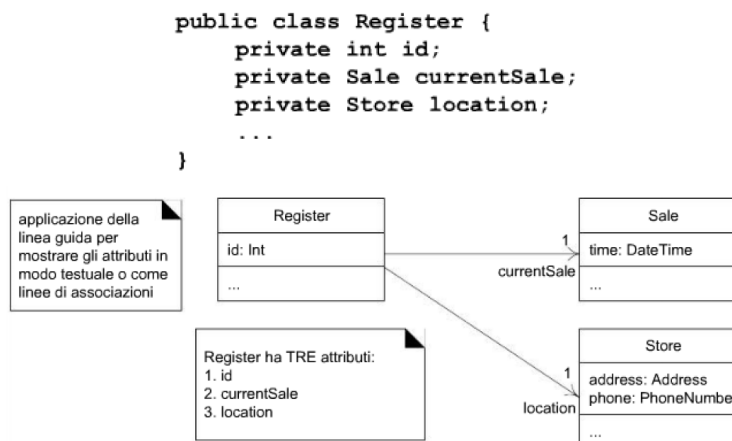
- Se non viene indicata alcuna visibilità, solitamente si ipotizza che gli attributi siano privati;

- Una freccia di navigabilità rivolta dalla classe sorgente alla classe destinazione dell'associazione, che indica che un oggetto della classe sorgente ha un attributo di tipo della classe destinazione;
- Una molteplicità all'estremità vicina alla destinazione, ma non all'estremità vicina alla sorgente;
- Un nome di ruolo solo all'estremità vicina alla destinazione, per indicare il nome dell'attributo;
- Nessun nome per l'associazione.

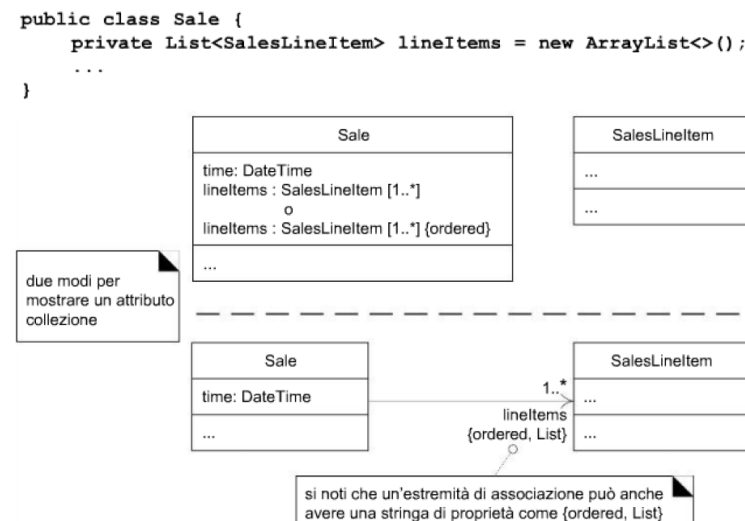
## Diagrammi delle classi: **modello di dominio VS DCD**



## Diagrammi delle classi: **associazioni e attributi**

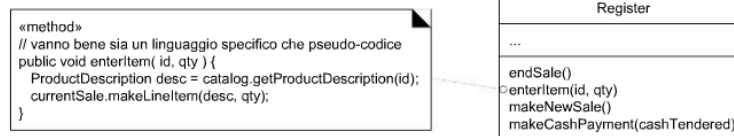


## Diagrammi delle classi: **attributi collezione e note**



## Diagrammi delle classi: operazioni e metodi

- Un'operazione una dichiarazione di un metodo, sintassi:  
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.



## Diagrammi delle classi: parole chiave

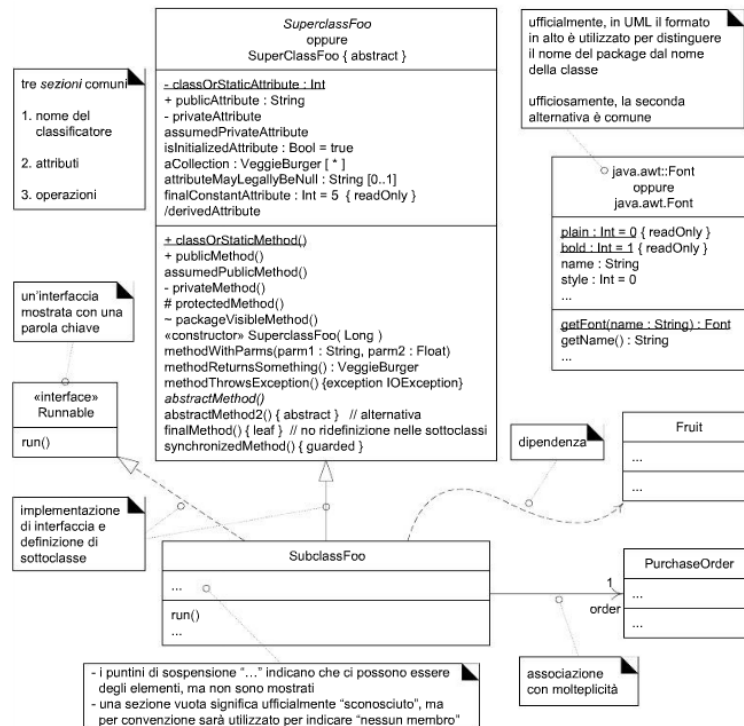
Decoratori testuali per classificare un elemento di un modello.

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 nome di un'operazione	nei diagrammi delle classi, dopo il nome di un classificatore o il
{ordered}	un insieme di oggetti ha un ordine predefinito	nei diagrammi delle classi, a un'estremità di associazione

## Diagrammi delle classi: generalizzazione

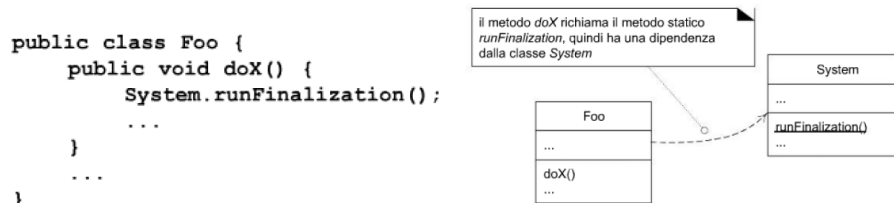
- È una **relazione tassonomica** tra un classificatore più generale e un classificatore più specifico. Ogni istanza del classificatore più specifico è anche un'istanza indiretta del classificatore più generale. Pertanto, il classificatore più specifico possiede indirettamente le caratteristiche del classificatore più generale;
- La generalizzazione implica l'**ereditarietà** nei linguaggi OO;
- Uso del tag {abstract} per le **classi astratte**.





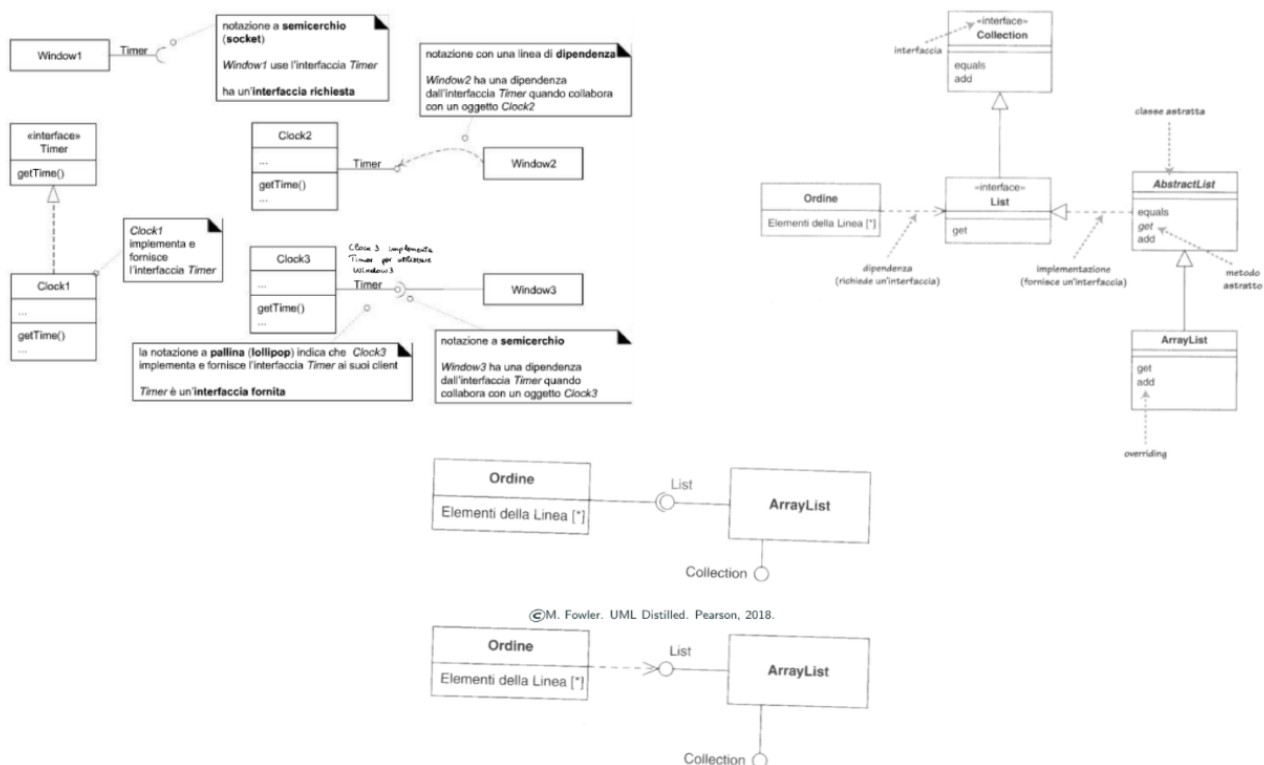
## Diagrammi delle classi: **dipendenze**

- Avere un attributo del tipo del 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, o 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.



## Diagrammi delle classi: **interfacce**

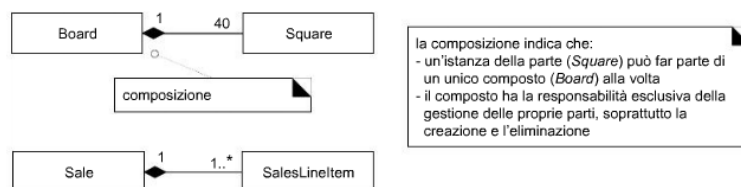
- L'implementazione di un'interfaccia viene chiamata una realizzazione di interfaccia;
- La notazione a pallina (**lollipop**) indica che una classe X implementa (fornisce) un'interfaccia Y, senza disegnare il rettangolo per l'interfaccia Y;
- La notazione a semicerchio (**socket**) indica che una classe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



## Diagrammi delle classi: **composizione**

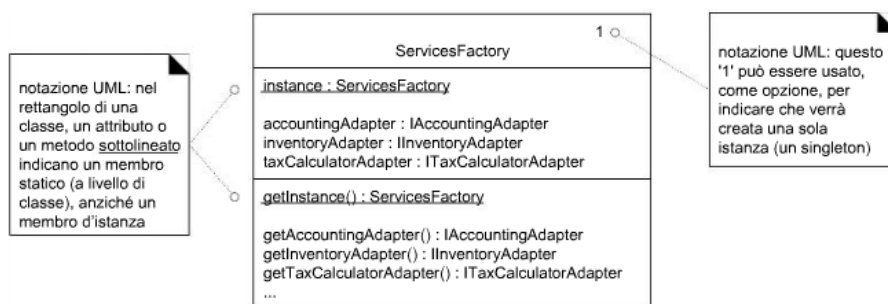
Una possibile interpretazione di una composizione tra le classi A e B è la seguente:

- gli oggetti B non possono esistere indipendentemente da un oggetto A;
- l'oggetto A è responsabile della creazione e distruzione dei suoi oggetti B.



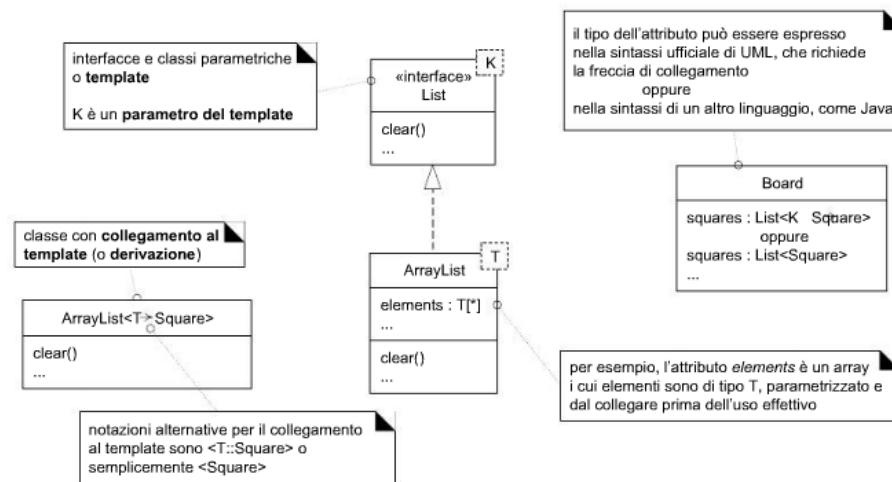
## Diagrammi delle classi: **singleton**

Esiste una sola istanza della classe, mai due.

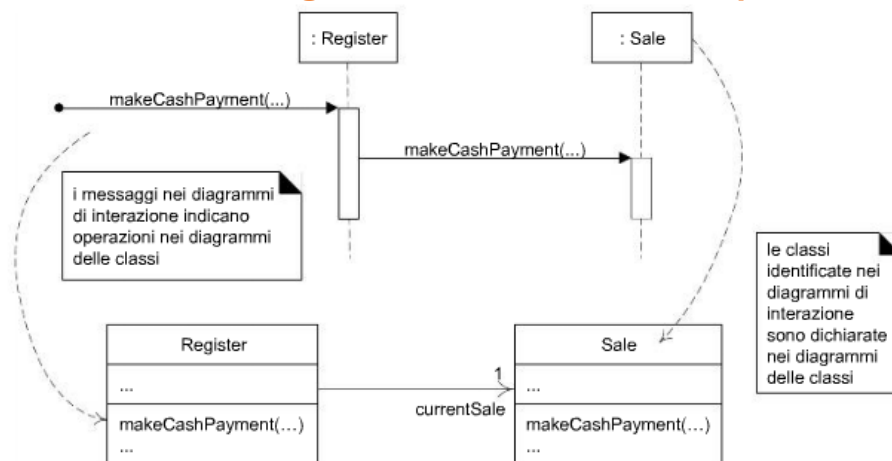


## Diagrammi delle classi: **template**

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



## Diagrammi delle classi e diagrammi di interazione/sequenza



## Capitolo 10 - GRASP (General Responsibility Assignment Software Patterns)

Finora abbiamo:

- Identificato i requisiti;
- Creato il modello di dominio.

Adesso bisogna:

- Aggiungere i metodi alle classe appropriate;
- Definire i messaggi fra gli oggetti per soddisfare i requisiti.

Principi di progettazione OO da applicare:

- Pattern **GRASP** (Schemi Generali per l'Assegnazione di Responsabilità nel Software);
- Design pattern **GoF** (Gang-of-Four).

## RDD (Responsability-Driven Development)

L'approccio complessivo al fare la modellazione per la progettazione OO si baserà sulla metafora della progettazione guidata dalle responsabilità (**RDD**), ovvero pensare a come assegnare le responsabilità a degli oggetti che collaborano. Si pensa alla progettazione di

oggetti software in termini di responsabilità, ruoli e collaborazioni. Gli **oggetti software sono considerati come dotati di responsabilità**.

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

Le responsabilità sono di due tipi:

- **Responsabilità di fare:**
  - Fare qualcosa l'oggetto stesso (creare un oggetto, eseguire un calcolo, ...);
  - Chiedere ad altri oggetti di eseguire azioni;
  - Controllare e coordinare le attività di altri oggetti;
- **Responsabilità di conoscere:**
  - Conoscere i propri dati privati incapsulati;
  - Conoscere gli oggetti correlati;
  - Conoscere cose che può derivare o calcolare.

Un **esempio** è il seguente:

- **Responsabilità di fare:** dichiarare che *“una Sale (vendita) è responsabile della creazione di oggetti SaleLineItems (articoli in vendita)”*;
- **Responsabilità di conoscere:** dichiarare che *“una Sale è responsabile di conoscere il suo totale”*.

## Passi della RDD

La progettazione guidata dalle responsabilità viene fatta **iterativamente** così:

- **Identifica le responsabilità e considerale** una alla volta;
- Chiediti **a quale oggetto software assegnare questa responsabilità**, potrebbe essere un oggetto già identificato oppure nuovo;
- Chiediti **come fa l'oggetto scelto a soddisfare questa responsabilità**, potrebbe fare tutto da solo oppure collaborare con altri oggetti.

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

## GRASP

I pattern **GRASP** sono uno strumento d'aiuto per acquisire la padronanza delle basi dell'OOD e a comprendere l'assegnazione di responsabilità nella progettazione a oggetti.

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

**Pattern:** descrizione con nome di un problema di progettazione ricorrente e di una sua soluzione ben provata che può essere applicata a nuovi contesti.

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 sull'assegnazione di responsabilità;
  - **GoF:** per idee di progettazione più avanzate;

I pattern possono essere applicati sia durante la modellazione che durante la codifica

- **UML** per la **modellazione visuale** per la progettazione OO, nel corso della quale possono essere applicati sia i pattern GRASP che quello GoF.

**LRG (Low Representational Gap):** nella fase di progettazione vale sempre il principio LRG, o salto rappresentazionale basso, tra il modo in cui si pensa al dominio e una corrispondenza diretta con gli oggetti software.

## GRASP Patterns

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.

**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**) **coesi** e **debolmente accoppiati**.

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

I **nove** pattern GRASP (dei quali ne verranno analizzati solamente 5) sono:

1. **Creator;**
2. **Information Expert;**
3. **Low Coupling;**
4. **Controller;**
5. **High Cohesion;**
6. Polymorphism;
7. Pure Fabrication;
8. Indirection;
9. Protected Variations.

### GRASP Pattern **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 (più sono vere meglio è):

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

### GRASP Pattern **Information 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, ...

## **GRASP Pattern** **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.

L'accoppiamento (coupling) è una misura di quanto fortemente un elemento è connesso ad altri elementi, ha conoscenza di altri elementi e dipende da altri elementi.

Una classe con un accoppiamento alto dipende da molte altre classi. Tali classi fortemente accoppiate possono essere inopportune e presentare o seguenti problemi:

- I cambiamenti nelle classi correlate obbligano a cambiamenti locali anche in queste classi;
- Queste classi sono più difficili da comprendere 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.

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

- 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 o 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 sottoclasse 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. Il problema, 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.

## GRASP Pattern **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:

1. 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);
2. 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)

**Controller come 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.

Il controller consente di progettare gli oggetti di dominio in modo indipendente dagli oggetti dell’interfaccia utente che potrebbero interagire con essi.

**N.B.:** 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 High Cohesion.

## GRASP Pattern **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.

High Cohesion è un **principio di valutazione** per scegliere tra diverse alternative:

- **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**.

- **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 moderata:** una classe ha, da sola, responsabilità in poche aree diverse, che sono logicamente correlate al concetto rappresentato dalla classe, ma non l'una all'altra;
- **Coesione alta:** una classe ha responsabilità moderate in un'area funzionale e collabora con altre classi per svolgere i suoi compiti.

Vantaggi:

- Spesso sostiene Low Coupling;
- Migliora chiarezza, comprensione, manutenzione e riuso del software.

## Capitolo 11 - GoF (Gang-of-Four)

Sono degli schemi di progettazione avanzata. Ciascun design pattern descrive una soluzione progettuale comune a un problema di progettazione ricorrente.

Sono classificati in base al loro scopo:

- **Creazionale:** risolvono problematiche inerenti all'istanziamento degli oggetti;
- **Strutturale:** risolvono problematiche inerenti alla struttura delle classi e degli oggetti;
- **Comportamentale:** forniscono soluzione alle più comuni tipologie di interazione fra oggetti.

(**Composizione** > **Ereditarietà**), GoF preferisce la composizione rispetto all'ereditarietà tra classi perché aiuta a mantenere le classi incapsulate e coese.



- **Ereditarietà di classi:**
  - Definiamo un oggetto in termini di un altro;
  - Riuso **white-box**: la visibilità della super classe è la visibilità della sottoclasse;
- **Composizione di oggetti:**
  - Le funzionalità sono ottenute assemblando o componendo gli oggetti per avere funzionalità più complesse;
  - Riuso **black-box**: i dettagli interni non sono conosciuti.

**N.B.:** Il riuso **black-box** è preferibile al riuso **white-box**.

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.

**N.B.:** Il **meccanismo di delega** è preferibile al **meccanismo di specializzazione** per il riuso del codice.

Il **meccanismo di ereditarietà** può essere utilizzato in due modi diversi:

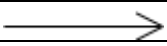
- **Polimorfismo**: le sottoclassi possono essere scambiate una per l'altra, possono essere "castate" in base al loro tipo, nascondendo il loro effettivo tipo alle classi cliente;
- **Specializzazione**: le sottoclassi guadagnano elementi e proprietà rispetto la classe base, creando versioni specializzare rispetto alla classe base.

I **pattern GoF suggeriscono di diffidare della specializzazione**, la quasi totalità dei pattern utilizza l'ereditarietà per creare polimorfismo.

## GoF Patterns

- **Creazionali**: Abstract Factory e Singleton;
- **Strutturali**: Adapter, Composite e Decorator;
- **Comportamentali**: Observer, State, Strategy e Visitor.

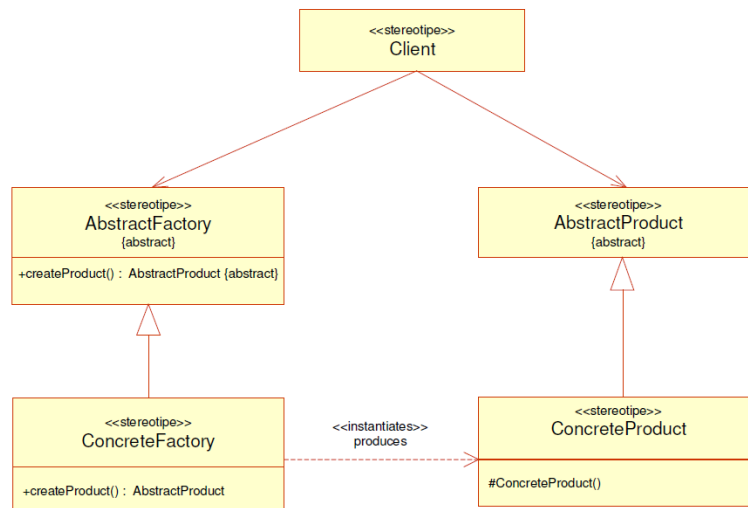
## Notazione dei GoF Patterns

Costrutto	Significato
	Implementa un'interfaccia
	Estende una classe
	Usa quella classe

## GoF Pattern Creazionali: **Abstract Factory**

Questo pattern è composto dai seguenti partecipanti:

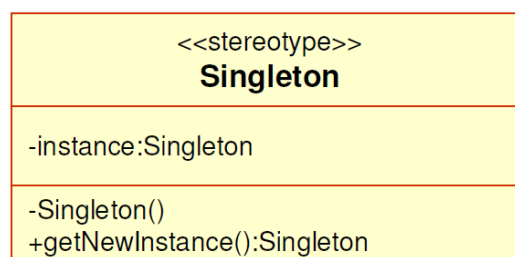
- **AbstractFactory**: interfaccia di esposizione di operazioni realizzate dai prodotti concreti;
- **ConcreteFactory**: implementazione delle operazioni per la creazione degli oggetti dei prodotti;
- **AbstractProduct**: interfaccia di esposizione delle operazioni dei prodotti concreti;
- **ConcreteProduct**: implementazione delle operazioni dei prodotti concreti;
- **Client**: invocazione delle interfacce per la creazione dei prodotti.



## GoF Pattern Creazionali: Singleton

Questo pattern è composto dai seguenti partecipanti:

- **Singleton**: una classe che implementa il pattern Singleton. Di solito, la classe Singleton ha un costruttore privato per impedire la creazione di istanze tramite l'operatore new;
- **Instance**: è l'unico oggetto istanziato dalla classe Singleton e rappresenta l'istanza condivisa AbstractProduct: interfaccia di esposizione delle operazioni dei prodotti concreti;
- **Metodo statico di accesso**: un metodo statico fornito dalla classe Singleton per ottenere l'istanza unica. Questo metodo controlla se l'istanza è già stata creata e la restituisce. Se l'istanza non esiste, il metodo crea prima l'istanza e la restituisce.

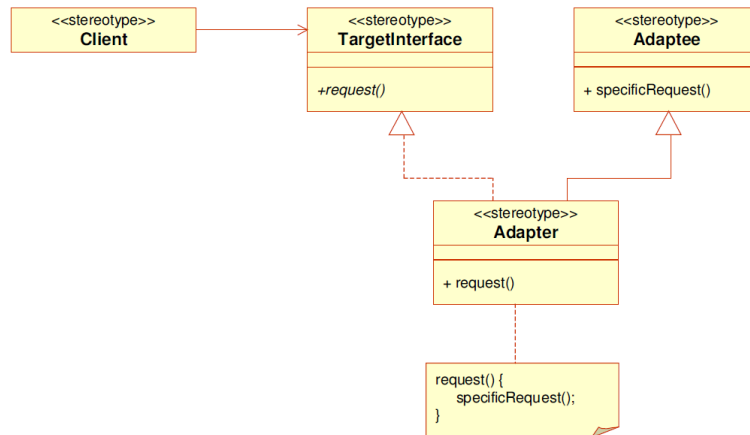


## GoF Pattern Strutturali: Adapter

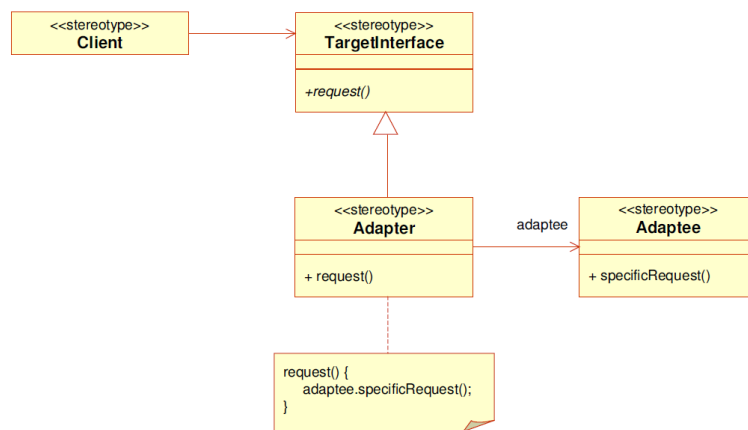
Questo pattern è composto dai seguenti partecipanti:

- **Client**: colui che effettua l'invocazione all'operazione di interesse;
- **TargetInterface**: definisce l'interfaccia specifica del dominio applicativo utilizzata dal Client;
- **Adaptee**: definisce l'interfaccia di un diverso dominio applicativo da dover adattare per l'invocazione da parte del Client;

- **Adapter:** definisce l'interfaccia compatibile con il Target che maschera l'invocazione dell'Adaptee.



Adapter (class)

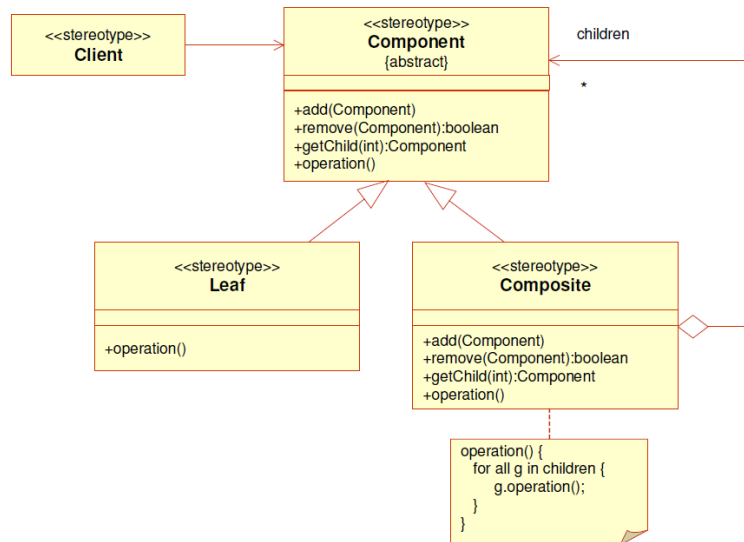


Adapter (object)

## GoF Pattern Strutturali: Composite

Questo pattern è composto dai seguenti partecipanti:

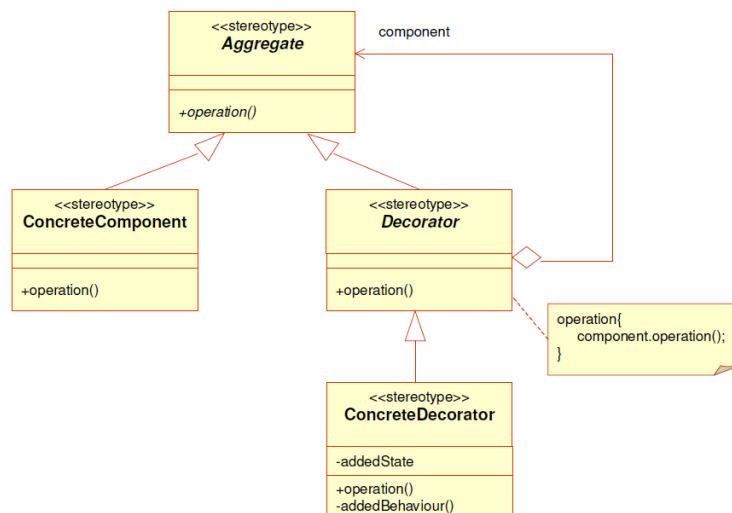
- **Client:** colui che effettua l'invocazione all'operazione di interesse;
- **Component:** definisce l'interfaccia degli oggetti della composizione;
- **Leaf:** rappresenta l'oggetto foglia della composizione. Non ha figli. Definisce il comportamento "primitivo" dell'oggetto della composizione;
- **Composite:** definisce il comportamento degli oggetti usati come contenitori e detiene il riferimento ai componenti "figli".



## GoF Pattern Strutturali: **Decorator**

Questo pattern è composto dai seguenti partecipanti:

- **Aggregate**: definisce l'interfaccia degli oggetti per i quali verranno aggiunte nuove funzionalità;
- **ConcreteComponent**: definisce un oggetto al quale verrà aggiunta una nuova funzionalità;
- **Decorator**: definisce l'interfaccia conforme all'interfaccia dell'Aggregate e mantiene l'associazione con l'oggetto Aggregate;
- **ConcreteDecorator**: implementa l'interfaccia Decorator al fine di aggiungere nuove funzionalità all'oggetto.

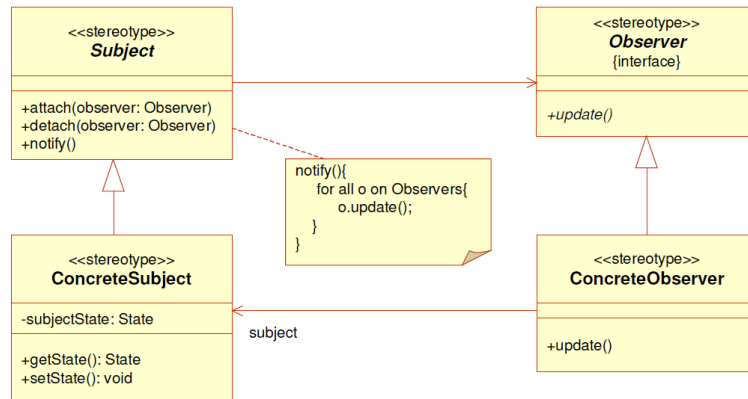


## GoF Pattern Comportamentali: **Observer**

Questo pattern è composto dai seguenti partecipanti:

- **Subject**: espone l'interfaccia che consente agli osservatori di iscriversi e cancellarsi; mantiene una reference a tutti gli osservatori iscritti;
- **Observer**: espone l'interfaccia che consente di aggiornare gli osservatori in caso di cambio di stato del soggetto osservato;

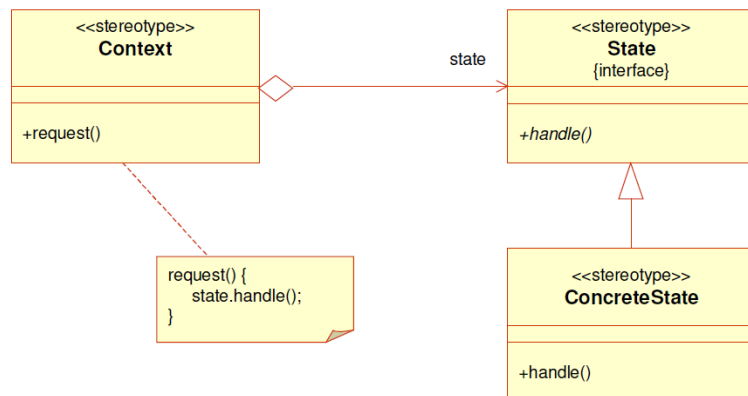
- **ConcreteSubject** : mantiene lo stato del soggetto osservato e notifica gli osservatori in caso di un cambio di stato;
- **ConcreteObserver**: implementa l'interfaccia dell'Observer definendo il comportamento in caso di cambio di stato del soggetto osservato.



## GoF Pattern Comportamentali: **State**

Questo pattern è composto dai seguenti partecipanti:

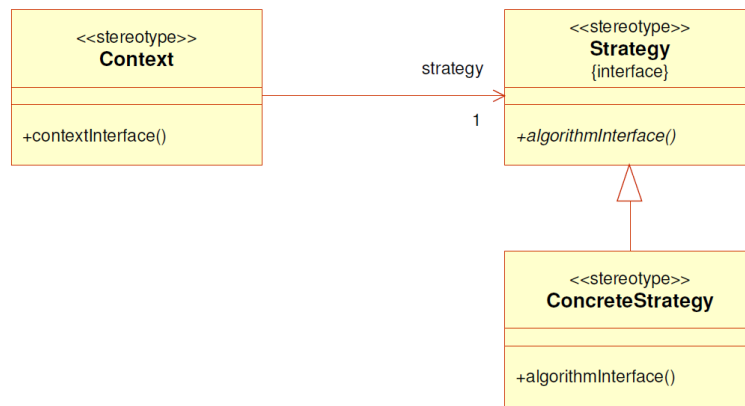
- **Context**: definisce l'interfaccia di interesse del Client e mantiene un'istanza della **ConcreteState** che definisce lo stato corrente;
- **State**: definisce un'interfaccia per incapsulare il comportamento associato con un particolare stato;
- **ConcreteState**: ogni sotto classe che implementa un comportamento associato con uno stato.



## GoF Pattern Comportamentali: **Strategy**

Questo pattern è composto dai seguenti partecipanti:

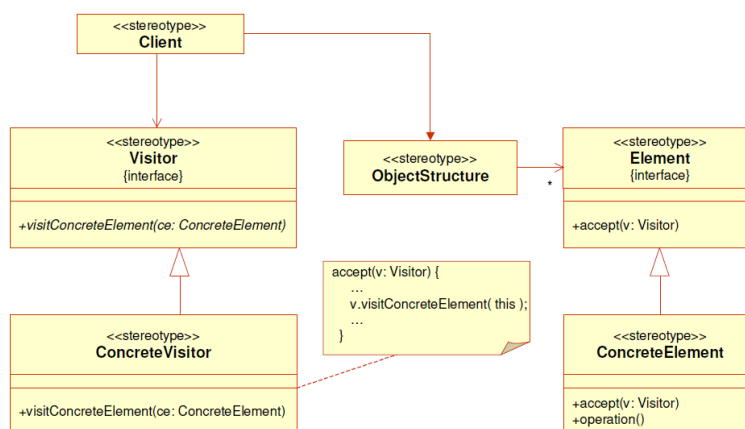
- **Strategy**: dichiara una interfaccia che verrà invocata dal Context in base all'algoritmo prescelto;
- **ConcreteStrategy**: effettua overwrite del metodo del Context al fine di ritornare l'implementazione dell'algoritmo;
- **Context**: detiene le informazioni di contesto (dati ed algoritmo da utilizzare) ed ha il compito di invocare l'algoritmo.



## GoF Pattern Comportamentali: **Visitor**

Questo pattern è composto dai seguenti partecipanti:

- **Element**: definisce il metodo `accept()` che prende un `Visitor` come argomento;
- **ConcreteElement**: implementa un oggetto `Element` che prende un `Visitor` come argomento;
- **ObjectStructure**: contiene una collezione di `Element` che può essere visitata dagli oggetti `Visitor`;
- **Visitor**: dichiara un metodo `visit()` per ogni `Element`; il nome del metodo ed il parametro identificano la classe `Element` che ha effettuato l'invocazione;
- **ConcreteVisitor**: implementa il metodo `visit()` e definisce l'algoritmo da applicare per l'`Element` passato come parametro.



## Capitolo 12 – **Dal progetto al codice**

Durante la programmazione ci si devono aspettare e si devono pianificare numerosi cambiamenti e deviazioni rispetto al progetto realizzato.

Questo è un atteggiamento **essenziale** e **pragmatico** nei metodi iterativi ed evolutivi.

### Sviluppo guidato dai test e refactoring

#### **XP** (Extreme Programming) e **TDD** (Test-Driven Development)

- **Extreme Programming (XP) e test**: Extreme Programming ha promosso la pratica dei test: scrivere i test per primi;

- **Extreme Programming (XP) e refactoring:** Extreme Programming ha inoltre promosso il refactoring continuo del codice per migliorare la qualità: meno duplicazioni, maggiore chiarezza, ecc.;
- **Test-Driven Development (TDD):** è una pratica promossa dal metodo iterativo e agile XP (applicabile a UP) è lo sviluppo guidato dai test, noto come sviluppo preceduto dai test.  
Il codice dei test è scritto **prima** del codice da verificare, immaginando che il codice da testare sia scritto.

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. Sono quindi utilizzati per dimostrare che il refactoring non abbia causato una regressione;
- **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.

Un **metodo di test unitario** è logicamente composto da 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:** opzionalmente rilascia o ripulisce gli oggetti e/o le risorse utilizzate nel test (per evitare che altri test vengano corrotti).

## Refactoring

Il **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. Il refactoring continuo del codice è un'altra pratica di XP **applicabile a tutti i metodi iterativi** (compreso UP).

L'**essenza del refactoring** è applicare piccole trasformazioni che preservano il comportamento.

**Refactoring e test:** 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.

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;

- altro...

## Ricordi? **Pillole di Java**

Qual è la differenza principale tra interfaccia e classe astratta?

- **Interfaccia:** viene implementata utilizzando la parola chiave “**implements**”. Tale keyword implica l’implementazione di tutti i metodi dichiarati nell’interfaccia;
- **Classe astratta:** viene estesa utilizzando la parola chiave “**extends**”. Tale keyword viene utilizzato per estendere una classe astratta ed ereditare i suoi membri, inclusi i metodi concreti e astratti.