

SVILUPPO DI APPLICAZIONI SOFTWARE

Indice

- 1) Processi per lo sviluppo software
- 2) UP (Unified Process)
- 3) Ideazione
- 4) Casi d'Uso
- 5) Elaborazione
- 6) Modello di Dominio
- 7) SSD (Diagrammi di Sequenza di Sistema))
- 8) Contratti
- 9) Architettura Logica
- 10) Modellazione Dinamica (Class Diagram DCD) e Dinamica (Diagrammi di Sequenza DSD) con UML
- 11) Pattern GRASP
- 12) Pattern GOF
- 13) Dal Progetto al Codice e Test

RIASSUNTO - SVILUPPO DI APPLICAZIONI SOFTWARE Object Oriented

SIGLE

- OOA/D = Object - Oriented Analysis/Design
- UML = Unified Modeling Language
- UP = Unified Process
- RAD = sviluppo rapido di applicazione
- XP = Extreme Programming
- UC = Use Cases (casi d'uso)
- SS = Specifiche Supplementari
- EBP = Elementary Business Process (processo di business elementare)
- SSD = System Sequence Diagrams (diagrammi di sequenza di sistema)
- DSD = Design Sequence Diagram
- DCD = Design Class Diagram
- GRASP = General Responsibility Assignment Software Patterns
- RDD = Responsibility - Driven Development
- LRG = Low Representational Gap (salto rappresentazionale basso)
- GoF = Gang-of-Four (banda dei quattro)
- TDD = Test Driven Development

PROCESSI PER LO SVILUPPO SOFTWARE

Progettazione orientata agli oggetti:

pone l'enfasi sulla definizione di oggetti software e del modo in cui questi collaborano per soddisfare i requisiti

Analisi orientata agli oggetti:

pone l'enfasi sull'identificazione e la descrizione degli oggetti, ovvero dei concetti nel dominio del problema

L'analisi dei requisiti e l'OOA/D vanno svolte nel contesto di qualche metodologia di sviluppo:

- ❖ **processo di sviluppo iterativo;**
- ❖ **Waterfall (a cascata);**
- ❖ **approccio agile;**
- ❖ **Unified Process.**

L'**UML** è un linguaggio visuale per la specifica, la costruzione e la documentazione degli elaborati di un sistema software ed è uno standard de fact per la notazione di diagrammi per disegnare o rappresentare figure relative al software, e in particolare al software OO.

Il suo utilizzo è dato da un:

- ❖ **punto di vista concettuale (modello di dominio):** la notazione dei diagrammi di UML è utilizzata per visualizzare concetti del mondo reale (classe concettuale);
- ❖ **punto di vista software (diagramma delle classi di progetto):** la notazione dei diagrammi delle classi di UML è utilizzata per visualizzare elementi software (classe software).

Processo: descrive chi fa che cosa, come e quando per raggiungere un obiettivo.

Processo per lo sviluppo del software (o processo software):

descrive un approccio disciplinato alla costruzione, al rilascio ed eventualmente alla manutenzione del software.

Esempi: processi a cascata, UP, Scrum, modello di sviluppo a spirale, RAD, XP.

Processi a cascata (o sequenziale):

è basato su uno svolgimento sequenziale delle diverse attività dello sviluppo del software:

- all'inizio del progetto vengono definiti in dettaglio tutti i requisiti e viene definito un piano temporale dettagliato delle attività da svolgere;
- si prosegue con la modellazione (analisi e progettazione) e viene creato un progetto completo del software;
- in seguito si inizia con la programmazione del sistema software e seguono verifica e rilascio (e successiva manutenzione) del prodotto realizzato.

Il metodo a cascata è una pratica mediocre per la maggior parte dei progetti software ed è caratterizzato da una minore produttività e da maggiori percentuali di difetti; le stime iniziali dei tempi e dei costi variano notevolmente dai valori finali.

Infatti le sue specifiche sono prevedibili e stabili e possono essere definite correttamente sin dall'inizio, a fronte di un basso tasso di cambiamenti.

Sviluppo iterativo ed evolutivo (o incrementale):

- comporta fin dall'inizio la programmazione e il test di un sistema software;
- comporta che lo sviluppo inizi prima che tutti i requisiti siano stati definiti in modo dettagliato;
- lo sviluppo è organizzato in una serie di mini-progetti brevi, di lunghezza fissa (timeboxed), chiamati iterazioni;
- ciascuna iterazione comprende le proprie attività di analisi dei requisiti, progettazione, implementazione e test;
- questo sviluppo è:
 - **incrementale:** il sistema cresce in modo incrementale nel tempo, iterazione dopo iterazione;
 - **evolutivo:** il feedback e l'adattamento fanno evolvere le specifiche e il progetto.
- il risultato di ogni iterazione è un sistema eseguibile, testato e integrato ma incompleto; infatti, esso non è un prototipo ma un sottoinsieme del sistema finale;
- ogni iterazione comporta la scelta di un piccolo sottoinsieme di requisiti, una rapida progettazione, implementazione e test, questo permette feedback rapidi da parte degli utenti, degli sviluppatori e dei test;
- attraverso il feedback iterativo e l'adattamento, il sistema evolve e converge verso i requisiti corretti e il progetto più appropriato;
- vantaggi:
 - riduzione precoce dei rischi maggiori (tecnicici, requisiti, obiettivi, ...);
 - progresso visibile dall'inizio;
 - feedback precoce, coinvolgimento dell'utente e adattamento;
 - gestione della complessità;

- la pianificazione è guidata sia dal rischio sia dal cliente:
 - le iterazioni iniziali vengono scelte per identificare e attenuare i rischi maggiori;
 - per costruire e rendere visibili le caratteristiche a cui il cliente tiene di più;
 - stabilizzare il nucleo dell'architettura del software;
 - **iterativa**: alla fine di ciascuna iterazione per quella successiva;
 - **adattiva**: all'inizio di ogni iterazione, per stabilire il piano dell'iterazione corrente.

Unified Process:

è un processo **iterativo ed evolutivo** (incrementale) per lo sviluppo del software per la costruzione di sistemi **orientati agli oggetti**. Le **iterazioni iniziali sono guidate dal rischio, dal cliente e dall'architettura**.

UP è flessibile e può essere applicato usando un approccio agile come XP e Scrum.

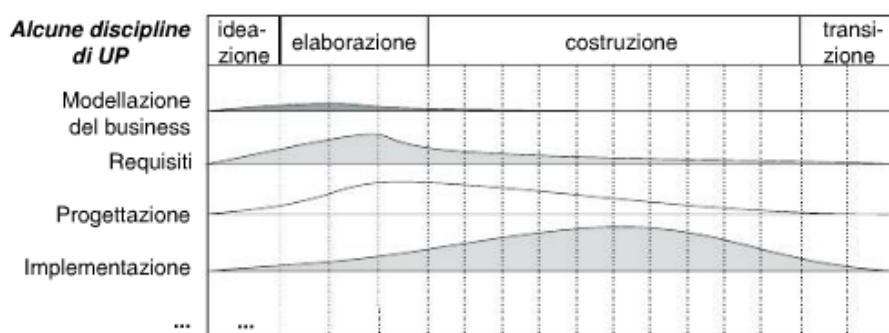
Un UP è composto da:

1. un'organizzazione del piano di progetto per fasi sequenziali;
2. indicazioni sulle attività da svolgere nell'ambito di discipline e sulle loro inter-relazioni;
3. un insieme di ruoli predefiniti;
4. un insieme di artefatti da produrre.

Un progetto UP organizza il lavoro e le iterazioni in 4 fasi:

- 1) **ideazione (inception)**: visione approssimativa, studio economico, portata, stime approssimative dei costi e dei tempi. In questa fase viene eseguita un'indagine sufficiente per decidere di proseguire con il progetto o di interromperlo;
- 2) **elaborazione (elaboration)**: visione raffinata, implementazione iterativa del nucleo dell'architettura, risoluzione dei rischi maggiori, identificazione della maggior parte dei requisiti e della portata, stime più realistiche sulle loro inter-relazioni. In questa fase viene implementata in modo iterativo l'architettura del sistema e vengono mitigati i rischi maggiori;
- 3) **costruzione (construction)**: implementazione iterativa degli elementi rimanenti, più facili e a rischio minore, preparazione al rilascio;
- 4) **transizione (transition)**: beta test, rilascio.

Milestone delle 4 fasi (dalla prima alla quarta): **obiettivi, architetturale, capacità operazionale, rilascio prodotto**.



L'impegno relativo nelle discipline cambia a seconda delle fasi.

Questo esempio è solo un suggerimento, non è da prendere alla lettera.

Le attività lavorative in UP si eseguono nell'ambito di discipline (Core Workflow).

Disciplina: è un insieme di attività e dei relativi elaborati in una determinata area, come le attività relative all'analisi dei requisiti.

Elaborato (o artefatti o work product, product in RUP):

è il termine generico che indica un qualsiasi prodotto di lavoro: codice, schemi di dati, documenti di testo, diagrammi, modelli, ecc.

Altre caratteristiche di UP:

- usa UML come linguaggio di modellazione;
- i diagrammi UML si usano con variabilità: si usano solo quando necessario ed è l'UP a dire quando utilizzarli;
- i diagrammi che si usano utilizzano le caratteristiche di interazione ed incremento;
- in UP quasi tutto (tra artefatti e pratiche) è opzionale, eccetto per lo sviluppo iterativo e guidato dal rischio, la verifica continua della qualità e naturalmente il codice;
- la scelta delle pratiche e artefatti UP per un progetto si riassume in un documento chiamato "scenario di sviluppo" (artefatto della disciplina infrastruttura).

ATTENZIONE!

- le fasi sono sequenziali e la fine di una fase corrisponde ad una milestone;
- le discipline (tipologie di attività) non sono sequenziali e si eseguono nel progetto in ogni iterazione;
- il numero di iterazioni dipende dalla decisione del manager di progetto e dai rischi del progetto;
- le iterazioni iniziali tendono in modo naturale a dare una maggiore enfasi relativa sui requisiti e sulla progettazione, mentre quelle successive lo fanno in misura minore. Questo perché i requisiti e il progetto si stabilizzano attraverso un processo di feedback e adattamento;
- 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, l'enfasi è maggiore sull'implementazione e minore sull'analisi dei requisiti.

Sviluppo agile: l'enfasi è data su una risposta rapida e flessibile ai cambiamenti (agilità): iterazioni brevi, raffinamento evolutivo dei piani, dei requisiti e del progetto.

Agile Modeling:

lo scopo della modellazione è principalmente quello di comprendere, di agevolare la comunicazione, non di documentare. Ossia, esplorare rapidamente le alternative e il percorso verso un buon progetto OO.

Pratiche:

- modellare insieme agli altri, modellazione di gruppo;
- creare diversi modelli in parallelo (sia dinamici che statici).

Sviluppo agile e Agile Modelling servono soprattutto per comprendere meglio tutto il lavoro fatto fino ad un certo punto.

Tabella 2.1 Scenario di Sviluppo di esempio (i – inizio; r – raffinamento).

Disciplina	Pratica	Elaborato Iterazione →	Ideazione I1	Elaboraz. E1..En	Costr. C1..Cn	Transiz. T1..T2
Modellazione del business	modellazione agile workshop requisiti	Modello di Dominio		i		
Requisiti	workshop requisiti esercizio sulla visione votazione a punti	Modello dei Casi d'Uso	i	r		
		Visione	i	r		
		Specifiche Supplementare	i	r		
		Glossario	i	r		
Progettazione	modellazione agile sviluppo guidato dai test	Modello di Progetto		i	r	
		Documento dell'Architettura Software		i		
		Modello dei Dati		i	r	
Implementazione	sviluppo guidato dai test programmazione a coppie integrazione continua standard di codifica	...				
Gestione del progetto	gestione del progetto agile riunioni Scrum giornaliere	...				
...						

IDEAZIONE

Requisito: è una capacità o una condizione a cui il sistema, e più in grande il progetto, deve essere conforme.

Sorgenti dei requisiti: i requisiti derivano da richieste degli utenti del sistema, per risolvere dei problemi e raggiungere degli obiettivi.

Tipi di requisiti:

- **requisiti funzionali:** i requisiti comportamentali descrivono il comportamento del sistema, in termini di funzionalità fornite ai suoi utenti;
- **requisiti non funzionali:** le proprietà del sistema nel suo complesso, come ad esempio sicurezza, prestazioni (tempo di risposta, throughput, uso di risorse), scalabilità, usabilità (fattori umani), ecc.

Le tipologie dei requisiti si basano sul **Modello FURPS+**, acronimo di:

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

In UP l'acquisizione dei requisiti deve essere abile e lo si fa attraverso:

- scrivere i casi d'uso con i clienti;
- workshop dei requisiti a cui partecipano sia sviluppatori che clienti;
- gruppi di lavoro con rappresentanti dei clienti;
- dimostrazione ai clienti dei risultati di ciascuna iterazione, per sollecitare il feedback.

UP ha diversi elaborati:

- **modello dei casi d'uso (principale):** scenari tipici dell'utilizzo di un sistema (requisiti funzionali, comportamento);
- **specifiche supplementari:** ciò che non rientra nei casi d'uso, requisiti non funzionali o funzionali non esprimibili attraverso casi d'uso (es. generazione di un report);
- **glossario:** termini significativi, dizionari dei dati (requisiti relativi ai dati, regole di validazione, valori accettabili);
- **visione:** riassume i requisiti ad alto livello, un documento sintetico per apprendere rapidamente le idee principali del progetto;
- **regole di Business:** regole di dominio, i requisiti o le politiche che trascendono un unico progetto software e a cui il sistema deve conformarsi (es. leggi fiscali dello stato);
- **lista dei rischi e piano di gestione:** descrive i rischi e le idee per attenuarli o rispondervi;
- **prototipi e proof-of-concept:** chiarire le visione e verificare le idee dell'elaborazione;

- **piano delle fasi e piano di sviluppo software:** ipotesi (poco precise) riguardo la durata e lo sforzo della fase di elaborazione;
- **scenario di sviluppo:** una descrizione della personalizzazione dei passi e degli elaborati UP per un progetto. In UP si effettua sempre una personalizzazione.

N.B.

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

Che cos'è l'**ideazione**?

L'ideazione permette di stabilire una visione comune e la portata del progetto (studio di fattibilità); infatti, il suo scopo non è quello di definire tutti i requisiti (che avviene soprattutto durante la fase di elaborazione, in parallelo alla prime fasi di programmazione), né di generare una stima o un piano di progetto affidabili.

Durante l'**ideazione**:

si tratta di decidere se il progetto merita un'indagine più seria (durante l'elaborazione), non di effettuare questa indagine. I passi che si effettuano sono:

- ❖ si analizzano circa il 10% dei casi d'uso in dettaglio e 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 (primo workshop dei requisiti e pianificazione della prima iterazione dell'elaborazione), generalmente 1-2 settimane.

Lo scopo della creazione degli elaborati e dei modelli è nel pensare:

- 1) si scelgono quelli che aggiungono valore al progetto;
- 2) si completano parzialmente;
- 3) sono preliminari ed approssimativi.

Specifiche Supplementari:

raccolgono altri requisiti, informazioni e vincoli che non sono facilmente colti dai casi d'uso o nel glossario, compresi gli attributi di qualità e i requisiti "URPS+" (usabilità, affidabilità, prestazioni, sostenibilità,...) a livello di intero sistema.

Due differenti documenti:

- **Visione:** riassume alcune delle informazioni contenute nel modello dei casi d'uso e nelle specifiche supplementari e descrive brevemente il progetto al fine di stabilire una visione comune del progetto;
- **Glossario:** è un elenco dei termini significativi e delle relative definizioni e include pseudonimi e termini composti. E' importante in quanto ci possono essere molti termini (tecnichi o specifici del dominio) che vengono usati in modo diverso. L'obiettivo è eliminare le discrepanze per ridurre i problemi di comunicazione e di ambiguità dei requisiti.

Struttura del Glossario:

Termine	Descrizione	Sinonimi
Sezione		

Dizionario dei dati: in UP, il glossario prende anche il ruolo di dizionario dei dati, un documento dati relativi ad altri dati (metadati).

Regole di dominio (o regole di business): stabiliscono come può funzionare un dominio o un business.

Tempi di realizzazione:

- 1) **Ideazione:** i documenti vengono abbozzati e definiti in maniera “leggera”;
- 2) **Elaborazione:** i documenti vengono raffinati, sulla base del feedback proveniente dalla costruzione incrementale di parti del sistema, dall'adattamento e dei vari workshop sui requisiti tenuti durante le iterazioni di sviluppo;
- 3) **Costruzione:** i requisiti principali, funzionali o meno, dovrebbero essere stabilizzati.

Congelamento e firma per accettazione:

stipulare un accordo tra le parti interessate su ciò che verrà fatto nel resto del progetto, e assumersi impegni (contrattuali) sui requisiti e sui tempi, si esegue alla fine dell'elaborazione.

CASI D'USO

Disciplina dei requisiti: processo per scoprire cosa deve essere costruito ed orientare lo sviluppo verso il sistema corretto.

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

Flusso delle attività in UP:

passi principali (non necessariamente eseguiti separatamente):

- produrre una lista dei requisiti potenziali;
- capire il contesto del sistema;
- catturare requisiti funzionali (di comportamento);
- catturare i requisiti non funzionali.

Lista dei requisiti: è usata anche per stimare la taglia del progetto e per decidere come suddividere il progetto in sequenze di iterazioni.

Ogni requisito è caratterizzato da: breve descrizione, stato (es. proposto, validato), costi di implementazione stimato, priorità, rischio associato per la sua implementazione.

Ci sono due approcci per capire il contesto del sistema:

- 1) **modellazione del dominio:** descrivere i concetti importanti del sistema come oggetti di dominio e relaziona i concetti con associazioni;
- 2) **modellazione del business:** come si sviluppano le varie attività
 - a) è un super-insieme del modello di dominio, descrivere i processi di business;
 - b) è un prodotto dell’ingegneria del business;
 - c) ha lo scopo di migliorare i processi di business;

Approccio UP-Agile:

- i requisiti funzionali sono catturati con i UC;
- se ci sono requisiti non funzionali relazionali, questi vengono inclusi nel caso d'uso;
- i requisiti non funzionali “generali” sono inclusi nel documento di SS; specifiche supplementari
- il contesto del sistema è catturato dal diagramma UML dei UC;
- tali artefatti (UC & SS) costituiscono l per definire il modello di dominio;
- non viene considerato il modello di business.

UP è una metodologia “UC driven”:

- i casi d'uso si utilizzano:
 - per descrivere i requisiti funzionali;
 - per pianificare le iterazioni;
- l'analisi e la progettazione si basano sulla realizzazione di casi d'uso;
- i test si basano sui casi d'uso realizzati;
- i casi d'uso influiscono nella redazione dei manuali utente e nella definizione della visione del progetto.

UC sono la parte portante di tutta la metodologia agile



Lavorare con gli UC in UP

	Identificazione UC	Descrizione dettagliata UC	Realizzazione UC
Ideazione	50/70%	10%	5%
Elaborazione	~100%	40/80%	meno del 10%
Costruzione	100%	100%	100%
Transizione			

I casi d'uso sono descrizioni (testuali) di scenari di uso interessanti del sistema software che si deve realizzare, mettono in risalto gli obiettivi degli utenti e il loro punto di vista e sono il meccanismo centrale per la scoperta e la definizione dei requisiti (funzionali).

Elementi coinvolti nei modelli dei dominio:

- **attori**: qualcosa o qualcuno dotato di un comportamento e sono ruoli svolti da persone, organizzazioni, software, macchine. Sono di tre tipi:
 - attore primario:
 - raggiunge gli obiettivi utente utilizzando i servizi del sistema;
 - utile per trovare gli obiettivi utente;
 - di supporto:
 - offre un servizio al sistema;
 - utile per chiarire le interfacce esterne e i protocolli;
 - fuori scena:
 - ha un interesse nel comportamento dei UC;
 - utile per garantire che tutti gli interessi necessari vengano soddisfatti;
- **scenario (o istanza di caso d'uso)**: 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.

ATTENZIONE!

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

Modello dei casi d'uso: modello delle funzionalità del sistema

- include un diagramma UML dei UC che serve come modello di contesto del sistema e come indice dei nome di UC;
- i UC non sono una pratica di OOA/D classica, però sono utili a rappresentare i requisiti come input all'OOA/D. In altre parole, i casi d'uso non sono orientati agli oggetti;
- i UC definiscono i contratti in relazione al comportamento del sistema.

Non dicono cosa deve fare il sistema ma gli obiettivi dal punto di vista dell'utente.

Non si descrive il sistema di per se ma si descrive come è utilizzato



I formati dei UC sono 3:

- **formato breve:** riepilogo conciso di un solo paragrafo, relativo al solo scenario principale di successo e serve a capire rapidamente l'argomento e la portata;
- **formato informale:** più paragrafi, scritti in modo informale, relativi a vari scenari e ha la stessa funzione del formato breve ma con maggiore dettaglio;
- **formato dettagliato:** tutti i passi e le variazioni sono scritti in dettaglio, include pre-condizioni e garanzie di successo e si scrive a partire dal formato breve o informale.

Come scrivere un caso d'uso

Ci sono tre elementi:

- 1) **preambolo:** è tutto ciò che precede lo scenario principale e le estensioni, i suoi elementi sono:
 - a) portata: descrive i confini del sistema in progettazione;
 - b) livello: tipicamente livello di obiettivo utente o livello di sottofunzione;
 - c) 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 coincidono;
 - d) parti interessate: elenco delle parti interessate, ossia chi ha interessi nel raggiungimento dell'obiettivo espresso dal caso d'uso in oggetto;
 - e) garanzie di successo (post-condizioni): che cosa deve essere vera quando è stato completato con successo il caso d'uso;
- 2) **scenario principale di successo (o percorso felice o flusso base o flusso tipico)**: è costituito da una sequenza di passi, che può contenere passi da ripetere più volte, ma che di solito non comprende alcuna condizione o diramazione.
- 3) **estensioni**: hanno lo scopo di descrivere tutti gli altri scenari oltre a quello principale, sia di successo che di fallimento e solitamente sono descritte per differenza dallo scenario principale. Gli scenari relativi alle estensioni sono diramazioni dello scenario principale di successo, e vengono indicati con riferimento ai suoi passi.

sono quasi tutte interazioni

Le estensioni sono costituite da due parti:

- 1) **condizione**: va scritta come qualcosa che possa essere rilevato dal sistema o da un attore;
- 2) **gestione**: può essere riassunta 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.

NON è un algoritmo, descrive solo l'interazione fino al successo

Non devo neanche descrivere come funziona l'interfaccia, perché è dinuovo un COME.
Noi vogliamo individuare i requisiti del sistema in cui l'interfaccia è una parte modulare che può cambiare, è il modo con cui il nostro software cattura gli eventi di interazione e li tramuta in chiamate al sistema.

L'attore genera un evento e noi stiamo descrivendo con gli UC questo evento, che viene poi catturato dall'interfaccia grafica ma a noi ora non interessa.

il formato "a due colonne" che enfatizza la conversazione tra gli attori e il sistema (responsabilità del sistema)

Esempio di descrizione di caso d'uso:

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 assegnandole un nome.	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.	

1 a) perché è la prima alternativa del passo 1

situazione alternativa e quindi i passi alternativi che vanno a fondersi nello scenario

Estensione 1a: lavora su un menu esistente

#	Attore	Sistema
1a.1	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
1b.1	<u>Crea una copia di un menù esistente</u>	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
3a.1	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
3b.1	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>	

alternative al passo 4
dello scenario principale

Estensione 4a: elimina una voce dal menù

#	Attore	Sistema
4a.1	Elimina una voce dalla sezione del menù.	Mostra la sezione aggiornata (senza la voce eliminata). Se sono state eliminate tutte le voci dalla sezione e il menù era indicato come completo, segnala che non lo è più.

Estensione 4b: modifica una voce del menù

#	Attore	Sistema
4b.1	Modifica una voce nella sezione del menù, indicando un nuovo testo o sostituendo la ricetta a cui si riferisce.	Mostra la sezione aggiornata (con la voce modificata)

Estensione 4c: cambia il nome di una sezione

#	Attore	Sistema
4c.1	Indica un nuovo nome per la sezione del menù.	Mostra la sezione aggiornata (con il nuovo nome)

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

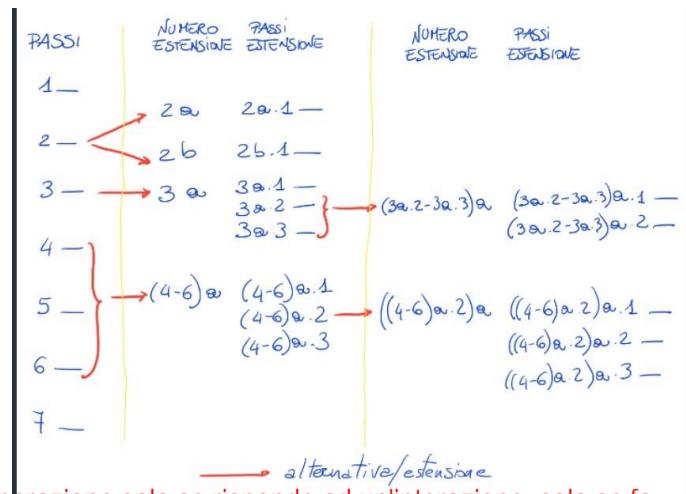
#	Attore	Sistema
(3-5)b.1	Elimina definitivamente il menù su cui sta lavorando	Notifica l'avvenuta eliminazione
	Va al passo 6 dello scenario principale	

Le estensioni possono essere usate per gestire almeno tre tipi di situazioni:

- l'attore vuole che l'esecuzione del UC proceda in modo diverso da quanto previsto nello scenario principale;
- il UC deve procedere diversamente da quanto previsto nello scenario principale, ed è il sistema che se ne accorge, 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.

raro e non consigliabile

Notazione da noi usata



Partendo dall'UC io doto il sistema di un'operazione solo se risponde ad un'interazione, solo se fa raggiungere un obiettivo. Io concordo con il cliente l'UC ma le operazioni le decido io. Non doto il sistema che non verranno mai usate. mi concentro sull'obiettivo dell'utente.

Formattazioni dei casi d'uso → 2 modelli

Sezione del caso d'uso	Commento
Nome del caso d'uso	Inizia con un verbo.
Portata	Il sistema che si sta progettando.
Livello	“Obiettivo utente” o “sottofunzione”.
Attore primario	Usa direttamente il sistema; gli chiede di fornirgli i suoi servizi per raggiungere un obiettivo.
Parti interessate e Interessi	A chi interessa questo caso d'uso e che cosa desidera.
Pre-condizioni	Che cosa deve essere vero all'inizio del caso d'uso – e vale la pena di dire al lettore.
Garanzia di successo	Che cosa deve essere vero se il caso d'uso viene completato con successo – e vale la pena di dire al lettore.
Scenario principale di successo	Uno scenario comune di attraversamento del caso d'uso, di successo e incondizionato.
Estensioni	Scenari alternativi, di successo e di fallimento.
Requisiti speciali	Requisiti non funzionali correlati.
Elenco delle varianti tecnologiche e dei dati	Varianti nei metodi di I/O e nel formato dei dati.
Frequenza di ripetizione	Frequenza prevista di esecuzione del caso d'uso.
Varie	Altri aspetti, come per esempio i problemi aperti.

Caso d'uso UC1: Elabora Vendita

Preambolo:

... come sopra ...

Scenario principale di successo:

Azione (o intenzione) dell'Attore

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.

Responsabilità del Sistema

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.
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.
8. Il Sistema gestisce il pagamento.
9. Il Sistema registra la vendita completa 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).
10. Il Sistema genera la ricevuta.

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

11. Il Cliente va via con la ricevuta e gli articoli acquistati.

Stile essenziale e conciso:

scrivere i UC in uno **stile essenziale**; ignorare l'interfaccia utente, **concentrarsi sull'obiettivo utente**.

La narrativa di un UC viene espressa a **livello delle intenzioni dell'utente e delle responsabilità del sistema**, anziché con riferimento ad azioni con 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.

Practical reasoning:

è il ragionamento sulle azioni, sul processo di capire cosa fare e si distingue da **theoretical reasoning** che riguarda le credenze.

Consiste in due azioni:

- 1) **deliberazione (deliberation)**: decidere quale stato di cose vogliamo raggiungere, il suo output sono le intenzioni;
- 2) **pianificazione (means-ends reasoning-planning)**: decidere come raggiungere questi stati di cose.

Queste azioni sono processi computazionali; come tali, in tutti gli agenti reali questi processi avranno luogo in presenza di risorse limitate. Ciò ha due importanti implicazioni:

- **il calcolo è una risorsa preziosa per gli agenti**: un agente deve controllare il suo ragionamento;
- **gli agenti non possono deliberare a tempo indeterminato**: ad un certo punto devono smettere di deliberare e, dopo aver scelto uno stato di cose, impegnarsi nel raggiungerlo.

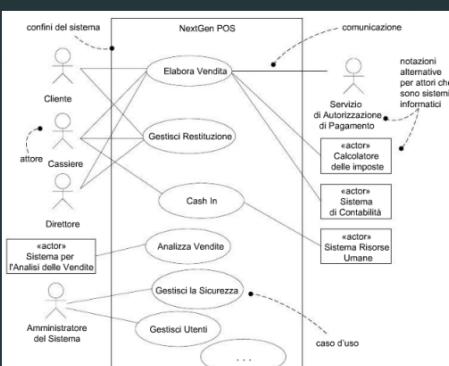
A scatola nera: il sistema è descritto come dotato di responsabilità. Si descrive **che cosa deve fare** (comportamento o requisiti funzionali) **senza** decidere **come** lo farà (progettazione).

Adottare il punto di vista dell'attore: secondo **Ivar Jacobson**, 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.

Come trovare un caso d'uso:

- 1) **scegliere i confini del sistema**;
- 2) **identificare gli attori primari** (coloro che raggiungono i propri obiettivi attraverso l'utilizzo dei servizi di sistema) e **i loro obiettivi**: sono sempre esterni al sistema e aiutano a definire i confini dello stesso;
- 3) **definire i casi d'uso che soddisfano gli obiettivi degli utenti** (il loro nome va scelto in base all'obiettivo). **Diagrammi dei casi d'uso (UCD)**

Disegnare un semplice diagramma dei casi d'uso insieme a un elenco attori-obiettivi.



Verificare l'utilità dei casi d'uso

Si può fare con 3 tipi di test:

non è un caso d'uso perché il capo non sarebbe felice

- 1) **test del capo**: "Cosa avete fatto tutto il giorno?" "Il login!" - Chiedersi: "Il capo sarà felice?";
- 2) **test EBP**: un EBP è un'attività svolta da una persona in un determinato tempo e luogo, in risposta a un evento di business, che aggiunge un valore di business misurabile e lascia i dati in uno stato consistente (es. Approva un credito);
- 3) **test della dimensione**: un caso d'uso è raramente costituito da una singola azione o passo; normalmente comprende diversi passi, e nel formato dettagliato richiede da 3 a 10 pagine di testo. da 3 a 10 pagine

Livello dei casi d'uso

- 1) **livello di obiettivo utente**: nell'analisi dei requisiti per un sistema software è utile concentrarsi soprattutto sui casi d'uso EBP: un caso d'uso di questo tipo è a livello di obiettivo utente, poiché consente all'utente di raggiungere un proprio obiettivo di valore mediante un singolo utilizzo del sistema;
- 2) **livello di sotto-funzione**: rappresenta una funzionalità nell'uso del sistema. Utili per mettere a fattor comune delle sequenze di passi condivise da più casi d'uso, per evitare la duplicazione del testo in comune.

RIASSUNTO IDEAZIONE

	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			

ELABORAZIONE



E' 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.

I requisiti e le iterazioni sono organizzate in base a:

- **rischio**: tecnico, incertezza dello sforzo, usabilità;
- **copertura**: le iterazioni iniziali devono coprire tutte le parti principali del sistema, implementazione in ampiezza e poco profonda;
- **criticità**: le funzioni che il cliente considera di elevato valore di business.

Nell'iterazione 1:

- i requisiti implementati nello sviluppo iterativo sono detti sottoinsiemi dei requisiti o dei casi d'uso completi;
- si inizia la programmazione di qualità-produzione e il test per un sottoinsieme dei requisiti, e si inizia lo sviluppo prima che l'analisi di tutti i requisiti sia stata completata, al contrario di quanto avviene in un processo a cascata.

Sviluppo incrementale dell'iterazione 1:

- un caso d'uso o una caratteristica sono spesso troppo complessi per poter essere completati in una sola breve iterazione;
- pertanto le varie parti o scenari possono essere distribuiti su diverse iterazioni;
- è comune lavorare su diversi scenari di uno stesso caso d'uso per diverse iterazioni, estendendo il sistema in modo graduale per gestire, alla fine, tutte le funzionalità richieste.

Elaborazione nb: durata di massimo un mese, va rilasciato codice, importante raccogliere feedback , vengono fatti test realistici fin da subito, si modella solo ciò che c'è scritto

Artefatti dell'elaborazione

Elaborato	Commento
Modello di Dominio	è importante, è il core, non ha a che fare con il software 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 e comprende diagrammi delle classi software, diagrammi di integrazione 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 ed 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 ad oggetti e la base di dati
StoryBoard dei casi d'uso, Prototipi UI	una descrizione dell'interfaccia utente, delle navigazione, dei modelli di usabilità e così via

MODELLO DI DOMINIO

classi concettuali (non software, no codice)

si passa dal testo al modello di dominio, che è comunque un modo per raffinare il glossario

Il modello di dominio può evolversi in modo da mostrare i concetti significativi relativi ai casi d'uso e può influenzare i contratti delle operazioni, il glossario e il Modello di Progetto.

Caratteristiche:

- rappresentazione visuale delle classi concettuali;
- sviluppato nell'ambito della disciplina di Modellazione del Business;
- insieme di diagrammi di classi UML, includono:
 - oggetti di dominio-classi concettuali;
 - associazioni tra classi concettuali;
 - attributi di classi concettuali;
- è un dizionario visuale;
- non è un modello dei dati;
- è un modo particolare di utilizzare un diagramma delle classi di UML;
- comprende i concetti chiave e la terminologia relativa al dominio del discorso del problema, più specificatamente:

- 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;
- come fonte di ispirazione per la progettazione dello strato del dominio.

Classe concettuale: rappresenta un concetto del mondo reale o nel dominio di interesse di un sistema che si sta modellando (modellazione ontologica VS modellazione concettuale o di dominio).

rappresenta un insieme di proprietà che caratterizzano un oggetto

le classi non contengono i metodi

riduzione del gap di rappresentazione col modello di Progetto, gap così corto che il modello di progetto a volte non lo si gurada neanche

Associazione:

è una relazione tra 2 o + classi che indica una connessione significativa e interessante tra le istanze di queste classi. E' possibile che due classi siano collegate da più di una associazione.

Un'associazione è caratterizzata con: registratore cassa - memorizza - vendita

- nome significativo seguendo la traccia “NomeClasse-FraseVerbale-NomeClasse”;
- molteplicità e direzione di lettura.

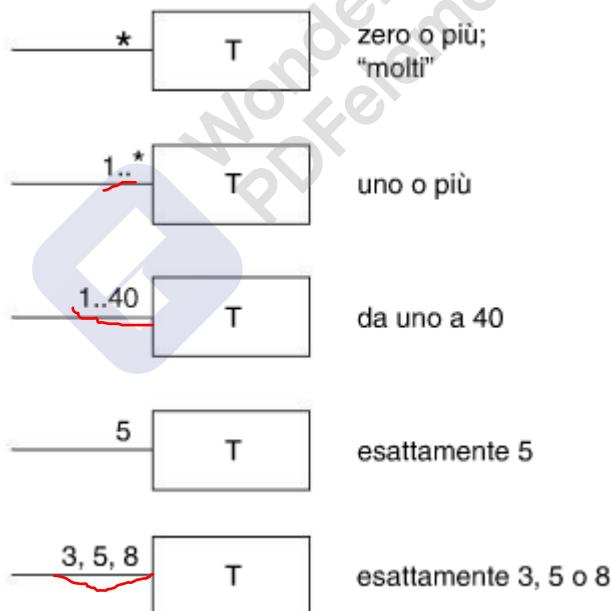
E' per natura bidirezionale, è bidirezionale, NO frecce, al massimo triangolino per indicare verso di lettura per comprensione

Ciascuna estremità di una associazione è anche chiamata ruolo. Un ruolo può avere (opzionalmente):

- **espressione di molteplicità:** definisce quante istanze di una classe possono essere associate ad una istanza di un'altra. I casi possibili di molteplicità sono: uno-a-molti, molti-a-uno, molti-a-molti, uno-a-uno.
Comunemente: A...B significa che A è la molteplicità minima (o 0 o 1) e B è la molteplicità massima (o 1 o *);
- nome;
- navigabilità.

Lettura di una associazione: da destra a sinistra e da sopra a sotto.

Tipi di molteplicità:



Associazione riflessiva: una classe può avere un'associazione con se stessa. I nomi di ruolo sono in questo caso utili per indicare la molteplicità 2 persone creano figli

uno store ha più item



un item può essere contenuto in più store nel tempo ma nel momento solo in uno (viene spostato)



rombo pieno

Composizione (o aggregazione composta): è forte quando ciascuna parte appartiene ad un composto necessariamente, ruota-macchina non è forte perché una è un tipo forte di aggregazione intero-parte: ruota sta anche da sola, ruota di scorta.

- 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).

Classe Descrizione: contiene informazioni che descrivono qualcos'altro. E' utile avere un'associazione che collega la classe e la sua classe descrizione (pattern Item-Descriptor). associazione "describes" è per la classe che descrive ad esempio un articolo ecc, meno frequente

Attributo:

è un valore logico (ovvero un dato, una proprietà elementare) degli oggetti di una classe.

Nel modello di dominio meglio aggiungere attributi che:

- sono tipi di dati primitivi;
- sono tipi enumerativi.

La loro caratterizzazione avviene con:

- origine: derivato (esempio l'età per la data di nascita)/non derivato;
- tipo di dato.

Gli attributi sono mostrati nella seconda sezione del rettangolo di una classe. Opzionalmente è possibile mostrare il loro tipo e altre informazioni.

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

notazione derivato: \total
oppure \quantity

Creare un modello di dominio:

si basa su 4 punti:

- 1) trovare le classi concettuali:

3 modi per farlo

- a) riuso-modifica di modelli esistenti;
- b) utilizzo di elenchi di categorie;
- c) analisi linguistica:
 - i) il mapping nome-classe non è automatico;
 - ii) il linguaggio naturale è ambiguo;
 - iii) fonti: casi d'uso descritti in formato dettagliato.

- 2) disegnarle come classi in un diagramma delle classi UML;
- 3) aggiungere le associazioni;
- 4) aggiungere gli attributi.

si evidenzia e si cercano frase per frase gli elementi
il cliente arriva alla classe con articoli (e mi segno tutti i concetti poi associazioni) importante essere in coppia, confronto è arricchente

Ultime verifiche del modello:

- verificare le classi concettuali introdotte:
 - alternativa classe classe-attributo attributo;
 - classe descrizione;
- verificare le associazioni:
 - indipendenza delle associazioni diverse che sono relative alle stesse classi;
- verificare attributi:
 - non introdurre attributi per riferirsi ad altre classi concettuali (chiavi esterne). Usare le associazioni in questo caso.

ad esempio, pagamento è una generalizzazione perché non esiste nella realtà, nellà realtà c'è il pagamento con carta, quello con contanti ecc. IO LI ASTRAGGO e creo questa classe astratta "pagamento" le cui specializzazioni verrano poi implementate nel software.

E' una generalizzazione solo se posso sostituire tra loro le specializzazioni

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 un elemento delle sue sottoclassi; il suo nome è scritto in corsivo o di fianco c'è scritto "abstract".

N.B.

la generalizzazione non è la specializzazione, quest'ultima è utilizzata in programmazione.

Classi di associazioni: permettono di aggiungere attributi e altre caratteristiche alle associazioni; la loro vita dipende dall'associazione.

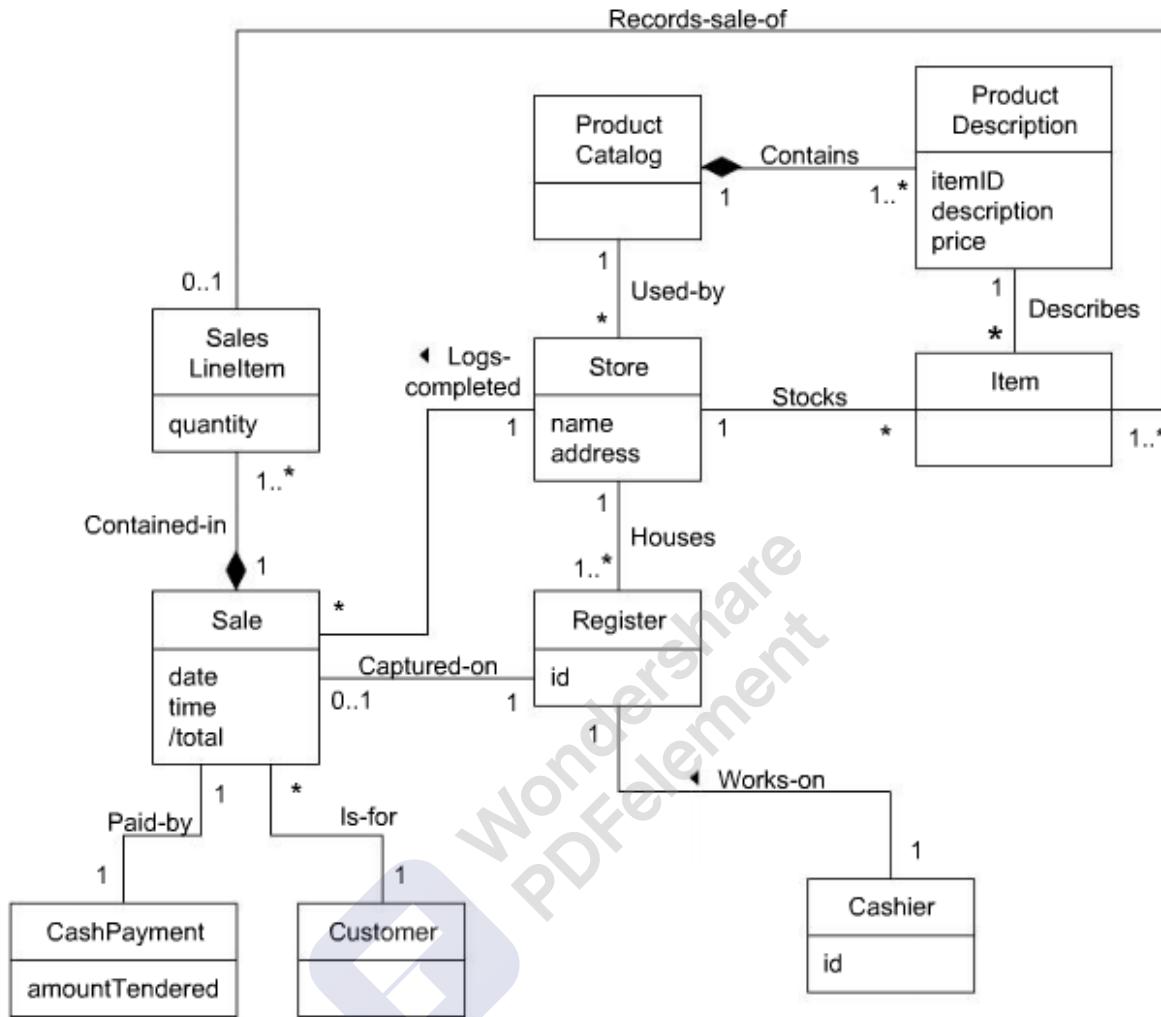
rappresentata con la freccia

i modelli agili attribuiscono più lavoro alle stesse persone, la stessa coppia, dall'analisi alla programmazione

ESERCITAZIONE SUI MODELLI DI DOMINIO

ESERCIZIO 1

Si consideri tale modello di dominio e si risponda alle domande:



- 1) Si consideri l'associazione "contained-in" che collega le due classi "SalesLineItem" e "Sale", dire se sono vere o false queste affermazioni:

 - a) una istanza di "Sale" deve essere distrutta prima di un'istanza di "SalesLineItem" → **falsa**;
 - b) una istanza "SalesLineItem" deve essere distrutta prima di un'istanza di "Sale" → **vera**;
 - c) le istanze di "SalesLineItem" appartengono ad una sola istanza di "Sale" alla volta → **vera**;
 - d) le istanze di "Sale" appartengono ad una sola istanza di "SalesLineItem" alla volta → **falsa**;
 - e) una istanza di "SalesLineItem" può essere creata dopo un'istanza di "Sale" → **vera**;
 - f) una istanza di "Sale" può essere creata dopo un'istanza di "SalesLineItem" → **falsa**;

- 2) Si consideri l'associazione "contains" che collega le due classi "ProductCatalog" e "ProductDescription". Si supponga che "ProductCatalog" = {c1, c2} e "ProductDescription" = {d1, d2, d3, d4}. Dire quali delle seguenti affermazioni sono vere e quali false:
- "Contains" può essere $\{(c1, d1), (c1, \underline{d2}), (c2, d2), (c2, d3), (c2, d4)\}$ → **falsa**: d2 non soddisfa i vincoli di molteplicità;
 - "Contains" può essere $\{(c1, d1), (c1, d2), (c1, d3), (c2, d4)\}$ → **vera**;
 - "Contains" può essere $\{(c1, d1), (c1, d2), (c2, d3)\}$ → **falsa**: d4 non soddisfa i vincoli di molteplicità;
 - "Contains" può essere $\{(c1, d1), (c1, d2), (c1, d3), (c1, d4)\}$ → **falsa**: c2 non soddisfa i vincoli di molteplicità.

ESERCIZIO 2

Si consideri il diagramma seguente:

A (1...3) ----- ← R ----- (0...1) B

Si supponga A = {a1, a2, a3, a4} e B = {b1, b2}.

Dire quali affermazioni siano vere e quali false:

- R può essere $\{(b1, a1), (b1, a2), (b2, a1)\}$ → **vera**;
- R può essere $\{(b1, a1), (b1, a2), (b1, a3)\}$ → **falsa**: b2 non soddisfa i vincoli di molteplicità;
- R può essere $\{(b1, a1), (b1, a2), (\underline{b1}, \underline{a3}), (b1, a4), (\underline{b2}, a3)\}$ → **falsa**: b1 e a3 non soddisfano i vincoli di molteplicità;
- R può essere $\{(b1, a1), (b1, a2), (b2, a3), (b2, a4)\}$ → **vera**.

ESERCIZIO 3

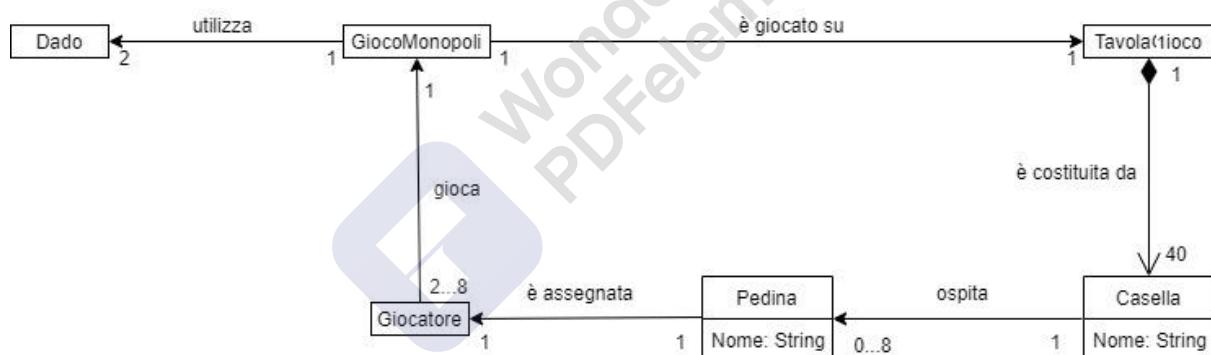
Si consideri la seguente descrizione del gioco Monopoli:

- il **gioco di Monopoli** è **giocato su una tavola di gioco** specifica per il Monopoli, su di una tavola di gioco si può giocare **solo un** gioco di Monopoli alla volta;
- la **tavola da gioco** di Monopoli è **costituita da 40 caselle** e ogni **casella ha il suo nome**;
- un **gioco** di Monopoli **utilizza due dadi**;
- un **dado** è utilizzato da un solo **gioco di Monopoli** alla volta;
- ad un gioco di Monopoli possono **giocare da due a otto giocatori**;
- un giocatore può giocare **un solo gioco** di Monopoli alla volta;
- ad ogni giocatore è **assegnata una pedina** diversa;
- ogni **pedina ha un nome**;
- ogni pedina è su di **una sola casella** alla volta;
- una casella **può ospitare da 0 a 8 pedine**.

Definire un Modello di Dominio per il gioco Monopoli.

Descrizione dei colori:

- **giallo** = concetto;
- **arancione** = proprietà;
- **azzurro** = associazione;
- **viola** = multimedialità.



ESERCIZIO 4

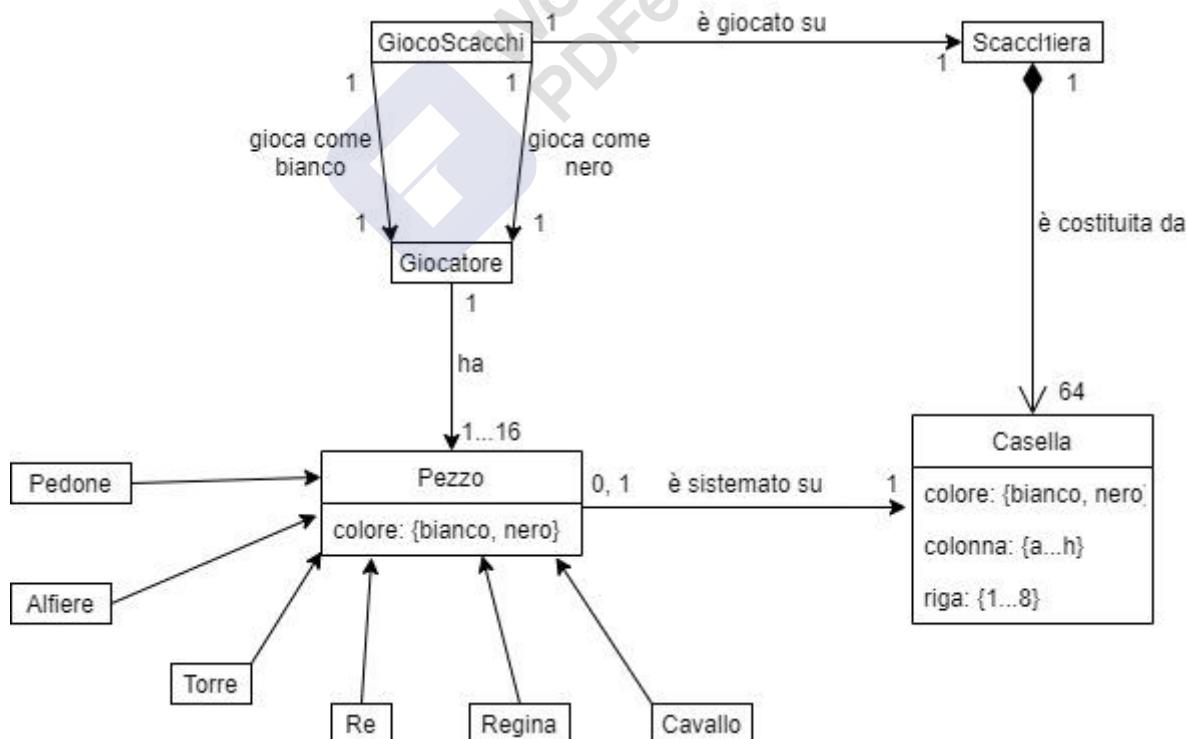
Si consideri la seguente descrizione del gioco degli Scacchi:

- il **gioco degli scacchi** è giocato su **una scacchiera**, su di una scacchiera si può giocare solo **un** gioco degli scacchi alla volta;
- la scacchiera è costituita da **64 caselle** e ogni casella è individuata da una lettera che va **da "a" a "h"** che indica la **colonna** e un numero **da 1 a 8** che indica la **riga**;
- una casella appartiene ad **una sola** scacchiera e ha un **colore** che può essere **bianco o nero**;
- ad un gioco degli scacchi è giocato da un **giocatore con il ruolo del bianco** e da un **giocatore con il ruolo del nero**;
- in una casella può essere **sistemato** **un solo pezzo** tra un **re**, una **regina**, un **alfiere**, una **torre**, un **cavallo** o un **pedone**;
- ogni pezzo deve essere sistemato su** una casella;
- un giocatore **ha** almeno un pezzo degli scacchi e un massimo di 16 pezzi degli scacchi;
- ogni pezzo ha un colore** (bianco o nero).

Definire un Modello di Dominio per il gioco degli Scacchi.

Descrizione dei colori:

- giallo** = concetto;
- arancione** = proprietà;
- azzurro** = associazione;
- viola** = multimedialità.



di sequenza perché vengono illustrate le interazioni tra attore e sistema

DIAGRAMMI DI SEQUENZA DI SISTEMA

Definizione: è un elaborato della disciplina dei requisiti che illustra eventi di input e di output relativi ai sistemi in discussione.

N.B.

Un **ssd**:

- non è menzionato esplicitamente in UP;
- è espresso attraverso i diagrammi di sequenza di UML;
- è modellato come una "scatola nera"; non scendiamo nelle implementazioni
- si modella un **ssd** per ogni UC per lo scenario principale e per ogni scenario alternativo;
- costituisce un **input** per i contratti delle operazioni per la progettazione ad oggetti.

Eventi: 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. l'interazione genera un evento che richiede un'operazione di sistema per gestirla (che è ciò che andremo poi ad implementare)

N.B.

- le operazioni di sistema sono operazioni che il sistema deve definire proprio per gestire gli 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 SSD sono utili per illustrare interazioni tra attori e le operazioni iniziate da essi; infatti, essi sono delle figure che mostrano, per un particolare scenario di un UC, gli eventi generati dagli attori esterni al sistema, il loro ordine e gli eventi inter-sistema.

Un **sistema** reagisce a tre cose:

- 1) eventi esterni da parte di attori umani o sistemi informatici;
- 2) eventi temporali;
- 3) 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 UC e può essere generato per ispezione da tale scenario.

Un **SSD** mostra:

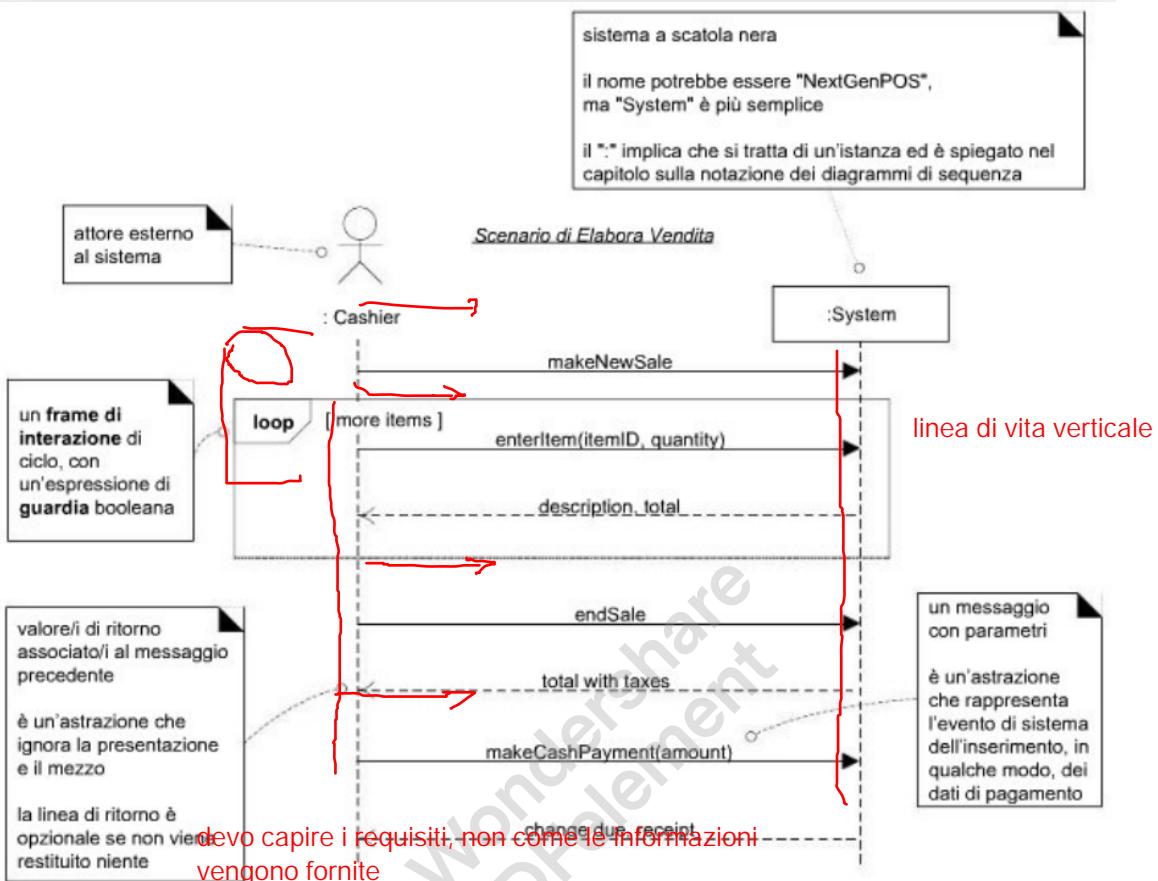
- l'attore primario del caso d'uso;
- il sistema in discussione;
- i passi che rappresentano le interazioni tra il sistema e l'attore.

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

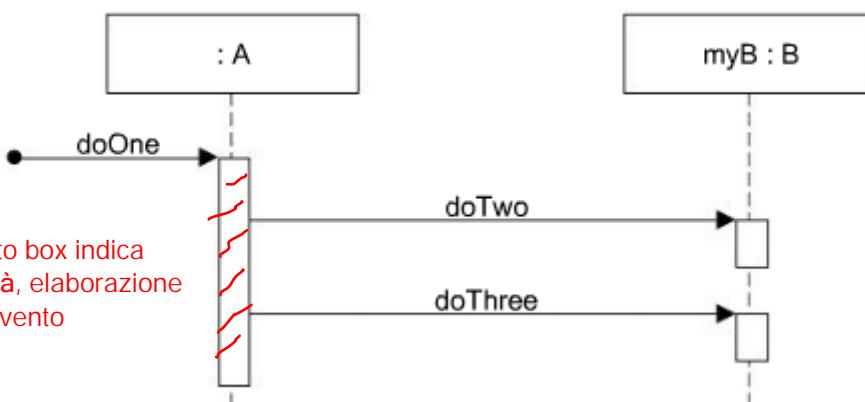
ESEMPIO DI SSD

scenario di base
con pagamento in
contanti

GUARDARE
SLIDE 11 SSD

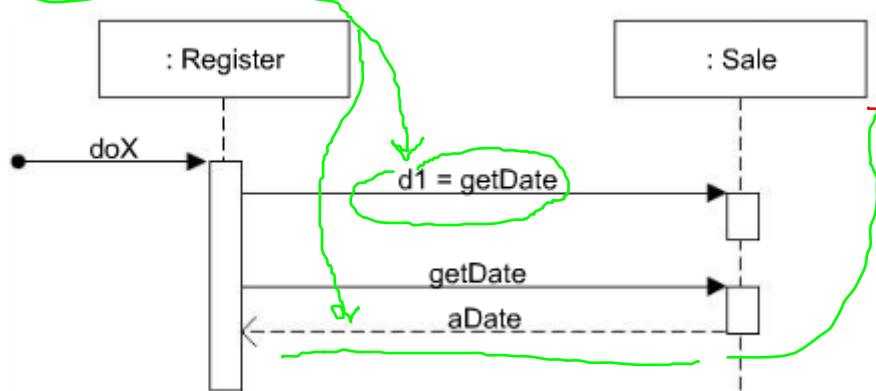


NOTAZIONE



doTwo e doThree sono messaggi sincroni e i rettangoli sono le elaborazioni delle richieste.

Due modi per mostrare un risultato di ritorno da un messaggio



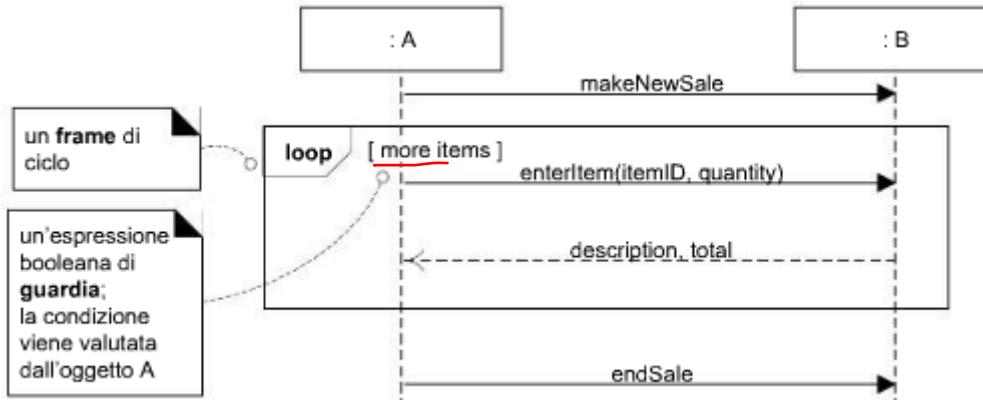
viene inviato il messaggio, parte l'elaborazione, linea tratteggiata è la risposta (viene fornita una data)

è più carina la freccia di ritorno

Operatori comuni per i frame di UML

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

per esempi guardare slide, si possono mettere frame annidati es un for dentro un if



Le operazioni di sistema vengono trovate mentre si abbozzano gli SSD i quali mostrano gli eventi di sistema che implica un'operazione per essere gestito

CONTRATTI

Definizione: è uno strumento ottimo nell'analisi dei requisiti o nell'analisi orientata agli oggetti per descrivere in modo dettagliato i cambiamenti richiesti dall'esecuzione di una operazione di sistema (in termini di oggetti del modello di dominio), senza però descrivere come devono essere ottenuti questi cambiamenti ci sarà un messaggio e un'operazione di sistema che dovrà gestirlo

I contratti delle operazioni di sistema usano pre-condizioni e post-condizioni per descrivere nel dettaglio i cambiamenti agli oggetti in un modello di dominio causati dall'esecuzione dell'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).

N.B.

- possono essere considerati parte del modello dei casi d'uso, poiché forniscono maggiori dettagli dell'analisi sull'effetto delle operazioni di sistema implicate dai casi d'uso;
- non sono menzionati esplicitamente in UP.

TEMPLATE

i contratti sono la fine dei requisiti e l'input della progettazione a oggetti
-> modelli di progetto (classi software)

- **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-condizioni:** è la parte più importante. Descrive i cambiamenti di stato degli oggetti (concettuali) nel modello di dominio dopo il completamento dell'operazione. I cambiamenti dello stato del modello di dominio comprendono gli oggetti creati, i collegamenti formati o rotti, e gli attributi modificati.

NON sono azioni da eseguire ma OSSERVAZIONI, fotografia di come il modello di dominio deve essere al termine dell'esecuzione

N.B.

nelle pre-condizioni è utile indicare gli oggetti rilevanti a quel punto del caso d'uso particolare quelli che si vogliono citare nelle post-condizioni.

Per scrivere un contratto si procede come segue:

che cosa e non come,
il come nell'
implementazione

- 1) identificare le operazioni di sistema degli ssd;
- 2) 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;
- 3) per descrivere le post-condizioni si utilizzano le seguenti sotto-categorie:
 - a) creazione o cancellazione di oggetto (o istanza);
 - b) formazione o rottura di collegamento;
 - c) modifica di attributo.

esempio slide!!
da slide 16

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.

Alcune domande utili:

- ❖ 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 è iterativo ed incrementale

ARCHITETTURA LOGICA E ORGANIZZAZIONE IN LAYER

Dove siamo? Nella progettazione!

lavoro a strati è un tipico modo, ogni strato aggiunge funzionalità

Nei requisiti il fuoco è “fare la cosa giusta”, ovvero capire alcuni degli obiettivi preminenti per i casi di studio e le relative regole e vincoli. Nella progettazione si pone l’accesso sul “fare la cosa bene”, ovvero sul progettare abilmente una soluzione che soddisfa i requisiti per l’iterazione corrente.

N.B. essendo analisti e programmati le stesse persone

E’ naturale scoprire e modificare alcuni requisiti durante il lavoro di progettazione e di implementazione, soprattutto nelle iterazioni iniziali.

La programmazione dall’inizio, i test e le demo aiutano a provocare presto questi cambiamenti inevitabili.

E’ lo scopo dello sviluppo iterativo.

Architettura: la progettazione di un tipico sistema orientato agli oggetti è basata su diversi strati architettonici, come uno stato dell’interfaccia utente, uno stato della logica applicativa (o “del dominio”) e così via.

Architettura logica di un sistema software: è la macro-organizzazione su larga scala delle classi software in package (o namespace), sottoinsiemi e stati. E’ logica in quanto non vengono prese decisioni su come questi elementi siano distribuiti sui processi o sui diversi computer fisici di una rete (platform independent architecture).

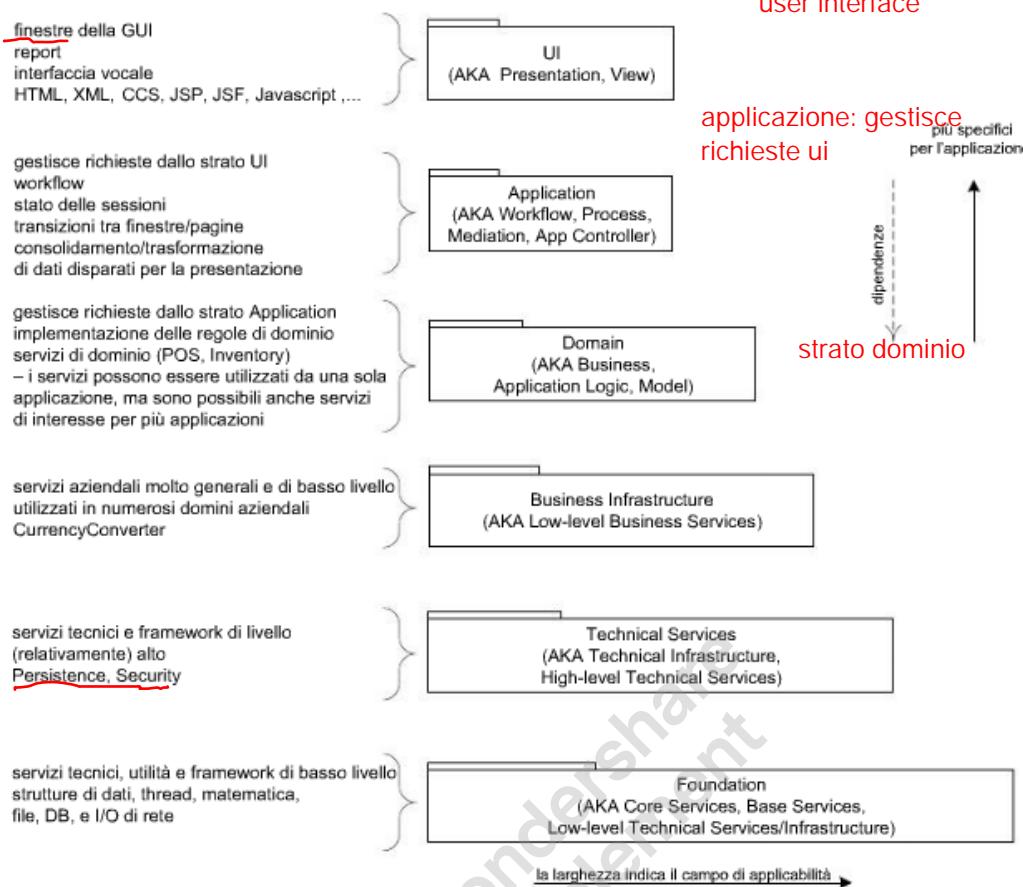
Layer o strato: gruppo di classi, packages, sottoinsiemi con responsabilità condivisa su un aspetto importante del sistema.

Gli strati di un’applicazione software comprendono normalmente: slide 7 esempio

- **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 sottoinsiemi 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.

Tipica architettura a strati



si va da quelli meno specifici (dal basso) a quelli più specifici (all'alto)

2 principi:

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

Alta coesione: in uno strato le responsabilità degli oggetti devono essere fortemente correlate, l'uno all'altro, e non devono essere mischiare con le responsabilità degli altri strati.

importando un package non importare roba che non mi serve (coesione, ciò che c'è dentro è correlato)

Come va progettata la logica applicativa con gli oggetti?

L'approccio consigliato consiste nel creare degli oggetti software con nome 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.

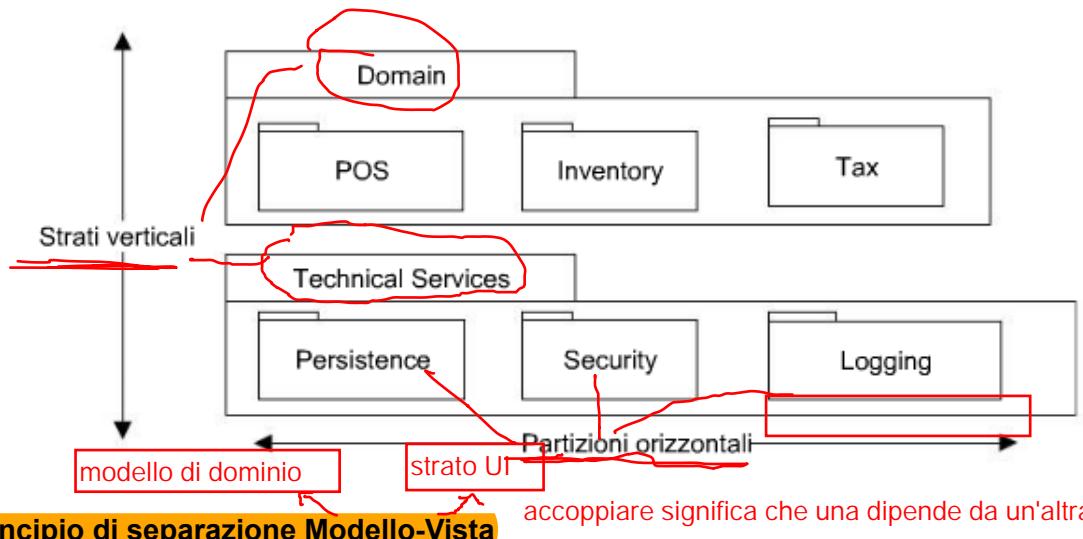
Lo stato di dominio fa riferimento al modello di dominio per trarre ispirazione per i nomi delle classi dello stato del dominio.

Creando uno stato 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.

all'interno dello stesso package

partendo dal modello di dominio realizzo delle classi software,
prendendo ispirazioni anche per i nomi



Questa separazione deriva dal pattern MVC Model-View-Controller (pattern object oriented finalizzato alla separazione di elementi e al riuso).
Introdotto da smallTalk-80

- non relazionare o accoppiare oggetti non UI con oggetti UI: gli oggetti non UI non devono essere connessi o accoppiati direttamente agli oggetti non UI;
- non incapsulare la logica dell'applicazione in metodi di UI: non mettere logica applicativa 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;
- gli oggetti del modello (dominio) non devono avere una conoscenza diretta degli oggetti della vista (UI), almeno in quanto oggetti della vista;
- modello = strato di dominio, vista = strato UI.

N.B.

- le finestre appartengono ad una applicazione in particolare, mentre gli oggetti non UI possono venire riutilizzati in nuove applicazioni od essere relazionati a nuove interfacce;
- gli oggetti UI inizializzano elementi UI, ricevono eventi UI e le richieste della logica dell'applicazione agli oggetti delegano non UI (oggetti di dominio).

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

Le classi della vista sono relativamente leggere, esse sono responsabili dell'input e dell'output e di catturare gli eventi della GUI ma non mantengono i dati dell'applicazione né forniscono direttamente della logica applicativa.

Vantaggi del principio di separazione Modello-Vista

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

Le operazioni di sistema stanno nello strato di dominio, l'interfaccia grafica cattura gli eventi dell'utente makeNewSale, endSale ecc, raccoglie le informazioni che costituiscono l'evento e poi lo delegano al dominio che implementa l'operazione

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 dello strato UI allo strato del dominio saranno i messaggi mostrati negli ssd.

Come progettare gli 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.

strategia più comune, con uso di UML per comunicare e comprendere nel team

Disegno leggero: con “disegno, poi codifica”, il costo aggiuntivo dovuto al disegno dovrebbe ripagare lo sforzo impiegato.

Ci sono due tipi di modelli per gli oggetti:

- **dinamici:** rappresentano il comportamento del sistema, la collaborazione tra gli oggetti software per realizzare (uno/due 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.

più importanti

spesso non si fa

perché è facile

derivarlo dal modello

di dominio

La progettazione ad oggetti richiede soprattutto la conoscenza di:

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

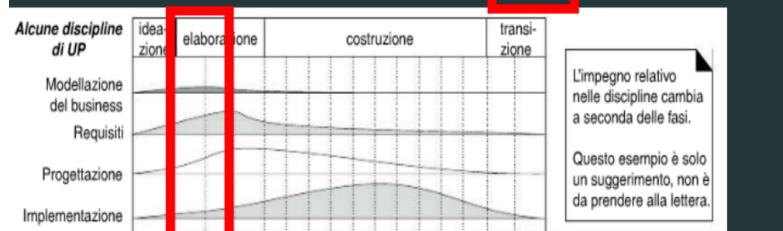
programmazione ad oggetti: quali sono le responsabilità dell'oggetto? Chi collabora con esso?

N.B.

- 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 ad 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 che si applicano la progettazione è guidata dalle responsabilità e i principi GRASP.

Tavola 2.1 Scenario di Sviluppo di esempio (i – inizio; r – raffinamento)						
Disciplina	Pratica	Elaborato Iterazione →	Ideazione I1	Elaboraz. E1..En	Costr. C1..Cn	Transiz. T1..T2
Modellazione del business	modellazione agile workshop requisiti	Modello di Dominio	i			
Requisiti	workshop requisiti esercizio sulla visione votazione a punti	Modello dei Caso d'Uso	i	r		
		Specifiche	i	r		
		Supplementare	i	r		
Progettazione	modellazione agile sviluppo guidato dai test	Modello di Progetto Documento dell'Architettura Software Modello dei Dati	i	r		
			i	r		
Gestione del progetto	dai test programmazione a coppie integrazione continua standard di codifica	gestione del progetto gille riunioni Scrum giornaliere	...			
...						

Entriamo nella Progettazione con il Modello di Dominio, non vedremo invece il modello dei dati per il quale potremmo ispirarci al 'ER



L'impegno relativo nelle discipline cambia a seconda delle fasi.

Questo esempio è solo un suggerimento, non è da prendere alla lettera.

DSD

DCD

MODELLAZIONE DINAMICA E STATICÀ CON UML

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

I diagrammi di interazione sono utilizzati per la modellazione dinamica degli oggetti e sono una generalizzazione dei tipi più specifici di diagrammi UML di sequenza e di comunicazione.
 noi vedremo solo quelli di sequenza, quelli di sequenza si sviluppano in modo orizzontale (e verticalmente ci sono i tempi), quelli di comunicazione in maniera bidimensionale

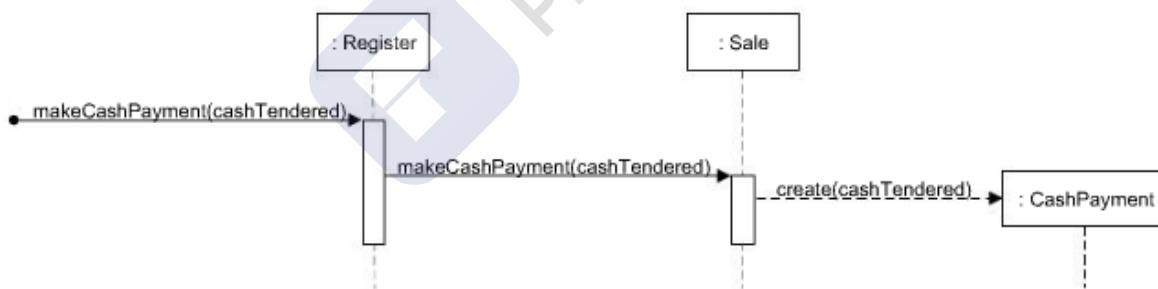
Interazione: è una specifica di come alcuni oggetti si scambiano messaggi nel tempo per eseguire un compito nell'ambito di un certo contesto.

- una 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; un ruolo perché ognuno ha capacità specifiche
- 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.

I **diagrammi di sequenza** mostrano le **iterazioni** 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.



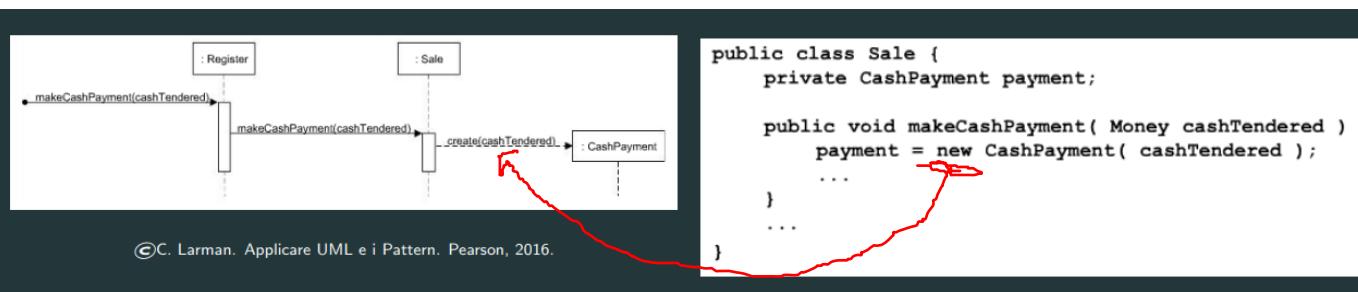
"Register" e "Sale" sono classi già esistenti
 "CashPayment" è una classe che viene creata

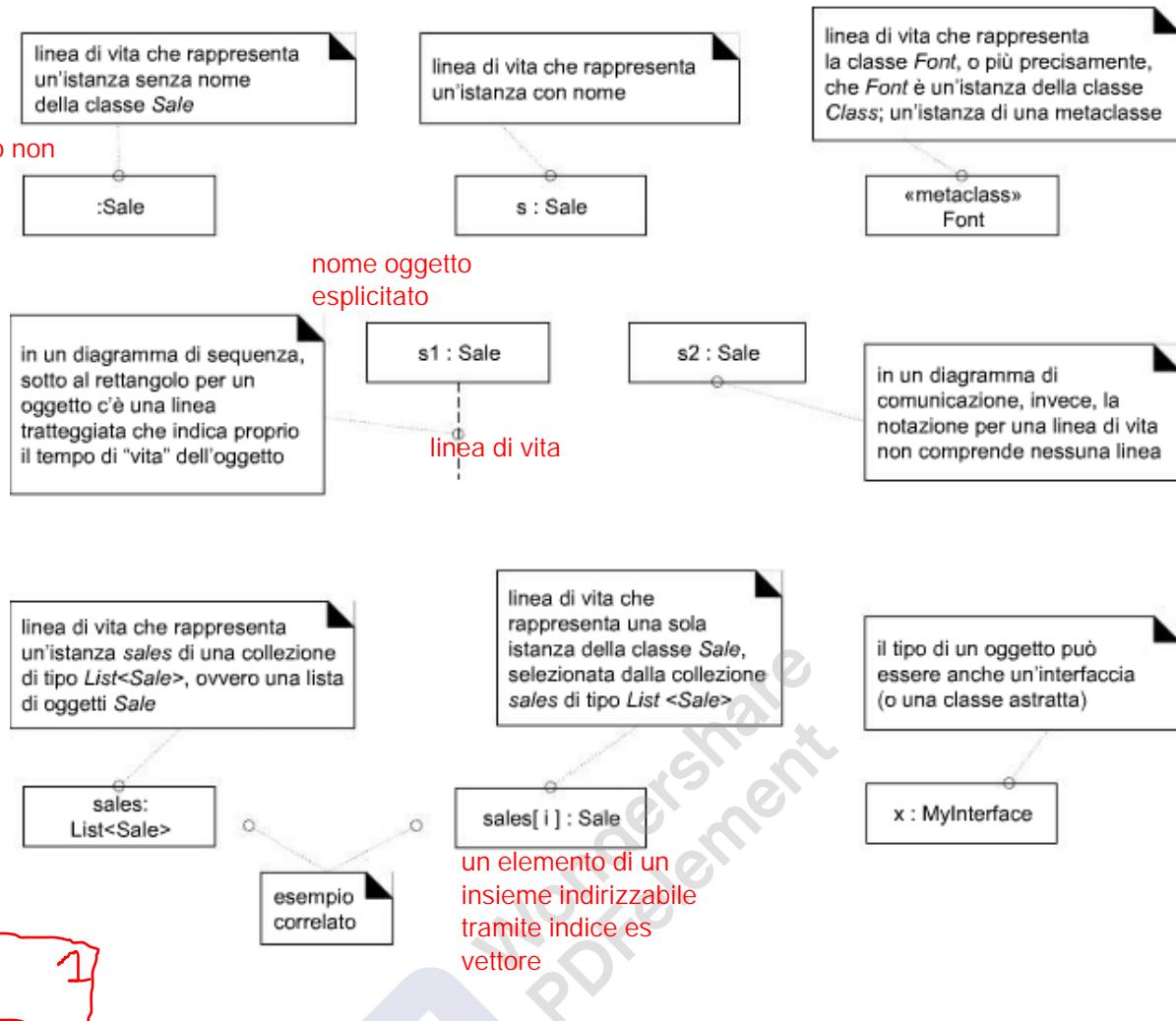
Design Sequence Diagram: è un **diagramma di sequenza** utilizzato da un punto di vista software o di progetto.

In UP, l'insieme di tutti i **DSD** fa parte del **modello di progetto** che comprende anche i diagrammi delle classi.

strumento è lo stesso, obiettivo è diverso, scoprire le responsabilità delle varie parti e come collaborano

Linee di vita (lifeline): rappresentano i **partecipanti all'interazione**, ovvero le parti correlate definite nel contesto di un qualche diagramma strutturale.



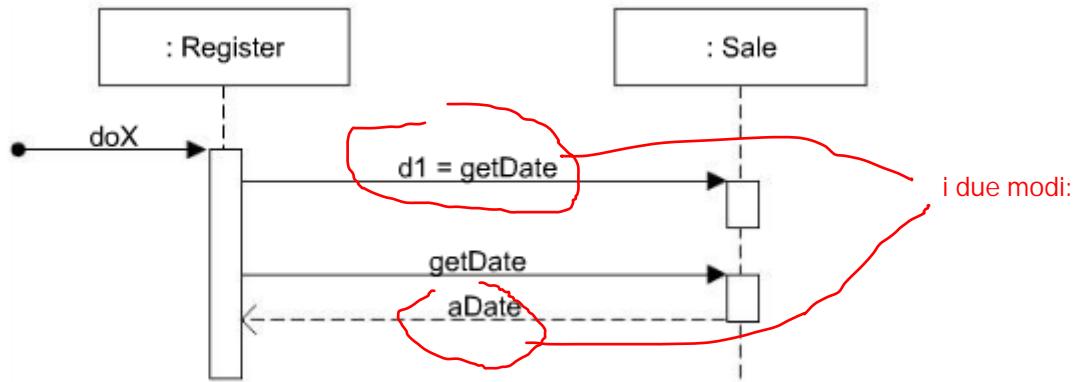


Singleton: pattern nel quale da una classe viene istanziata una sola istanza, mai due o più. In un diagramma di interazione, un tale oggetto "singleton" è contrassegnato da un "1" nell'angolo superiore destro della linea di vita.

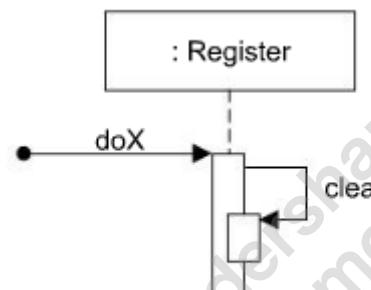
- una linea di vita comprende sia un rettangolo che una linea verticale che si estende sotto di esso;
- **ogni messaggio** (di solito sincrono) tra gli oggetti è rappresentato da un'espressione messaggio mostrata su una linea continua con una freccia piena → tra le linee di vita verticali (la punta sottile, non piena, → indica un messaggio asincrono);
- il messaggio iniziale è chiamato messaggio trovato;
- una barra di specifica di esecuzione (o barra di attivazione o attivazione) mostra l'esecuzione di un'operazione da parte di un oggetto.

Ci sono due modi per mostrare il risultato di ritorno:

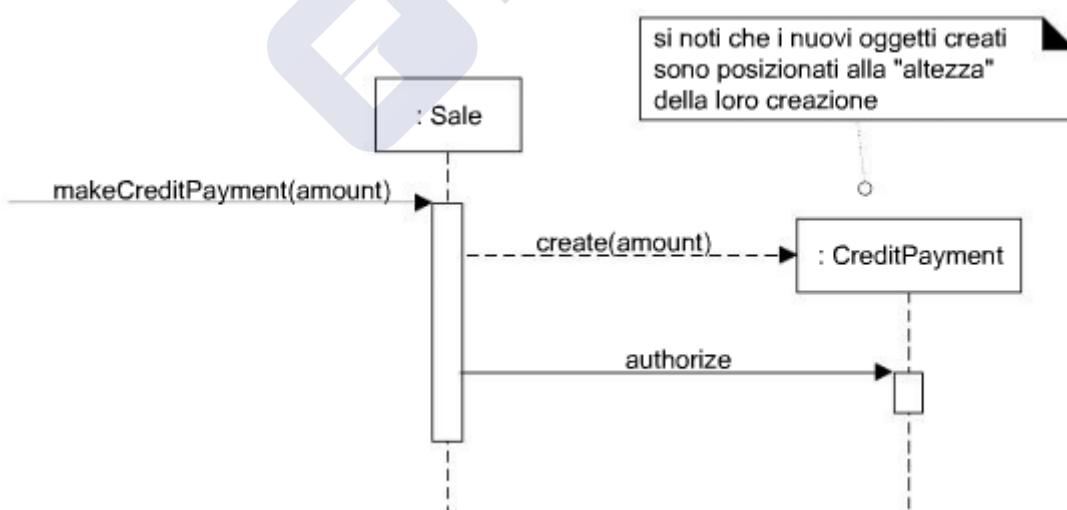
- utilizzando la sintassi `returnVar = message(parametri)`
- utilizzando una linea di messaggi di risposta (o ritorno) alla fine della barra di specifica di esecuzione.



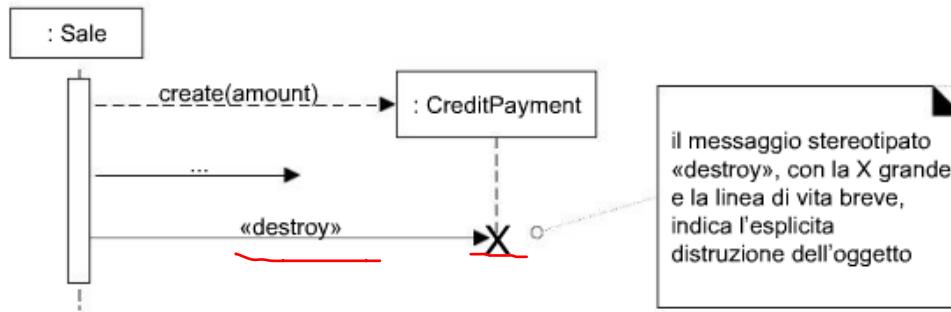
E' possibile mostrare un messaggio inviato da un oggetto a se stesso utilizzando una barra di specifica di esecuzione annidata.



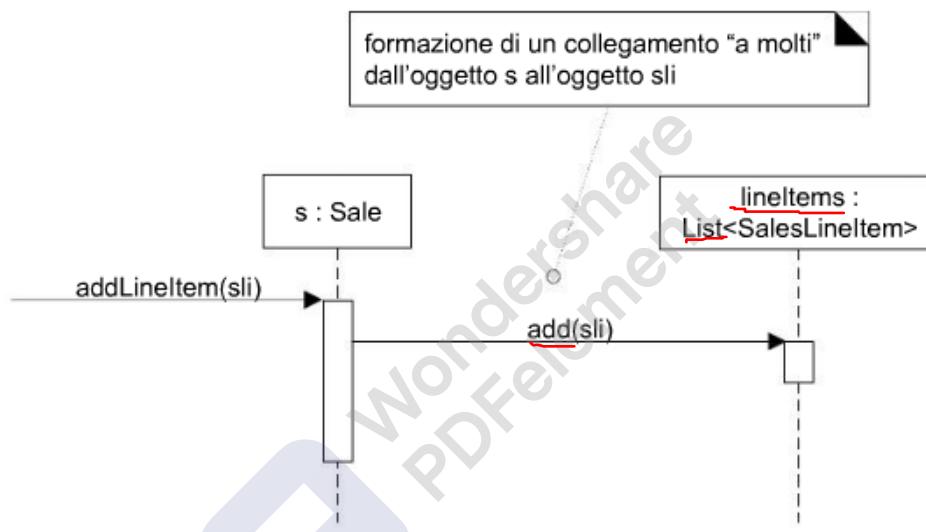
Si noti che i nuovi oggetti creati sono posizionati all'altezza della loro creazione.



Una distruzione esplicita di un oggetto.

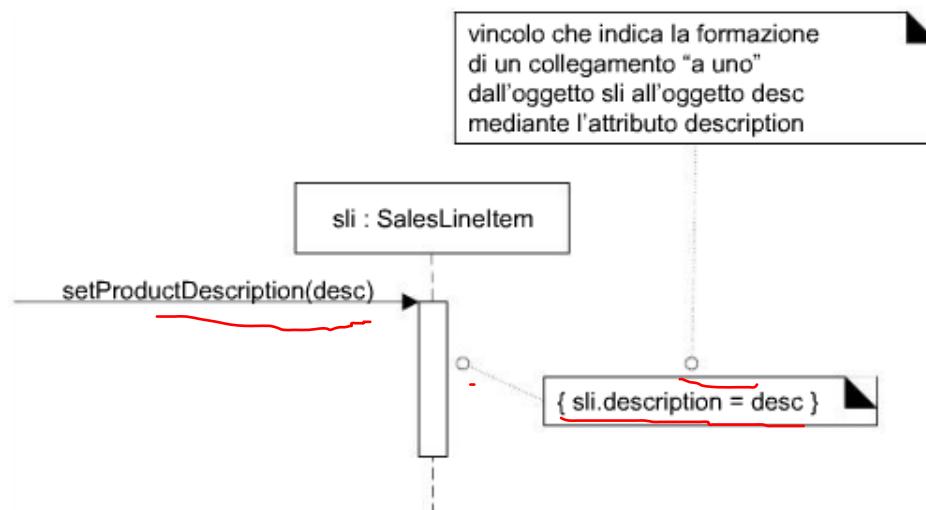


Formazione di un collegamento di un'associazione “a molti”.



Formazione di un collegamento di un'associazione “a uno”.

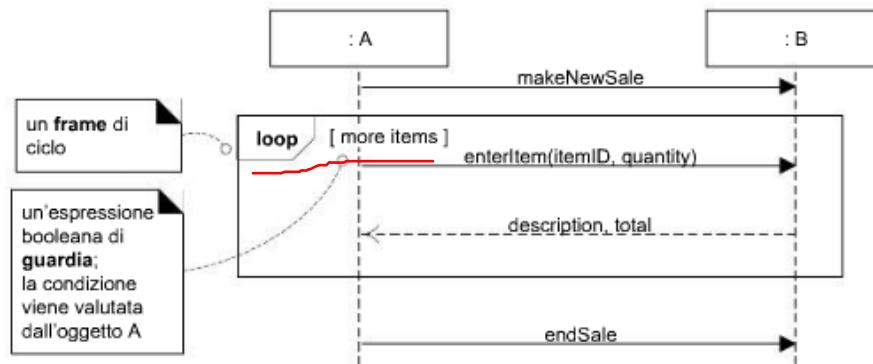
Il vincolo va interpretato come una post-condizione dell'operazione, ovvero una condizione che deve risultare vera al termine della sua esecuzione.



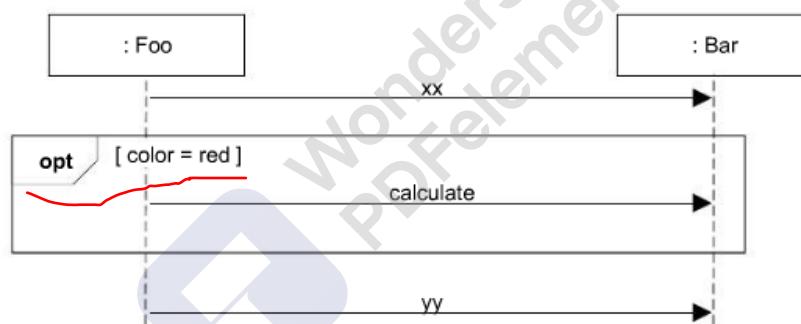
Come supporto alle "istruzioni" di controllo condizionali e di ciclo si utilizza i frame.

I **frame** sono regioni o frammenti dei diagrammi, hanno un operatore (etichetta) e una guardia (condizione o test booleano) che va posizionata sopra la linea di vita che deve valutare tale condizione.

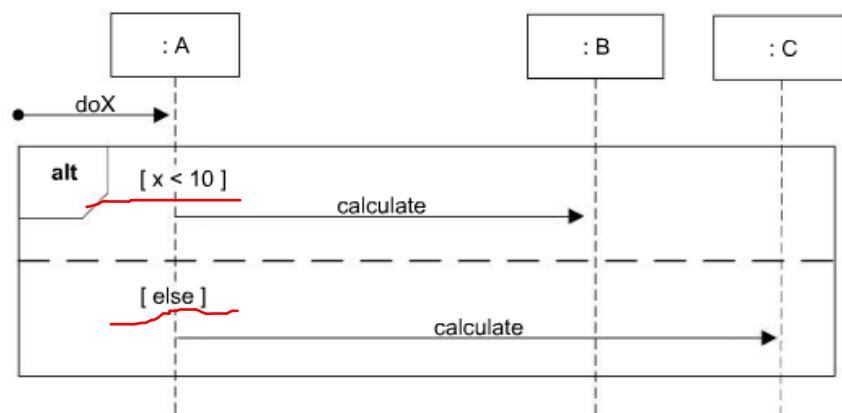
Un frame **LOOP** è posizionato attorno a uno o più messaggi da iterare.



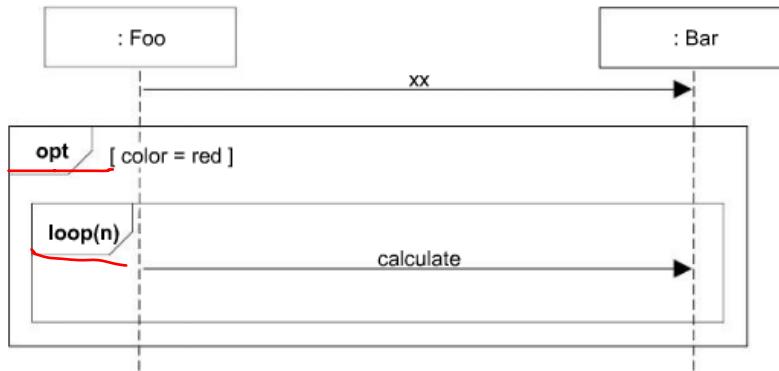
Un frame **OPT** (cioè Optional) è posizionato attorno a uno o più messaggi.



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



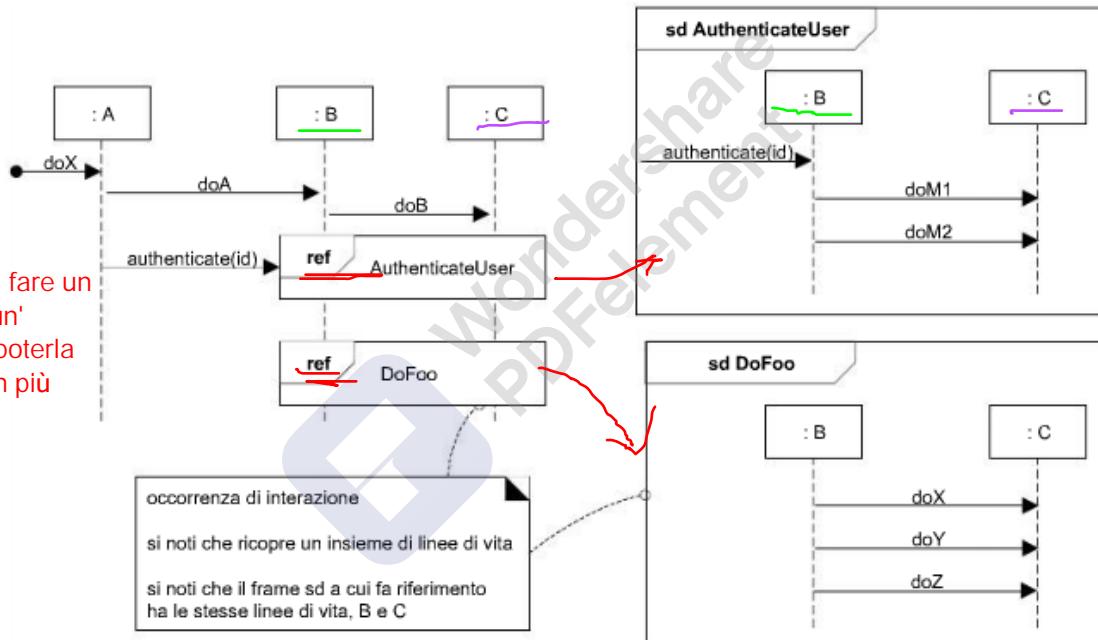
I frame possono essere annidati.



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.

ref

mi permette di fare un riferimento a un'interazione e poterla usare anche in più punti



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

metodo di classe,
statico



nel diagramma delle classi di progetto, a differenza di quello di dominio, si parla di software e quindi le associazioni al posto che essere bidirezionali, hanno una visibilità, rappresentata dalla freccia.

N.B.

la differenza tra le frecce è sottile. Non si dia per scontato che la forma della freccia sia corretta

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

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

ci sono vari decoratori testuali come



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

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

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

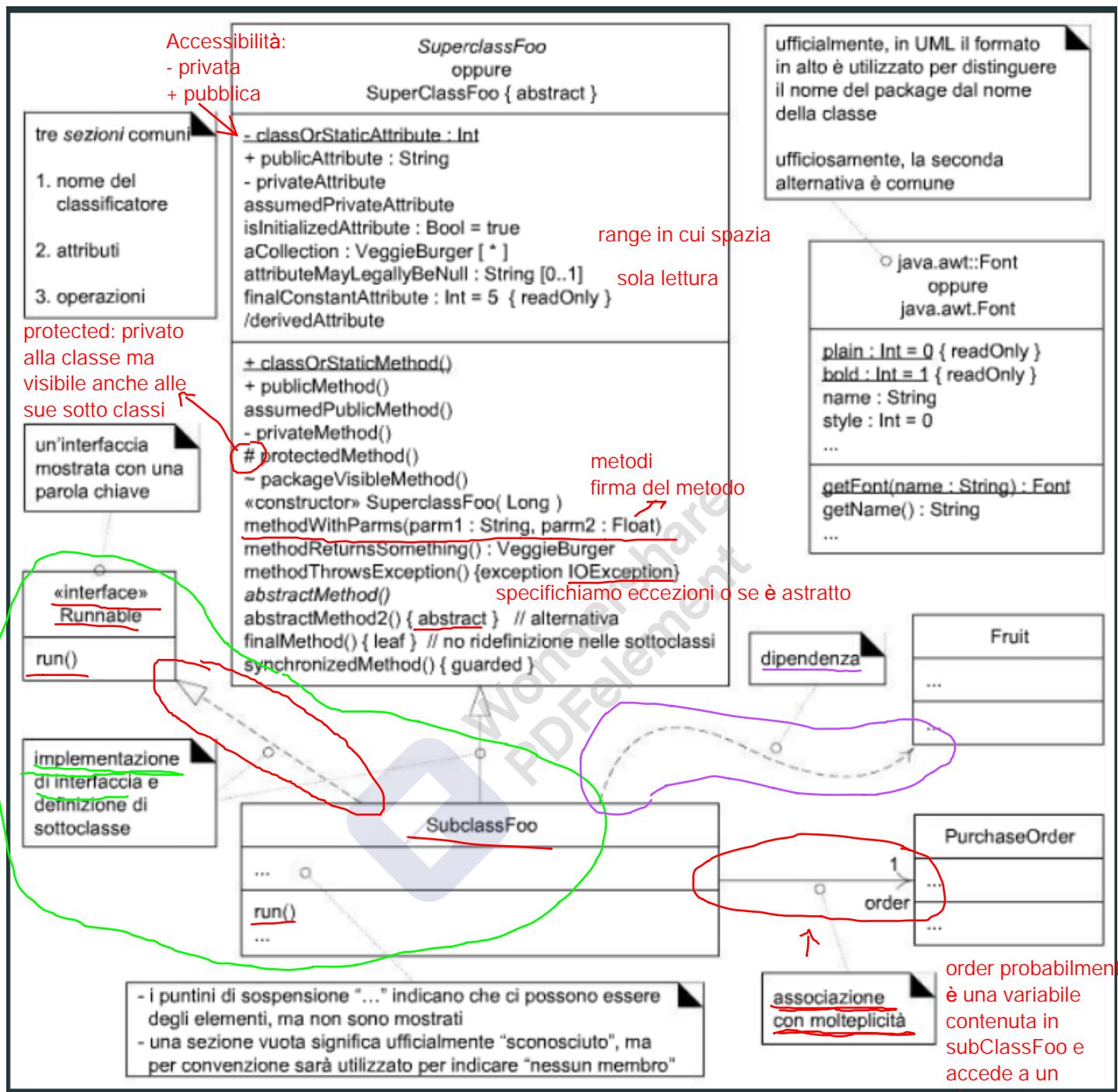
I diagrammi delle classi sono utilizzati per la modellazione statica di oggetti.

Notazioni per DCD:

- notazione testuale per un attributo (in alto);
- notazione con una linea di associazione (in mezzo);
- entrambe le notazioni, 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, che indica che un oggetto della classe sorgenti 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.

NOTAZIONE COMUNE DEI DIAGRAMMI DELLE CLASSI DI UML

pubblico anche al di fuori del package





Operazione: dichiarazione di un metodo e hanno visibilità pubblica per default.

Sintassi:

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

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).

Relazione tassonomica: avviene 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. eredita proprietà e metodi

Generalizzazione: implica l'ereditarietà nei linguaggi OO.

Linee di dipendenza: 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 (= accoppiamento). noi vogliamo ridurre al minimo l'accoppiamento (in progettazione)

Esistono vari tipi di dipendenza:

- avere un attributo del tipo del fornitore;
- inviare un messaggio a un fornitore; la visibilità verso un 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 di fornitore;
- il fornitore è una superclasse o un'interfaccia implementata.

Singleton: classi di cui si può fare una sola istanza

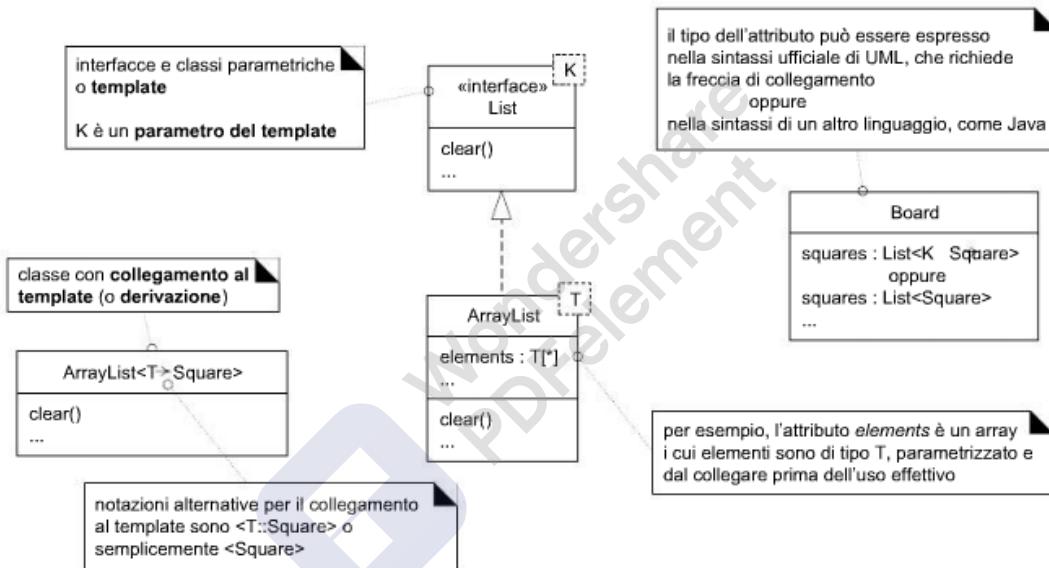
Interfacce:

- L'implementazione di un'interfaccia viene chiamata una realizzazione di interfaccia.
- La notazione a pallina (lollipop) indica che una classe (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.

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

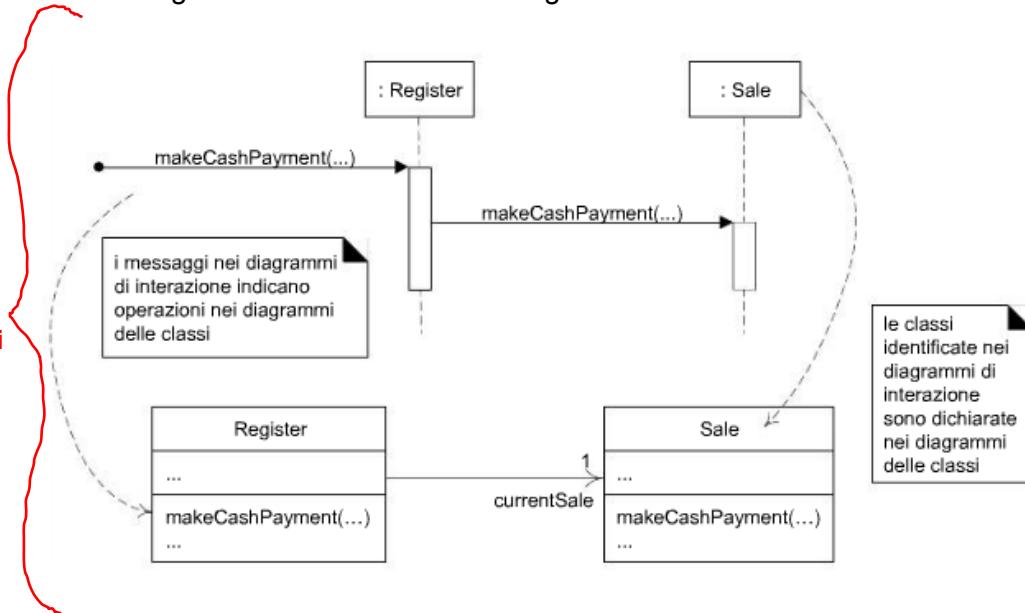
- gli oggetti B non possono esistere indipendentemente da un oggetto A;
- l'oggetto A è responsabile della creazione e distribuzione dei suoi oggetti B.

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



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

quando abbiamo
diagramma di
sequenza +
diagramma delle
classi, siamo pronti
per produrre il
codice



Prima della produzione del codice dobbiamo fare la progettazione, per farlo
utilizziamo i principi di programmazione GRASP

GENERAL RESPONSIBILITY ASSIGNMENT SOFTWARE PATTERNS

Secondo Martin Fowler, capire le responsabilità è fondamentale per una buona
programmazione ad oggetti; per fare ciò si devono aggiungere i metodi alle classi
appropriate e definire i messaggi fra gli oggetti per soddisfare i requisiti.

GRASP sono schemi e principi di assegnamento di responsabilità nel software

Principi e pattern: la padronanza dell'OOD coinvolge un insieme ampio di principi flessibili
(o pattern), con molti gradi di libertà quindi bisogna ragionarci molto sopra, fase delicata

RDD Responsibility - Driven Development: l'approccio comprensivo al fare la modellazione per
la progettazione OO si baserà sulla metafora della progettazione guidata dalle
responsabilità, ovvero pensare a come assegnare a degli oggetti che collaborano.

In questo approccio gli oggetti software sono considerati come dotati di responsabilità; per
responsabilità si intende un'astrazione di ciò che fa o rappresenta un oggetto o un
componente software.

Output della progettazione ad oggetti: diagrammi di interazione e diagrammi delle classi
UML. In particolare, dopo le modellazioni nella prima interazione:

- 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 delle basi di dati.

Responsabilità:

- in UML la responsabilità è un contratto o un obbligo di un classificatore;
- sono correlate agli obblighi o al comportamento di un oggetto in relazione al suo
ruolo;
- le responsabilità sono fondamentalmente di due tipi:
 - di **fare**: di un oggetto che comprendono:
 - fare qualcosa esso stesso, come per esempio creare un oggetto o
eseguire un calcolo;
 - chiedere ad altri oggetti di eseguire azioni;
 - controllare e coordinare le attività di altri oggetti;
 - di **conoscere**: di un oggetto che comprendono:
 - conoscere i propri dati privati encapsulati;
 - conoscere gli oggetti correlati;
 - conoscere cose che può derivare o calcolare;
- 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 vengono definite classi, metodi e variabili con lo scopo di soddisfare le
responsabilità, le quali sono implementate per mezzo degli oggetti e metodi che
agiscono da soli oppure che collaborano con altri oggetti e metodi;
- la RDD porta a considerare un progetto OO come una comunità di oggetti con
responsabilità che collaborano;

la responsabilità è una
metafora, non c'è
davvero nel software

- la progettazione guidata dalle responsabilità viene fatta, iterativamente, come segue:
 - identifica le responsabilità, e considerale una alla volta;
 - chiediti a quale oggetto software assegnare questa responsabilità, potrebbe essere un oggetto tra quelli già identificati, oppure un nuovo oggetto;
 - chiediti come fa l'oggetto scelto a soddisfare questa responsabilità, potrebbe fare tutto da solo, oppure collaborare con altri oggetti (l'identificatore di una collaborazione porta spesso ad identificare nuove responsabilità da assegnare).

Pattern: codificazione di principi e idiomì in un formato strutturato che descrive il problema e la soluzione a cui è assegnato un nome.

Un pattern è una coppia problema/soluzione ben conosciuta e con un nome, che può essere applicata in nuovi contesti, con consigli su come applicarla in situazioni nuove e con una discussione sui relativi compromessi, implementazioni, variazioni e così via.

Pattern GRASP: sono un aiuto per l'apprendimento degli aspetti essenziali della progettazione ad oggetti e per l'applicazione dei ragionamenti di progettazione in modo metodico, razionale e spiegabile. Sono uno strumento utile per acquisire padronanza delle basi di dati dell'OOD e a comprendere l'assegnazione di responsabilità nelle progettazione ad oggetti.

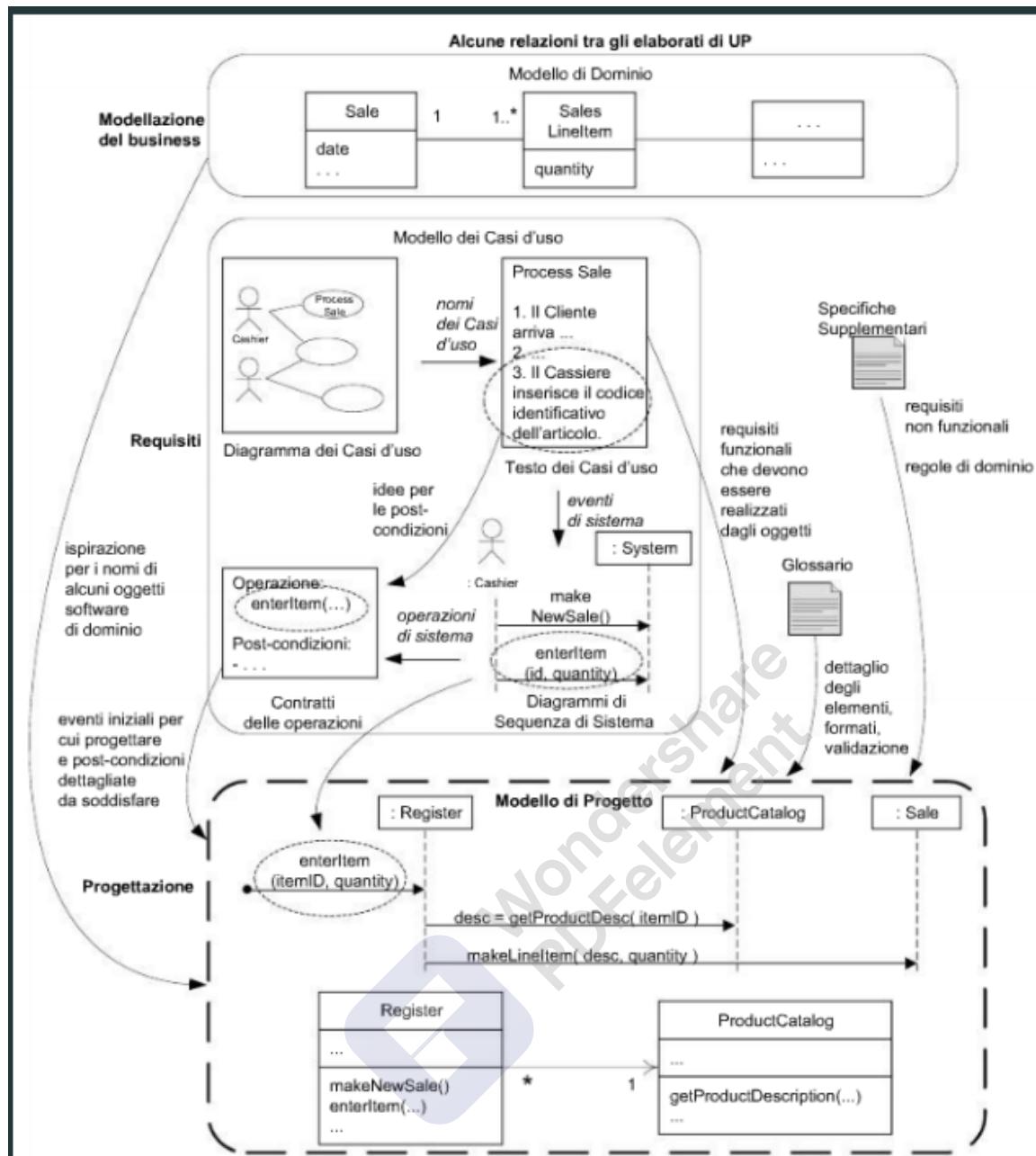
Low Representational Gap: principio nella fase di progettazione tra il modo in cui si pensa al dominio e una corrispondenza diretta con gli oggetti software. Va sempre guardato il modello di dominio per trarre ispirazione. piccolo salto rappresentazionale

Principio di GRASP (e dei principi della progettazione del software): 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: comprendizione, 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 mantenendo le funzionalità implementate (per esempio, a fronte di cambiamenti nei requisiti). Comprensibilità e facilità 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) coesi e debolmente accoppiati.

Output della progettazione ad oggetti

Ecco dove siamo
arrivati

I nove pattern GRASP (faremo solo i primi 5):

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

Pattern Creator

chi deve creare un oggetto

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):

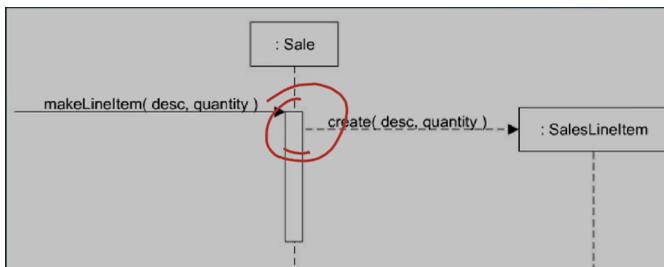
- B "contiene" o aggrega con una composizione oggetti di tipo A
- B registra A
- B utilizza strettamente A
- B possiede i dati per l'inizializzazione di A, che saranno passati ad A al momento della sua creazione (pertanto B è un Expert rispetto alla creazione di A). ha una variabile o un campo con riferimento a quell'oggetto -> allora ci chiediamo se possiamo metterlo direttamente lì

OSSERVAZIONI

- Trovare creatore che abbia veramente bisogno di essere collegato all'oggetto creato (low coupling), nell'esempio precedente utilizzato composto/contenitore;
- Si devono usare classi di supporto (ovvero pattern non-GRASP più complicati 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.

Sale è un buon candidato ad avere la responsabilità di creare istanze di SalesLineItems

Ciò richiede che nella classe sales sia definito un metodo in cui avviene la creazione di un oggetto SalesLineItem, per esempio un metodo makeLineItem



Pattern Expert**Nome:**

chi deve avere la responsabilità di fare o conoscere qualcosa

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.

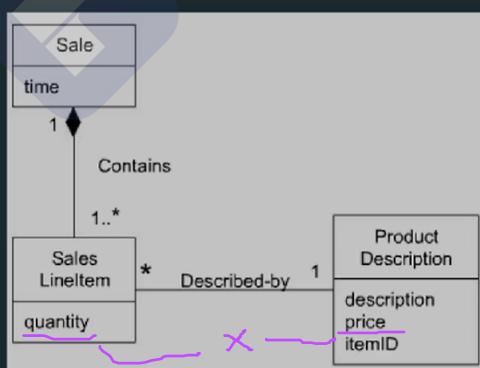
Una responsabilità necessita di informazioni per essere soddisfatta: informazioni su altri oggetti, sullo stato di un oggetto, sul mondo che circonda l'oggetto, informazioni che l'oggetto può ricavare, ...

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" (Peter Coad): gli oggetti software, a differenza di quelli reali, hanno la responsabilità di compiere delle azioni sulle cose che conoscono.

Esempio per Expert

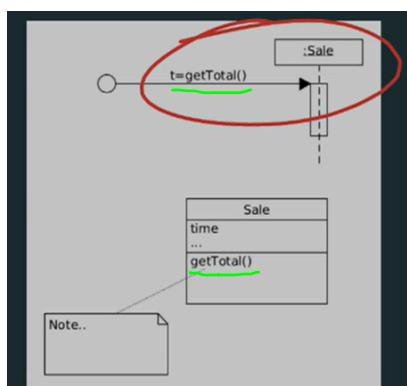
Secondo Expert, Sale è la migliore candidata: occorre conoscere tutte le istanze SalesLineItem della vendita e la somma dei relativi totali parziali. Un'istanza Sale li contiene, è un esperto delle informazioni per questo compito.



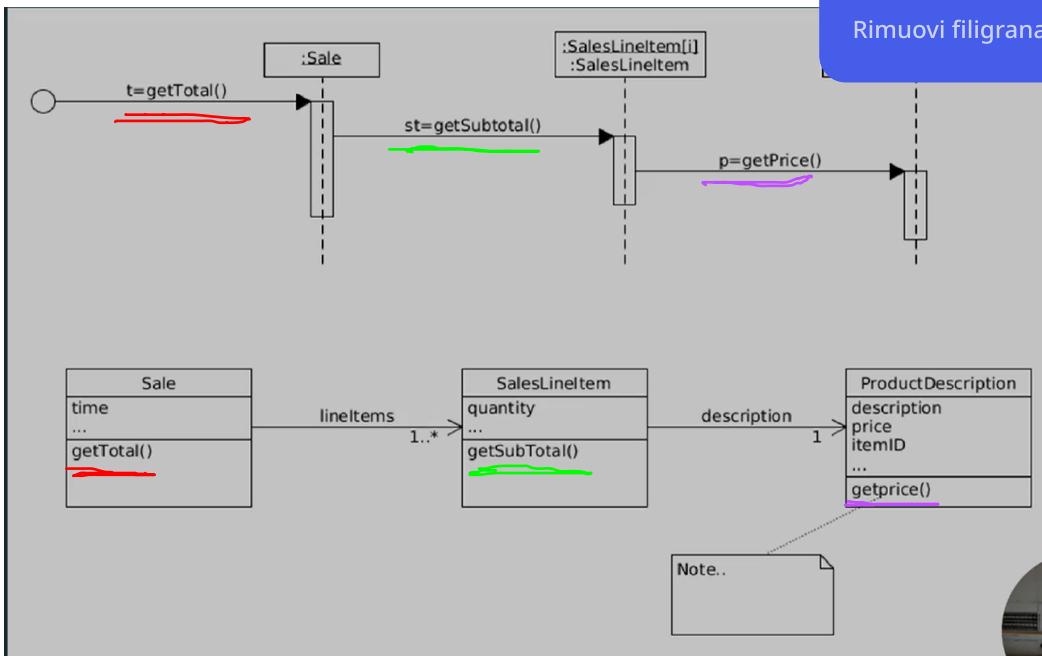
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Se il registratore di cassa dovesse farlo dovrebbe essere accoppiato a saleLineItem

Nota: quello riportato è il Modello di Dominio.



Classe di progetto	Responsabilità
Sale	sa calcolare il totale della vendita; conosce le righe di vendita della vendita
SalesLineItem	sa calcolare il totale parziale della riga di vendita; conosce il prodotto della riga di vendita
ProductDescription	conosce il prezzo del prodotto



Abbiamo analizzato questo pattern, processo mentale che facciamo già in automatico con l'esperienza!! Spesso ci basterà il modello di dominio

limita il numero di legami e dipendenze tra le classi

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 maggior 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 (o 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 cambiamenti locali anche in queste classi;
- queste classi sono più difficili da comprendere in isolamento, ovvero senza comprendere anche le classi da cui dipendono; leggendo il codice
- 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 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 richama 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;
- 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, o per loro pura e semplice presenza.

l'ereditarietà non è un buon principio di uso software, l'OO ha un meccanismo più potente, la delega, usare il metodo di un'altra.

Interfacce vanno bene, l'ereditarietà molto meno perché è un accoppiamento forte.

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.

Pattern High Cohesion**Nome:**

High Cohesion (Coesione Alta)

per mantenere facilmente una classe devo renderla coesa

devo mettere in una classe solo i metodi correlati tra loro e che assolvono a un obiettivo comune

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 una 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 scorrelate o se svolge troppo lavoro;
- Grady Booch: c'è una coesione funzionale alta quando gli elementi di un componente (es. classe) "lavorano tutti insieme per fornire un comportamento ben circoscritto".

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 lettere in poche aree diverse, che sono logicamente correlate al concetto rappresentato dalla classe, ma non l'una all'altra.

Regola pratica:

Una classe con coesione alta ha un numero di metodi relativamente basso, con delle funzionalità altamente correlate e focalizzate, e non fa troppo lavoro.

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

VANTAGGI

- High Cohesion sostiene 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 ne e altamente correlate, poiché una classe se coesa può essere usata per uno scopo molto specifico.

classe coesione alta
se DELEGA il lavoro

Pattern Controller**Nome:**

Controller (Controllore)

il controller è un oggetto artificiale che viene creato, introdotto, e non appartiene né allo strato di dominio né allo strato ui

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:

- 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);
- 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).

Nel caso di controller di caso d'uso o controller di sessione: un controller per ogni caso s'uso

- si utilizzi la stessa classe controller per tutti gli eventi di sistema nello stesso scenario di caso d'uso; favorisce l'high cohesion
 - 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.
- un ssd è un'istanza di conversazione

OSSERVAZIONI

- il controller è un pattern di delega: gli oggetti dello stato UI catturano gli eventi di sistema generati dagli attori e devono delegare le richieste di lavoro ad oggetti di un altro sistema;
- il controller coordina o controlla le attività, ma non esegue di per sé molto lavoro.

coordinano le operazioni di sistema e le delegano alle classi del dominio, possono controllare che gli

Corollario: eventi avvengano in un ordine preciso (la makePayment deve essere fatta dopo la endSale)

gli oggetti UI non devono avere la responsabilità di soddisfare gli eventi di sistema. In un'applicazione le operazioni di sistema devono essere gestite nello stato degli oggetti della logica applicativa o del dominio anziché nello stato UI.

Controller MVC: fa parte dell'interfaccia grafica e cattura l'evento actionPerformed

fa parte della UI e gestisce l'interazione con l'utente; la sua implementazione dipende in larga misura dalla tecnologia UI e dalla piattaforma che viene utilizzata.

Controller GRASP:

fa parte dello stato del dominio e controlla o coordina la gestione delle richieste delle operazioni di sistema. Non dipende dalla tecnologia UI utilizzata.

Esempio: l'interfaccia grafica preleva i dati per la enterItem e li passa come parametri

Facade controller: esegue il lavoro ma lo delega ad altri oggetti

rappresenta il sistema complessivo, una facciata sopra agli altri strati dell'applicazione e fornisce un punto di accesso principale per le chiamate dei servizi dallo strato UI agli altri strati sottostanti.

Controller dei casi d'uso:

un controller per ogni caso d'uso

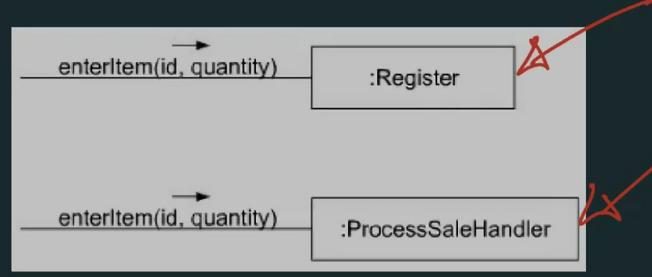
C. MVC vs GRASP

entrambe gestiscono le richieste degli utenti ma a livelli di astrazione diversi, il controller MVC delega le richieste di lavoro dell'utente al controlle GRASP del dominio

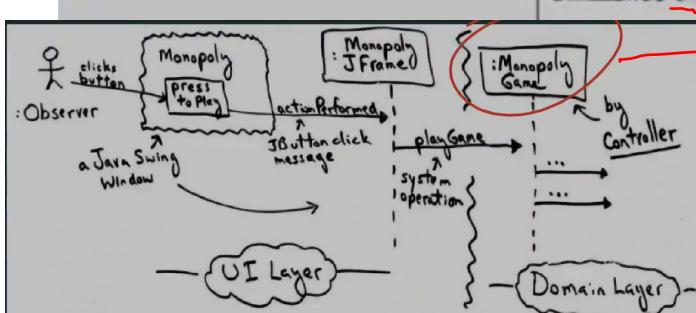
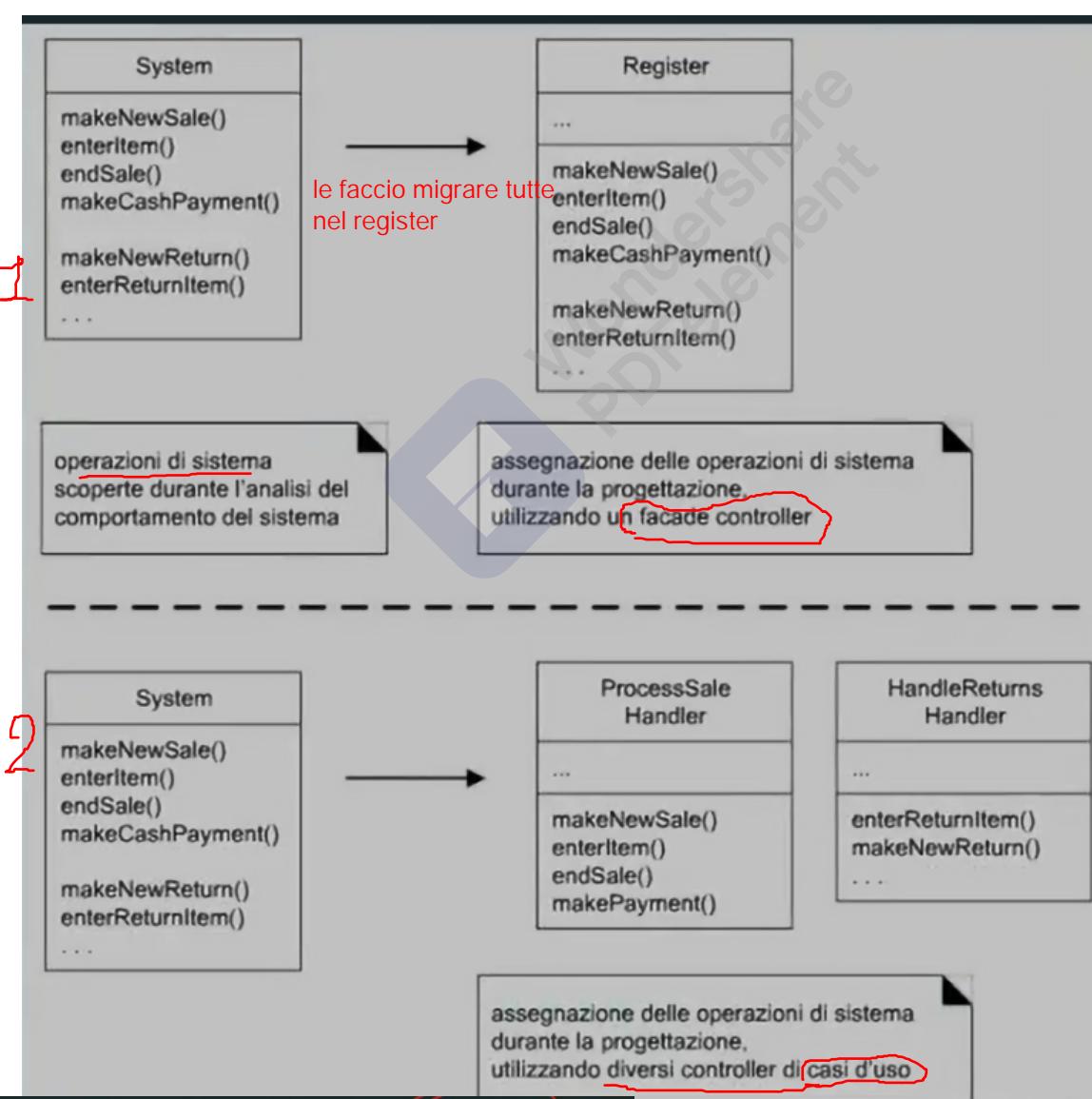
Nel software chi deve controllare *enterItem* e *endSale*?

- un oggetto che rappresenta il "sistema complessivo",
l'"oggetto radice", il dispositivo o il punto di accesso al software o un sottosistema: *POSSystem*, *Register*, *POSTerminal*?
- un oggetto di classe controller di caso d'uso: *ProcessSalehandler*, *ProcessSaleSession*?

Due possibili soluzioni.



©C. Larman. Applicare UML e i Pattern. Pearson, 2016.



il controller grasp individuato permette di connettere lo strato UI allo strato del dominio

potrebbe essere un oggetto radice che rappresenta il sistema complessivo (se ci sono poche operazioni possibili) OPPURE un oggetto che rappresenta un dispositivo in cui è eseguito il software (come register) OPPURE un oggetto che rappresenta il caso d'uso o la sessione

VANTAGGI

- **Maggiore potenziale di riuso e interfacce inseribili:**

La delega delle responsabilità delle operazioni di sistema a un controller favorisce il riuso della logica in altre applicazioni future ed è inoltre possibile utilizzarla con un'interfaccia diversa (o molte diverse allo stesso tempo).

- **Opportunità di ragionare sullo stato del caso d'uso:**

E' possibile assicurarsi che le operazioni di sistema si susseguano in una sequenza legale, oppure si desidera ragionare sullo stato corrente dell'attività e delle operazioni all'interno del caso d'uso in corso di esecuzione (sessione).

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

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

Il controller memorizza anche i dati della sessione (che sia il register o un controllore specifico è uguale): ad esempio endSale termina la vendita corrente, dove viene salvata la vendita corrente? non fa parte dei campi del dominio, viene salvata nel controller. Delega poi ai componenti dello strato di dominio l'esecuzione delle operazioni (High cohesion)

nei sistemi più complessi i controller possono essere separate dallo strato di dominio ed essere in uno strato application a parte

```

package com.craiglarman.nextgen.ui.swing;
import ...
// in Java, una JFrame è una tipica finestra
public class ProcessSaleJFrame extends JFrame {
    ① // la finestra ha un riferimento all'oggetto 'controller' del dominio
    private Register register;
    // alla creazione della finestra viene passato il controller
    public ProcessSaleJFrame(Register r) {
        register = r;
    }
    // si fa clic su questo pulsante per eseguire
    // l'operazione di sistema "enterItem"
    private JButton BTN_ENTER_ITEM;
    // crea il pulsante per eseguire enterItem
    // questo è il metodo importante!
    // qui viene mostrato il messaggio dallo strato UI
    // allo strato del dominio
    private JButton getBTN_ENTER_ITEM() {
        // il pulsante esiste già?
        if (BTN_ENTER_ITEM != null) {
            return BTN_ENTER_ITEM;
        }
        // altrimenti il pulsante deve essere inizializzato...
        BTN_ENTER_ITEM = new JButton();
        BTN_ENTER_ITEM.setText("Enter Item");
        // QUESTA È LA SEZIONE CHIAVE!
        // in Java, questo è il modo per definire
        // il gestore del clic per un pulsante
        BTN_ENTER_ITEM.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // Transformer è una classe di utilità per
                // trasformare le stringhe in altri tipi di dati
                // perché gli elementi JTextField della GUI
                // gestiscono solo stringhe
                ItemID id = Transformer.toItemID(getTXT_ID().getText());
                int qty = Transformer.toInt(getTXT_QTY().getText());
                // qui si supera il confine tra lo strato UI
                // e lo strato del dominio delegando al 'controller'
                // >>> QUESTA È L'ISTRUZIONE CHIAVE <<<
                register.enterItem(id, qty);
            }
        } ); // fine della chiamata a addActionListener
        return BTN_ENTER_ITEM;
    } // fine del metodo
    ...
} // fine della classe

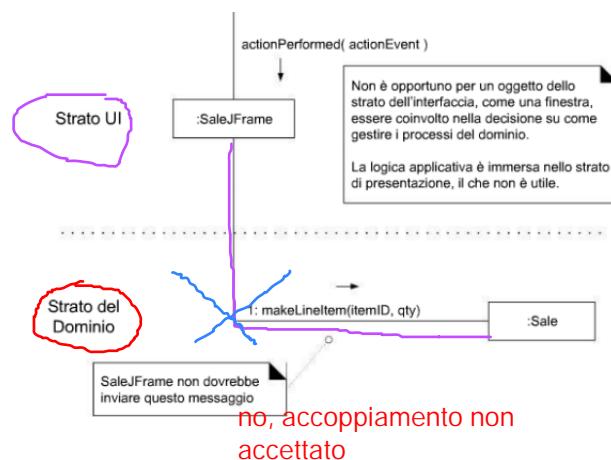
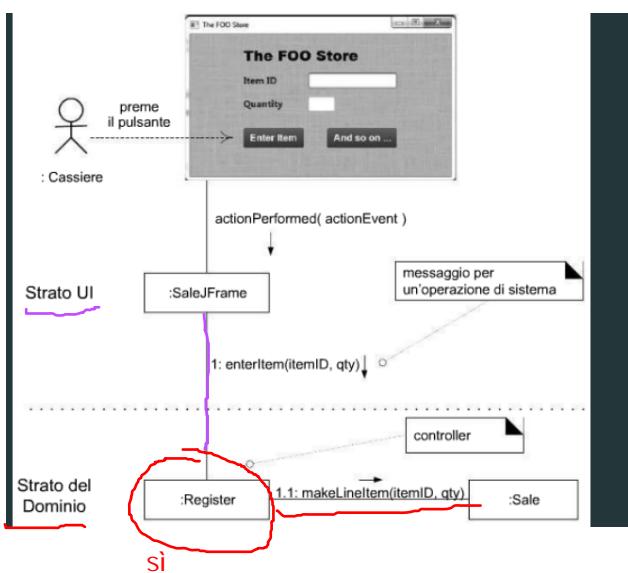
```

- (1) La finestra *ProcessSaleJFrame* ha un riferimento all'oggetto *controller* del dominio, *Register*.
 (2) si definisce l'oggetto per gestire il clic del pulsante.
 (3) l'invio del messaggio *enterItem* al controller nello strato del dominio.

UI

C. GRASP

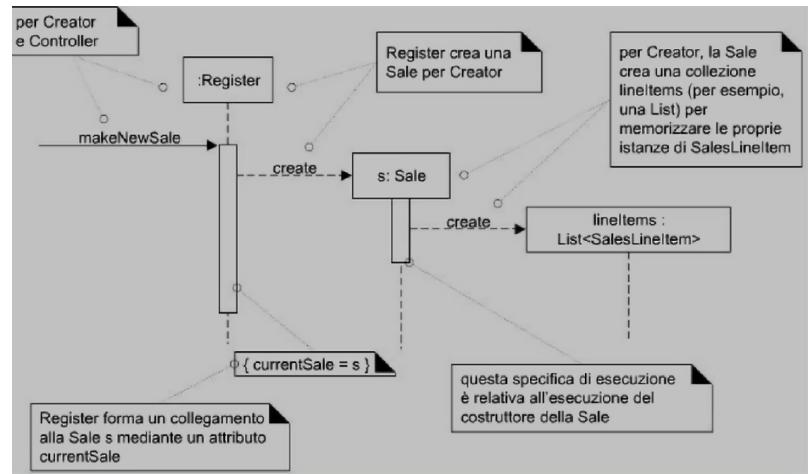
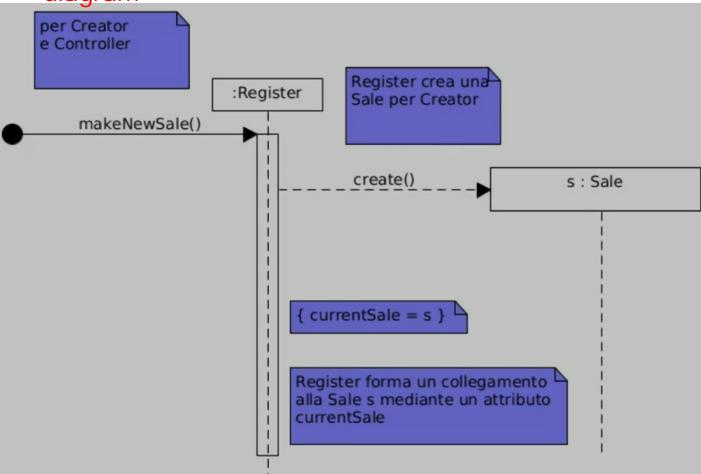
il click è l'evento (dell'ssd)

Controller MVC riceve
evento clickquesta è l'operazione di
sistema (che deve
gestire l'evento dell'
utente)delega al controllore
grasp

ESEMPI DI PROGETTAZIONE CON GRASP

vedere file pdf → “EsempioDiProgettazioneConGRASP” (moodle: corso SAS a.a. 20/21)

diagramma di interazione sequence diagram



class diagram

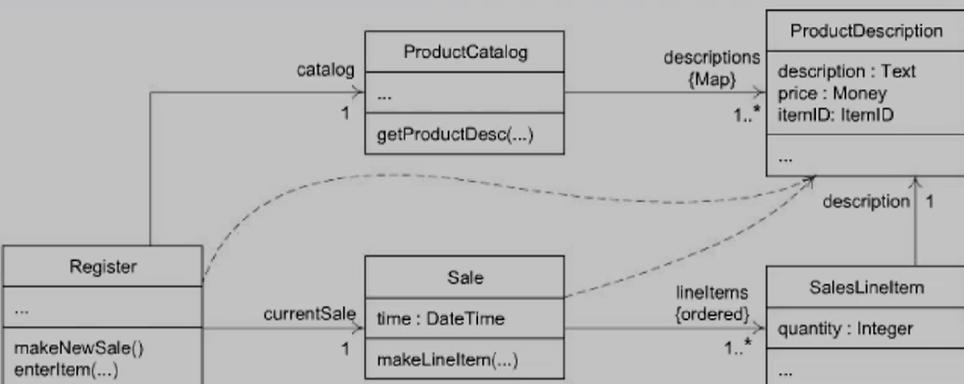
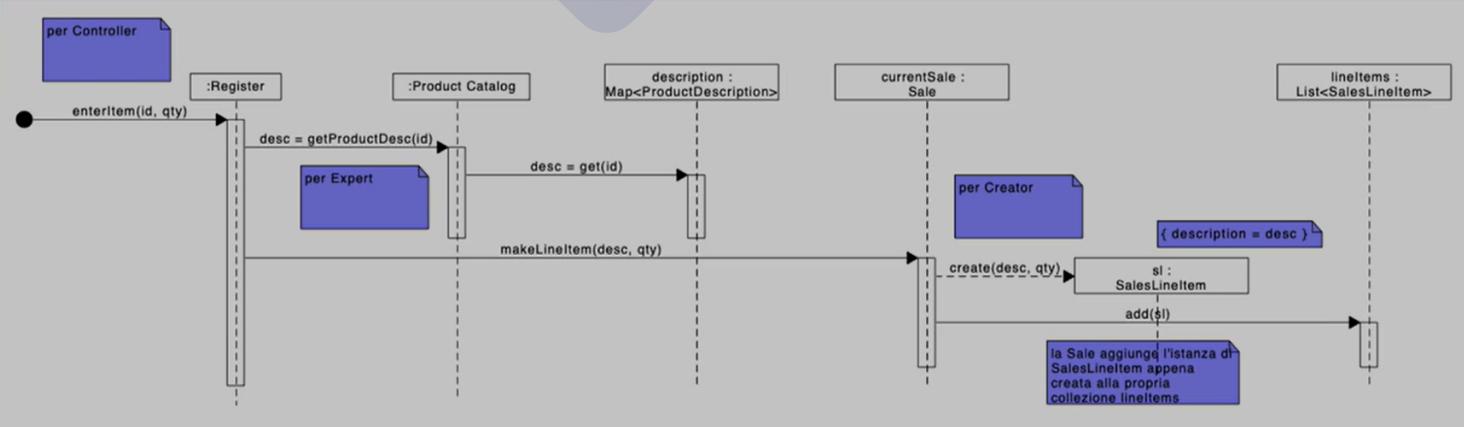


La nuova *SalesLineitem* va collegata a una *ProductDescription* che corrisponde all'*itemID* inserito.

Ciò implica la ricerca di una *ProductDescription* in base a una corrispondenza con *itemID*.

Responsabilità

Chi deve essere responsabile della conoscenza di una *ProductDescription* in base a una corrispondenza con *itemID*?



GoF Gang of Four
dai 4 creatori

sono schemi di
progettazione avanzata

DESIGN PATTERN GoF

I GoF sono:

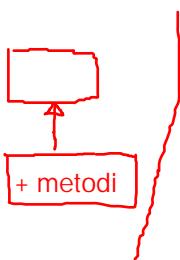
- i 23 pattern per la programmazione OO descritti nel libro "Design Pattern" di Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides;
- più degli "schemi di progettazione avanzata" che "principi" (come nel caso dei GRASP):
- ciascun design pattern descrive una soluzione progettuale comune a un problema di progettazione ricorrente.

I design pattern GoF sono classificati in base al loro scopo:

- creazionale:** risolvono problematiche inerenti l'istanziazione degli oggetti (Abstract Factory e Singleton);
- strutturale:** risolvono problematiche inerenti alla struttura delle classi e degli oggetti (Adapter, Composite e Decorator);
- comportamentale:** forniscono soluzioni alle più comuni tipologie di interazione tra gli oggetti (Observer, State, Strategy e Visitor).

GoF preferisce la composizione rispetto all'ereditarietà tra classi perché aiuta a mantenere la classi incapsulate e coese. La delegazione permette di rendere la composizione tanto potente quanto l'ereditarietà!

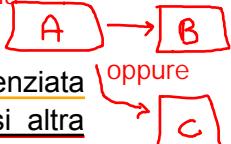
- Ereditarietà di classi:**



VS

- Composizione degli oggetti:**

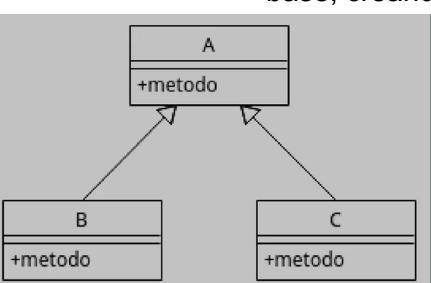
- la funzionalità sono ottenute assemblando o componendo gli oggetti per avere funzionalità più complesse;
- riuso black-box: i dettagli interni non sono conosciuti;
- se una classe usa un'altra classe, questa potrebbe essere referenziata attraverso una interfaccia, a runtime potrebbe esserci una qualsiasi altra classe che implementa l'interfaccia;
- la composizione attraverso un'interfaccia rispetta l'incapsulamento, solo una modifica all'interfaccia comporterebbe ripercussioni.



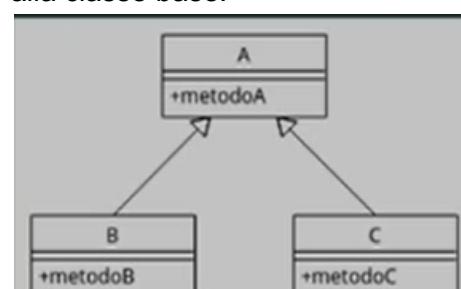
ci sono due concetti di EREDITARIETÀ nell'OO che vanno ben distinti

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

- Polimorfismo:** le sottoclassi possono essere scambiate l'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 specializzate rispetto alla classe base.



Polimorfismo: classi ridefiniscono metodi, non ne aggiungono.
Sotto classi possono essere scambiate una con l'altra



Specializzazione:
sotto classi aggiungono elemen

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

GoF CREAZIONALI

Pattern Abstract Factory

Nome:

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 concrete che la estendono.

Caratteristiche:

- 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:
 - di assicurarsi che il cliente crei soltanto prodotti vincolati fra di loro;
 - l'utilizzo di diverse famiglie di prodotti da parte dello stesso cliente;
- è usata nelle librerie Java per la creazione di famiglie di elementi GUI per diversi sistemi operativi e sottoinsiemi GUI.

Struttura del pattern

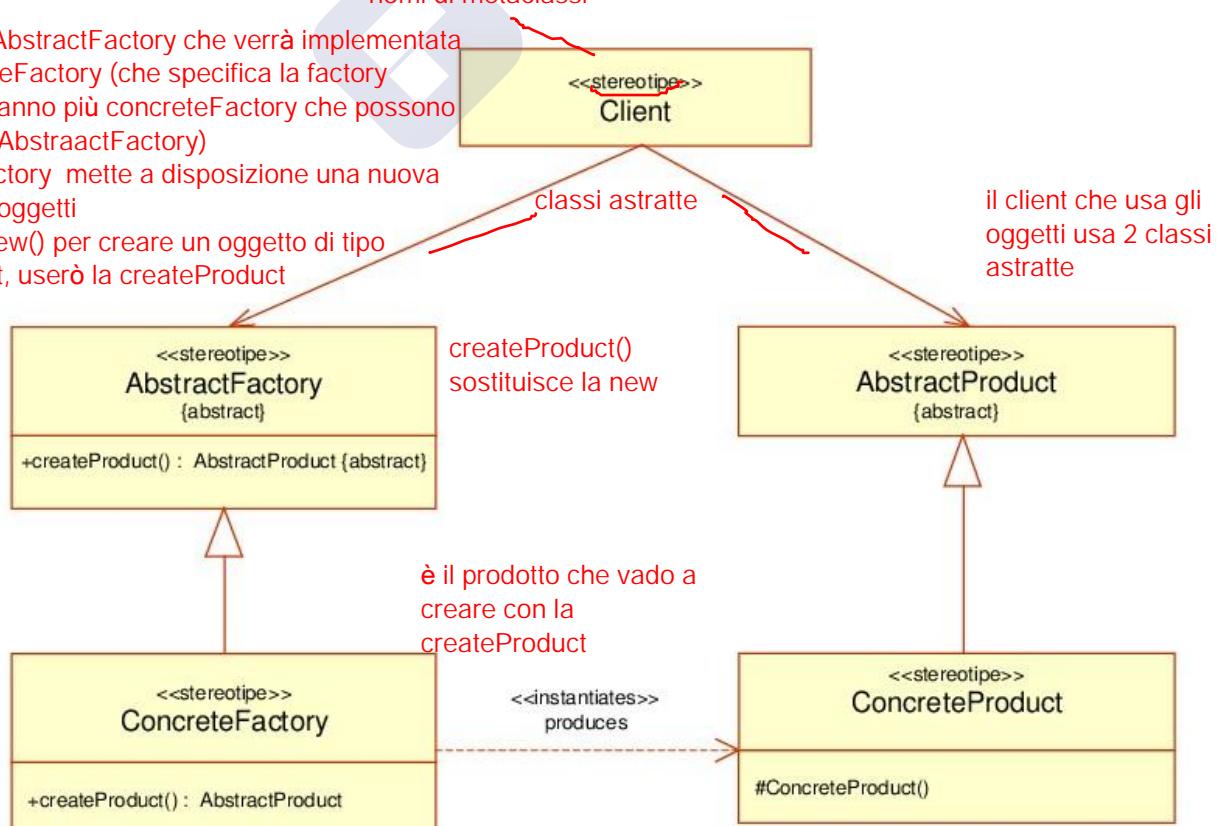
stereotype perché i nomi scritti non sono nomi di classi ma nomi di metaclassi

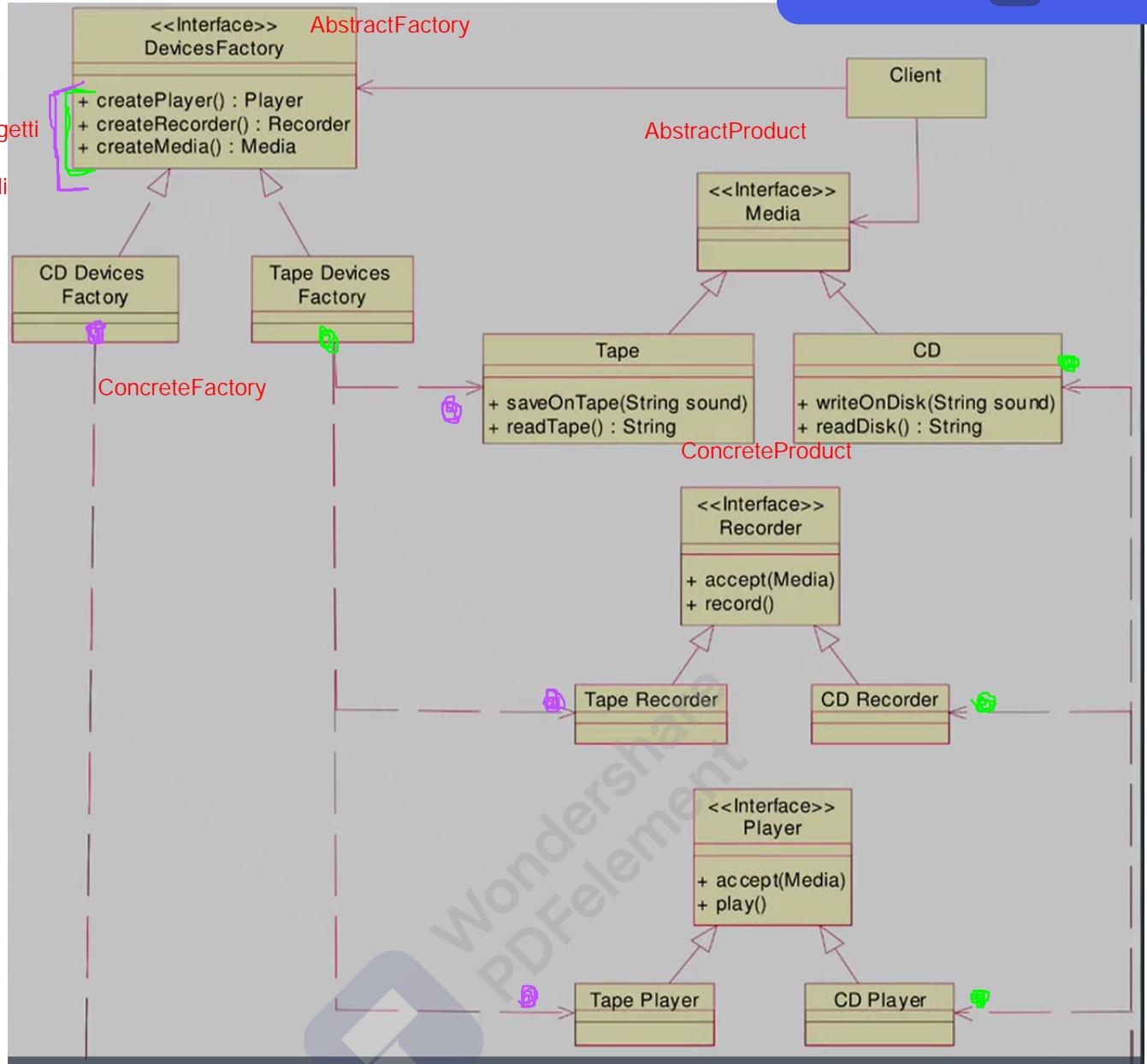
lo schema è:

istanziare una AbstractFactory che verrà implementata da una ConcreteFactory (che specifica la factory concreta, ci saranno più concreteFactory che possono implementare l'AbstractFactory)

La concrete Factory mette a disposizione una nuova new per questi oggetti

invece che la new() per creare un oggetto di tipo abstractProduct, userò la createProduct





E' una soluzione di POLIMORFISMO:

Il client, anziché fare delle new sui concrete product (Tape, Tape Recorder, CD Player ecc) fa delle new sulla factory astratta, che col binding dinamico prende la factory concreta istanziata .

Quindi si evita la cascata di if es: if hai creato un tape then crea un tape recorder.

E al posto, il binding dinamico in automatico se l'oggetto è di questo tipo crea le istanze di questo tipo (è l'interprete che verifica)

```

public class AbstractFactoryExample {

    public static void main ( String[] arg ) {

        Client client = new Client(); creato il client

        System.out.println( "***Testing tape devices" );
        client.selectTechnology( new TapeDevicesFactory() ), nel client imposto la tecnologia utilizzata (cd o nastro).
        client.test( "I wanna hold your hand..." );

        System.out.println( "***Testing CD devices" );
        client.selectTechnology( new CDDevicesFactory() ); ConcreteFactory passata come oggetto.
        client.test( "Fly me to the moon..." );
    }
}
  
```

il client invoca test e usa gli oggetti istanziati coordinati senza accorgersi

```

class Client {
    DevicesFactory technology;

    public void selectTechnology( DevicesFactory df ) {
        technology = df;
    }

    public void test(String song) {
        Media media = technology.createMedia();
        Recorder recorder = technology.createRecorder();
        Player player = technology.createPlayer();

        recorder.accept( media );
        System.out.println( "Recording the song : " + song );
        recorder.record( song );
        System.out.println( "Listening the record:" );
        player.accept( media );
        player.play();
    }
}

```

La classe Client definisce gli oggetti che utilizzano i prodotti d'ogni famiglia.

Il metodo selectTechnology riceve un oggetto corrispondente ad una famiglia particolare e lo registra dentro i propri attributi.

Il metodo test crea una istanza d'ogni particolare tipo di prodotto e applica i diversi metodi forniti dalle interfacce che implementano i prodotti.

Si deve notare che il cliente utilizza i prodotti, senza avere conoscenza di quali sono concretamente questi.

invece di scrivere:

if tape, if cd ecc then vai a creare
Media Recorder e Player corretti

si va a mettere il then nelle varie
classi e gli if spariscono perché
sostituiti dal bindng dinamico

Pattern Singleton

Nome:

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.

Caratteristiche:

- 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.

Tra le diverse implementazioni in java:

- Singleton come classe statica:** non è vero e proprio 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 creato un metodo statico che deve essere chiamato per restituire l'istanza del Singleton. L'oggetto verrà istanziato solo la prima volta. Le successive saranno restituiti un riferimento allo stesso oggetto (inizializzazione pigra);
- Singleton multi-thread:** versione multi-thread della soluzione precedente.

Il Singleton creato da un metodo statico è preferibile rispetto al Singleton come classe statica per tre motivi:

- 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.

ATTENZIONE problematiche derivanti anche da java

- presenza di singleton in virtual machine multiple;
- singleton caricati contemporaneamente da diversi class loader;
- singleton distrutti dal garbage collection e dopo caricati quando sono necessari;
- presenza di istanze multiple come sottoclassi di un singleton;
- copia di singleton come risultato di un doppio processo di deserializzazione.

Struttura del pattern

<<stereotype>>
Singleton

-instance:Singleton
-Singleton()
+getNewInstance():Singleton

a) Singleton come classe statica

E' il modello più semplice, ma che in realtà non è un vero e proprio **Singleton**, perché soltanto si lavora con una classe statica, non un oggetto [14], [17]. Questa classe statica ha metodi statici che offrono i servizi richiesti.

è un trucco

```
public static class PrinterSpooler {
    private PrinterSpooler() {
    }

    public static void print (String msg) {
        System.out.println( msg );
    }
}
```

Si noti che il costruttore è dichiarato privato, per evitare l'istanziazione di oggetti della classe.

L'invocazione all'oggetto sarà una istruzione simile a:

```
| PrinterSpooler.print( somethingToPrint );
```

istanziazione avverà solo al primo utilizzo al caricamento del class loader

b) Singleton creato da un metodo statico

Il **Singleton** è implementato come una classe che ha un metodo statico (getInstance) che deve essere chiamato per restituire l'istanza del **Singleton**. L'oggetto **Singleton** verrà istanziato solo la prima volta che il metodo sia invocato, in modo che eventuali informazioni necessarie per creare il **Singleton** possano essere fornite in tempo di esecuzione. Le veci successive sarà restituito un riferimento allo stesso oggetto.

```
public class PrinterSpooler {
    private static PrinterSpooler instance;
    private PrinterSpooler() {
    }

    public static PrinterSpooler getInstance() {
        if ( instance==null ) {
            instance = new PrinterSpooler();
        }
        return instance;
    }

    public void print (String msg) {
        System.out.println( msg );
    }
}
```

getInstance va a sostituire la new() che non è più possibile in quanto costruttore privato, se è la prima getInstance viene creato l'oggetto, altrimenti viene restituito l'oggetto già istanziato

In questa implementazione il costruttore continua ad essere statico, per costringere all'utilizzo del metodo `getInstance()`, per ricavare gli oggetti:

nella versione multi thread il metodo getInstance è synchronized

```
...
PrinterSpooler theUnique = PrinterSpooler.getInstance();
theUnique.print( somethingToPrint );
//if we try this:
PrinterSpooler maybeOther = PrinterSpooler.getInstance();
//then...
if( theUnique != maybeOther ) → false!!!
```

GoF STRUTTURALI

Pattern Adapter

Nome:

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.

Caratteristiche:

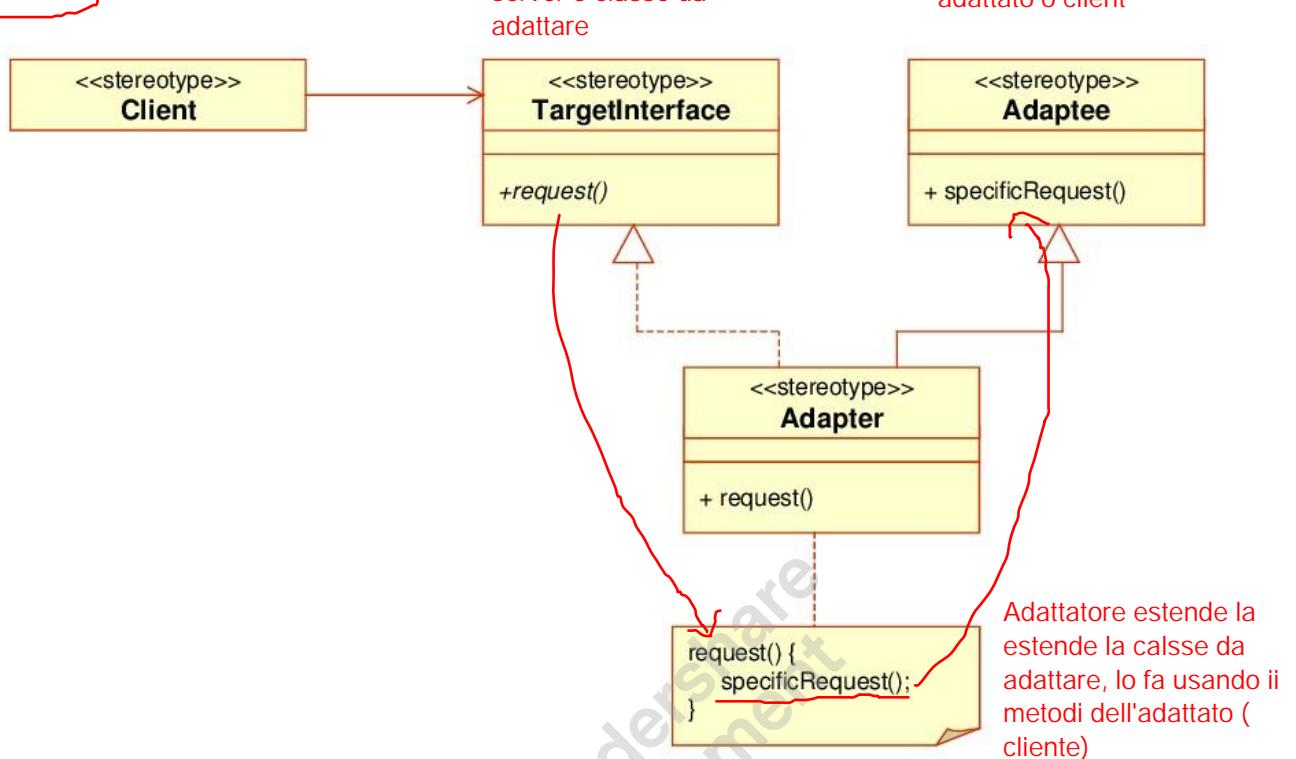
- si consideri una coppia di oggetti software in una relazione client-server. Si parla di interfacce incompatibili quando l'oggetto server offre servizi di interesse per l'oggetto client ma l'oggetto client vuole fruire di questi servizi in una modalità diversa da quella prevista dall'oggetto server (**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**).

Funzionamento:

- 1) un adattatore riceve richieste da suoi client, per esempio da un oggetto dello stesso dominio, nel formato client dell'adattatore;
- 2) l'adattatore poi adatta, trasforma, una richiesta ricevuta in una richiesta nel formato del server;
- 3) l'adattatore invia la richiesta al server;
- 4) se il server fornisce una risposta, lo fa nel formato del server;
- 5) l'adattatore adatta, trasforma, la risposta ricevuta dal server in una risposta nel formato del client e poi la restituisce al suo client.

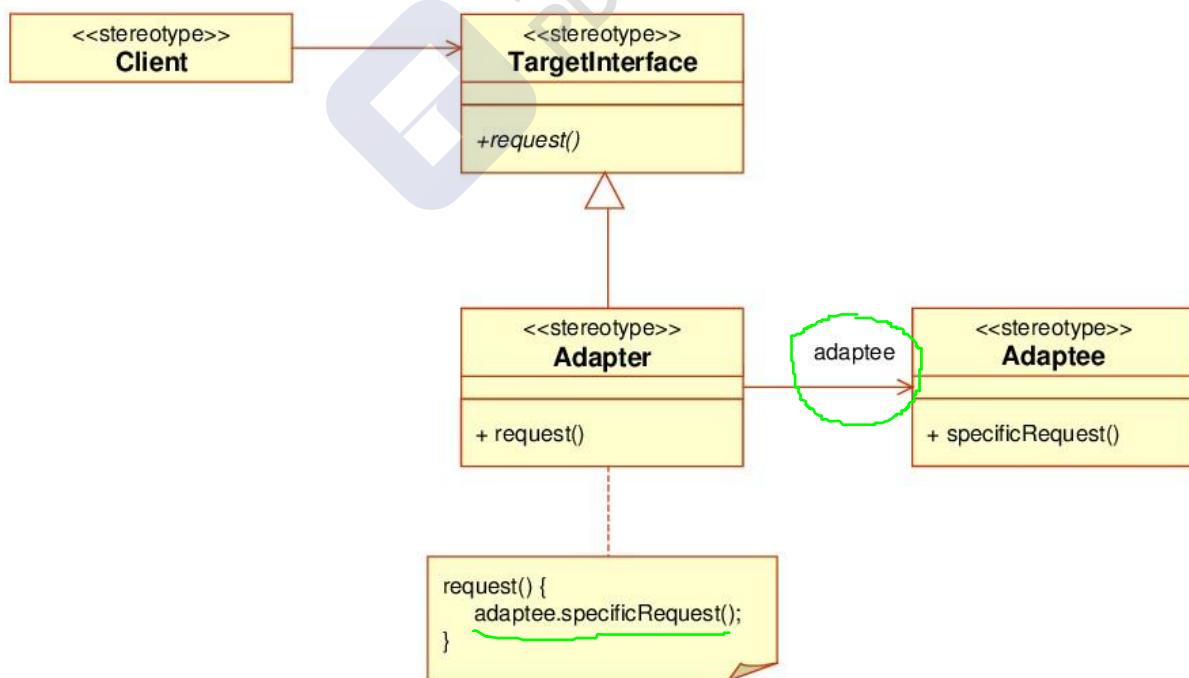
Struttura del pattern

Class



Object

Un altro modo migliore, in cui al posto che estendere lo incapsula, delega all'adattato



a) Implementazione come Class Adapter

La costruzione del Class Adapter per il Rectangle è basato nella sua estensione. Per questo obiettivo viene creata la classe RectangleClassAdapter che estende Rectangle e implementa l'interfaccia Polygon:

```
public class RectangleClassAdapter extends Rectangle implements Polygon{
    private String name = "NO NAME";
    public void define( float x0, float y0, float x1, float y1,
                        String color ) {
        float a = x1 - x0;
        float l = y1 - y0;
        setShape( x0, y0, a, l, color );
    }
    public float getSurface() {
        return getArea();
    }
    public float[] getCoordinates() {
        float aux[] = new float[4];
        aux[0] = getOriginX();
        aux[1] = getOriginY();
        aux[2] = getOppositeCornerX();
```



meglio perché ci libera di un extends



meglio anche per il
discorso di riuso white
box black box

b) Implementazione come Object Adapter

La costruzione dell'Object Adapter per il Rectangle, si basa nella creazione di una nuova classe (RectangleObjectAdapter) che avrà al suo interno un'oggetto della classe Rectangle, e che implementa l'interfaccia Polygon:

```
public class RectangleObjectAdapter implements Polygon {
    Rectangle adaptee;
    private String name = "NO NAME";
    public RectangleObjectAdapter() {
        adaptee = new Rectangle();
    }
    public void define( float x0, float y0, float x1, float y1,
                        String col ) {
        float a = x1 - x0;
        float l = y1 - y0;
        adaptee.setShape( x0, y0, a, l, col );
```



L'oggetto vuole essere visto sia come
composto che come componente.

Composite evita di chiedersi se si è una
componente atomica o un composto, lo fa
fare al binding dinamico

Pattern Composite

Nome:

Composite

Problema:

Come trattare un gruppo o una struttura composta di oggetti dello stesso tipo (polimorficamente quindi anche sottotipo) 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,

sia foglia che branch devono implementare albero (la quale ha operazioni come getChild())

Funzionalità:

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:

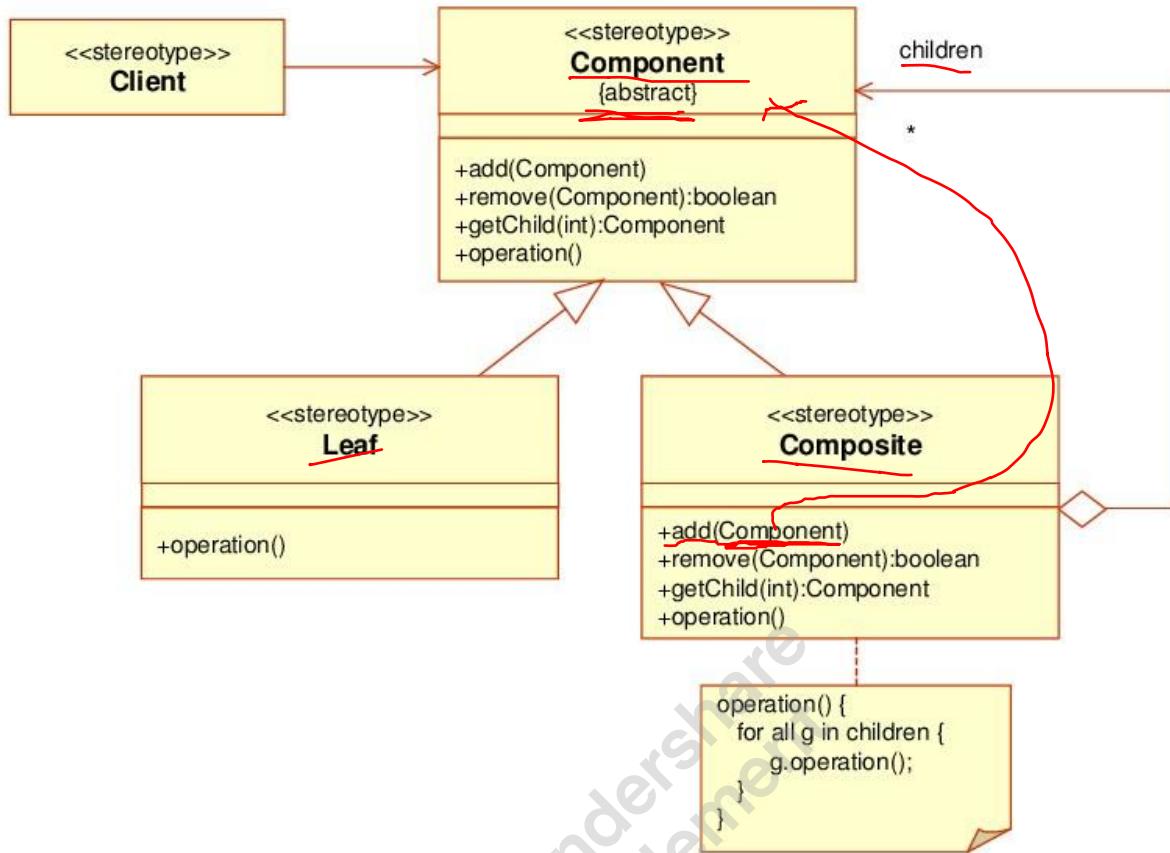
- una che rappresenta i singoli componenti (foglie);
- l'altra che rappresenta i componenti composti e che si implementa come contenitore di componenti;

N.B.

i componenti composti possono immagazzinare sia componenti singoli che altri contenitori

un nodo (oggetto composto) può immagazzinare sia una foglia (oggetto singolo) che altri nodi (oggetti composti)

Struttura del pattern



Il pattern composite permette di costruire strutture ricorsive (ad esempio un albero di elementi) in modo che ad un cliente (una classe che usa la struttura) l'intera struttura sia vista come una singola entità. Quindi l'interfaccia alle entità atomiche (foglie) e esattamente la stessa dell'interfaccia delle entità composte. In essenza tutti gli elementi della struttura hanno la stessa interfaccia senza considerare se sono composti o atomici.

```
public abstract class Tree {  
    public abstract boolean empty();  
    public abstract int getRootElement();  
    public abstract void stampaPostVisita();  
}  
  
public class TestTree {  
    public static void main(String[] args) {  
        ...  
        System.out.println("Stampo albero postvisita");  
        t.stampapostvisita();  
        System.out.println("");  
    }  
}
```

```
public class Leaf extends Tree {
    public Leaf() { }
    public boolean empty() { return true; }
    public int getRootElement() {
        assert false;
        return 0;
    }
    public void stampapostvisita() { }
}

public class Branch extends Tree {
    private int elem;
    private Tree left;
    private Tree right;
    public Branch(int elem, Tree left,
Tree right) {
        this.elem = elem;
        this.left = left;
        this.right = right;
    }
    public boolean empty() { return false; }
    public int getRootElement() { return elem; }
    public void stampapostvisita() {
        right.stampapostvisita();
        left.stampapostvisita();
        System.out.print(this.getRootElem() + " ");
    }
}
```

L'applicazione CompositeExample fa le veci del **Client** che gestisce i diversi pezzi singoli e composta un oggetto composto (mainSystem) con tre di questi oggetti singoli. L'oggetto composto appena creato serve, a sua volta, per creare, insieme ad altri pezzi singoli, un nuovo oggetto composto (computer). L'applicazione invoca poi il metodo `describe()` su un oggetto singolo, sull'oggetto composto soltanto da pezzi singoli, e sull'oggetto composto da pezzi singoli e pezzi composti. Finalmente fa un tentativo di aggiungere un componente ad un oggetto corrispondente a un pezzo singolo.

```
public class CompositeExample {
    public static void main(String[] args) {
        // Creates single parts
        Component monitor      = new SinglePart("LCD Monitor");
        Component keyboard     = new SinglePart("Italian Keyboard");
        Component processor   = new SinglePart("Pentium III Processor");
        Component ram         = new SinglePart("256 KB RAM");
        Component hardDisk    = new SinglePart("40 Gb Hard Disk");

        // A composite with 3 leaves
        Component mainSystem = new CompoundPart( "Main System" );
        try {
            mainSystem.add( processor );
            mainSystem.add( ram );
            mainSystem.add( hardDisk );
        }
        catch (SinglePartException e){
            e.printStackTrace();
        }

        // A Composite compound by another Composite and one Leaf
        Component computer = new CompoundPart("Computer");
        try{
            computer.add( monitor );
            computer.add( keyboard );
            computer.add( mainSystem );
        }
        catch (SinglePartException e){
            e.printStackTrace();
        }

        System.out.println("**Tries to describe the 'monitor' component");
        monitor.describe();
        System.out.println("**Tries to describe the 'main system' component");
        mainSystem.describe();
        System.out.println("**Tries to describe the 'computer' component");
        computer.describe();

        // Wrong: invocation of add() on a Leaf
        System.out.println( "***Tries to add a component to a single part
                           (leaf) " );
        try{
            monitor.add( mainSystem );
        }
        catch (SinglePartException e){
            e.printStackTrace();
        }
    }
}
```

```
public abstract class Component {
    public String name;

    public Component(String aName){
        name = aName;
    }

    public abstract void describe();

    public abstract void add(Component c) throws SinglePartException;
    public abstract void remove(Component c) throws SinglePartException;
    public abstract Component getChild(int n);
}
```

E la classe `SinglePart` dovrebbe implementare il codice riguardante tutti i metodi dichiarati astratti nella superclasse:

```
public class SinglePart extends Component {
    public SinglePart(String aName) {
        super(aName);
    }

    public void add(Component c) throws SinglePartException{
        throw new SinglePartException();
    }

    public void remove(Component c) throws SinglePartException{
        throw new SinglePartException();
    }

    public Component getChild(int n) {
        return null;
    }

    public void describe(){
        System.out.println( "Component: " + name );
    }
}
```

Pattern Decorator

Nome:

Decorator

aggiungere dei metodi

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

vogliamo evitare relazioni statiche

Soluzione:

Inglobare l'oggetto all'interno di un altro che aggiunge altre funzionalità

creando un wrapper

l'extends in pratica implementa un wrapper, noi vogliamo fare la stessa cosa ma per aggiungere responsabilità solo ad un oggetto e non a tutti

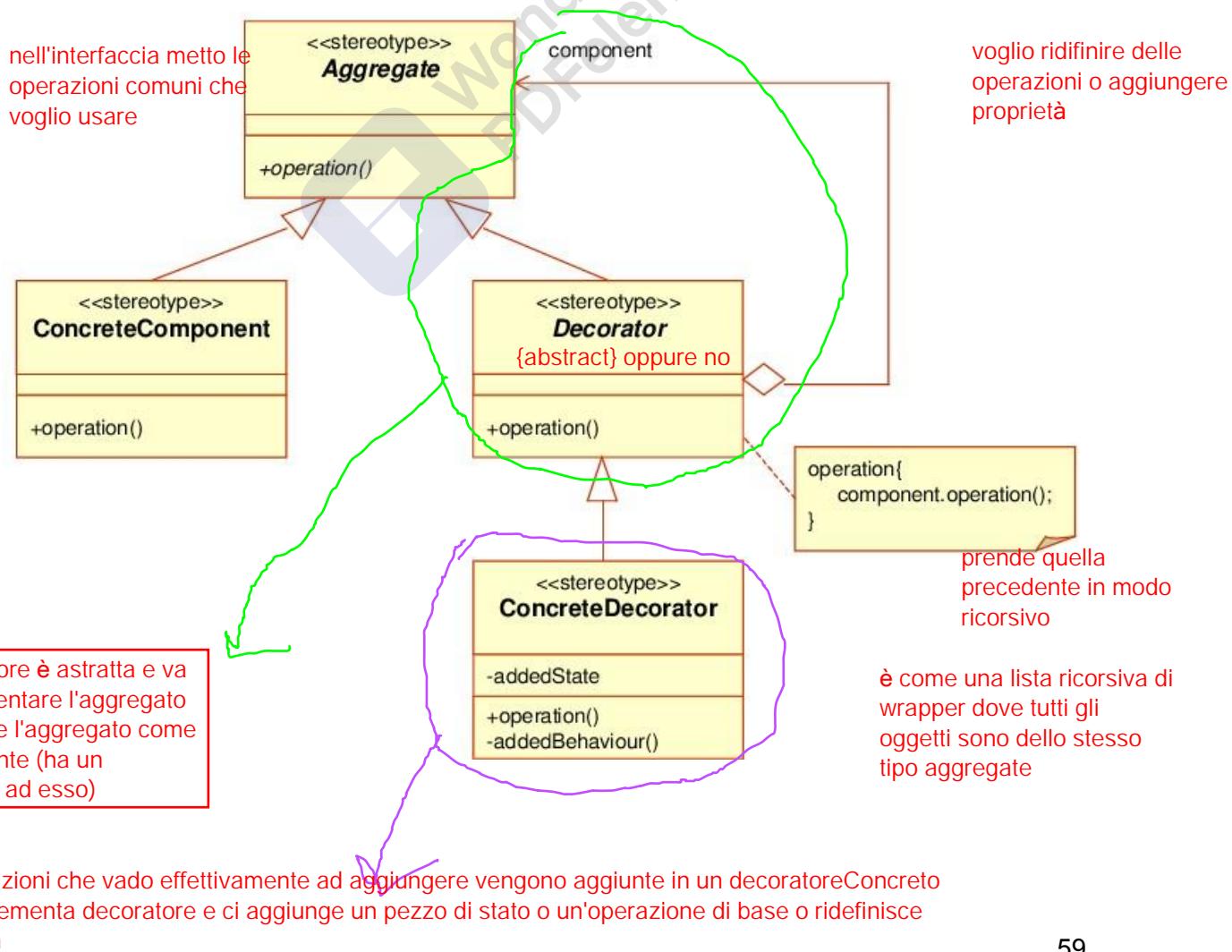
Vantaggi:

- 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 sottoclassi per supportare un ampio numero di estensioni e combinazioni di esse oppure quando le definizioni sono nascoste e non disponibili alle sottoclassi.

potrei avere tantissime sottoclassi statice che sono vincolanti, dipendenti l'una dall'altra

Struttura del pattern

I voglio vedere sia la classe originale che quella a cui ho aggiunto proprietà allo stesso modo, come la stessa classe in modo da poter invocare le stesse operazioni su tutti



9.5. Applicazione del pattern

Schema del modello

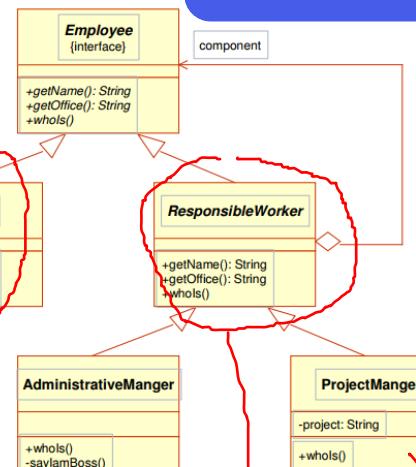
Partecipanti

- Component:** classe Employee.
 - Specifica l'interfaccia degli oggetti che possono avere delle responsabilità aggiunte dinamicamente.
- ConcreteComponent:** classe Engineer.

```
abstract class ResponsibleWorker implements Employee {
    protected Employee responsible;
    public ResponsibleWorker(Employee employee) {
        responsible = employee;
    }
    public String getName() {
        return responsible.getName();
    }
    public String getOffice() {
        return responsible.getOffice();
    }
    public void whoIs() {
        responsible.whoIs();
    }
}
```

L'ingegnere è l'elemento di base su cui poi si va a costruire, è un impiegato che è l'interfaccia comune

proprietà in più che vogliamo aggiungere all'ingegnere



Un responsibleWorker è un' interfaccia su cui andremo ad aggregare tutte le responsabilità in più che vengono date dinamicamente a questo ingegnere

funziona come un extend di classe ma uno può diventare (dinamicamente) sia un projectManager che un administrativeManager

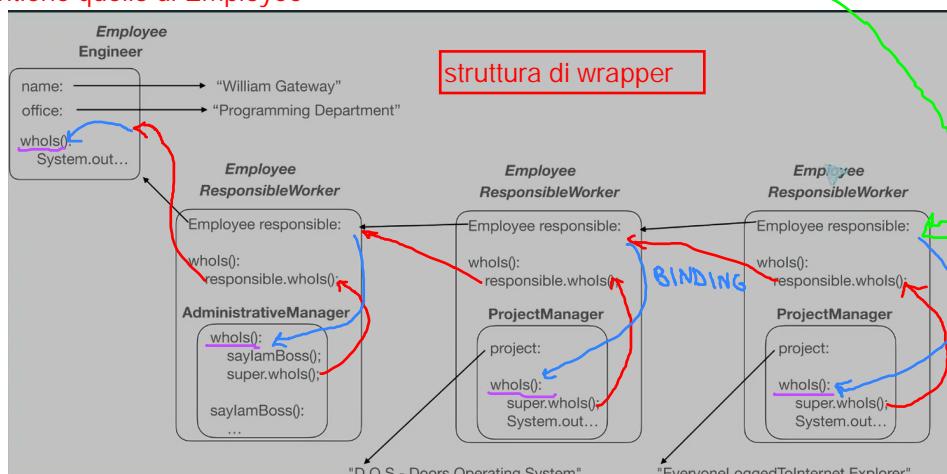
```
public class DecoratorExample1 {
    public static void main(String arg[]) {
        Employee thisWillBeFamous = new Engineer("William Gateway", "Programming Department");
        System.out.println("Who are you?");
        thisWillBeFamous.whoIs();
        thisWillBeFamous = new AdministrativeManager(thisWillBeFamous);
        System.out.println("Who are you now?");
        thisWillBeFamous.whoIs();
        thisWillBeFamous = new ProjectManager(thisWillBeFamous, "D.O.S.- Doors Operating System");
        System.out.println("Who are you now?");
        thisWillBeFamous.whoIs();
        thisWillBeFamous = new ProjectManager(thisWillBeFamous, "EveryoneLoggedToInternet Explorer");
        System.out.println("Who are you now?");
        thisWillBeFamous.whoIs();
    }
}
```

baldoni@Shizuka:~/Nextcloud/MaterialeCorsi/aa2122/SAS/GoF/GoF Strutturali/05-Decorator\$ java DecoratorExample1
Who are you?
I am William Gateway (Engineer@816f27d), and I am with the Programming Department.
Who are you now?
I am a boss. I am William Gateway (Engineer@816f27d), and I am with the Programming Department.
Who are you now?
I am a boss. I am William Gateway (Engineer@816f27d), and I am with the Programming Department.
I am the Manager of the Project:D.O.S.- Doors Operating System
Who are you now?
I am a boss. I am William Gateway (Engineer@816f27d), and I am with the Programming Department.
I am the Manager of the Project:D.O.S.- Doors Operating System
I am the Manager of the Project:EveryoneLoggedToInternet Explorer
baldoni@Shizuka:~/Nextcloud/MaterialeCorsi/aa2122/SAS/GoF/GoF Strutturali/05-Decorator\$

E' sempre lo stesso ingegnere, mantiene interfaccia comune !!! Con la struttura statica dell' extends avrei ottenuto 4 oggetti diversi (ENG, ADM, PM)

esattamente come quando si estende andiamo a costruire una gerarchia di oggetti e quando andiamo a prendere super viaggiamo su questi link.

Costruiamo così una struttura ricorsiva e per essere tale tutti gli elementi devono venire visti come employee, è una lista in cui ogni elemento contiene il puntatore al precedente elemento, ad esempio AdministrativeManager contiene il puntatore di ResponsibleWorker che contiene quello di Employee



Il pattern decorator permette ad una entità di contenere completamente un'altra entità così che l'utilizzo del decoratore sia identico all'entità contenuta. Questo consente al decoratore di modificare il comportamento e/o il contenuto di tutto ciò che sta encapsulato senza cambiare l'aspetto esteriore dell'entità. Ad esempio, è possibile utilizzare un decoratore per aggiungere l'attività di logging dell'elemento contenuto senza cambiare il comportamento di questo.

Composite vs Decorator

- Il *pattern composite*: fornisce un'interfaccia comune a elementi atomici (foglie) e composti
- Il *pattern decorator*: fornisce caratteristiche addizionali ad elementi atomici (foglie), mantenendo un'interfaccia comune

GoF **COMPORTAMENTALI**

Pattern **Observer**

Nome:

Observer (o Dependents o Publish-Subscribe)

Problema:

Diversi tipi di oggetti subscriber (abbonato) sono interessi ai cambiamenti di stato o agli eventi di un oggetto publisher (editore). 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. Che cosa fare?

Soluzione:

Definisci un'interfaccia subscriber o listener (ascoltatore). 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.

Caratteristiche:

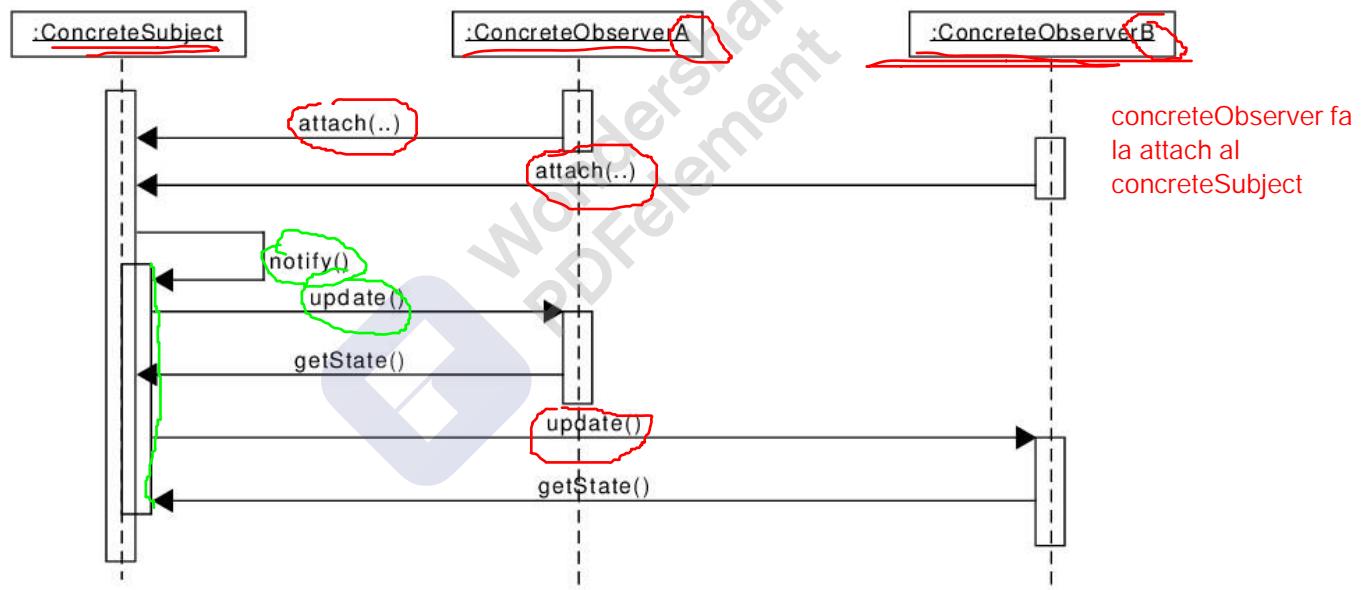
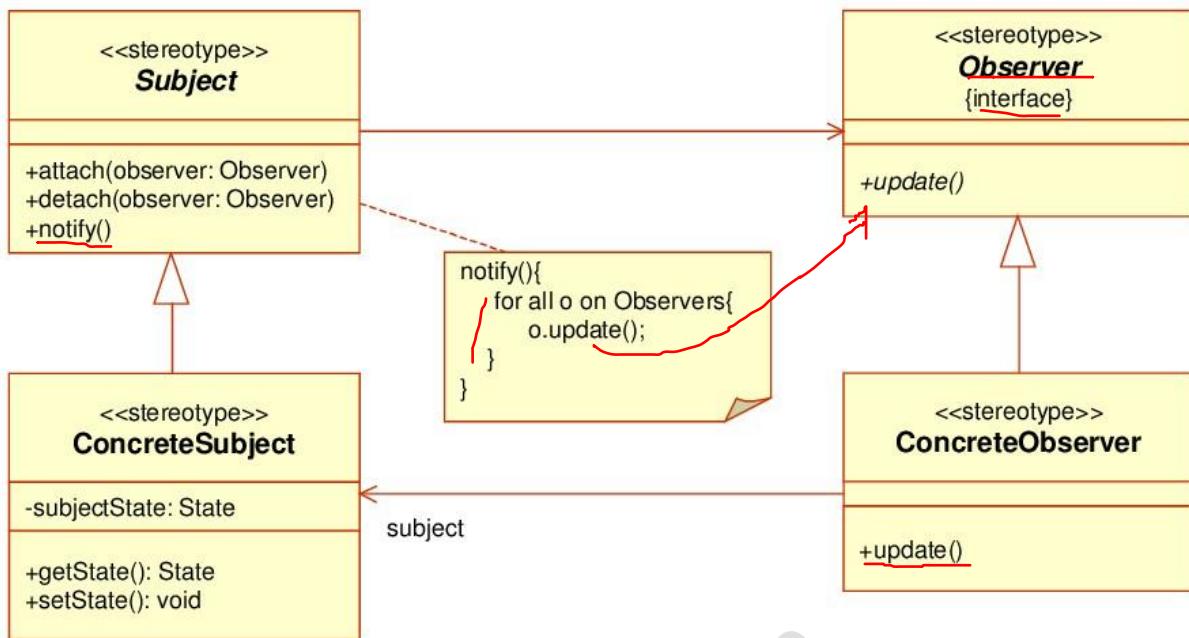
- definisce una dipendenza tra oggetti di tipo uno-a-molti: quando lo stato di un oggetto cambia, tale evento viene notificato a tutti gli oggetti dipendenti, essi vengono automaticamente aggiornati;
- l'oggetto che notifica il cambiamento di stato non fa alcuna assunzione sulla natura degli oggetti 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 maniera debole degli oggetti che devono comunicare (eventi). I publisher conoscono i subscriber solo attraverso un'interfaccia, e i subscriber possono registrarsi (o cancellare la propria registrazione) dinamicamente con i publisher.

N.B.

spesso associato al pattern MVC (Model-View-Controller): le modifiche al modello sono notificate agli osservatori che sono le viste

inserisco una mail nella lista di mails della mailbox che aggiorna in automatico la ListView del client

Struttura del pattern



Pattern State**Nome:**

State

Problema:

Il comportamento di un oggetto dipende da un 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?

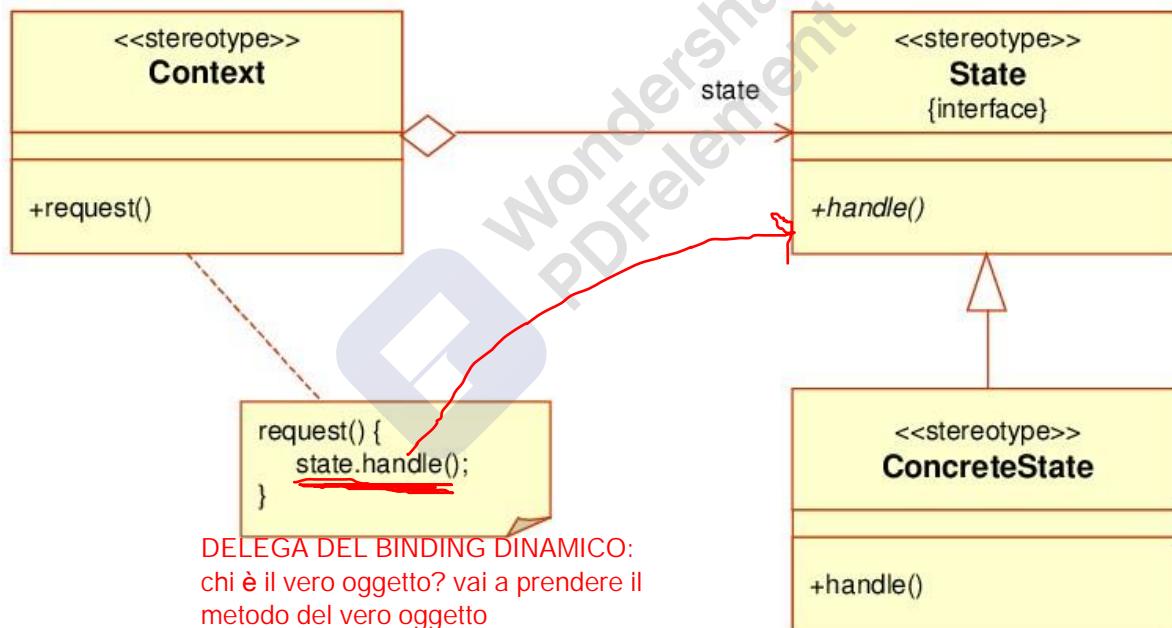
Soluzione:

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

Al posto degli if sullo stato, a seconda dello stato in cui mi trovo vado a prendere col binding dinamico le operazioni dello stato concreto

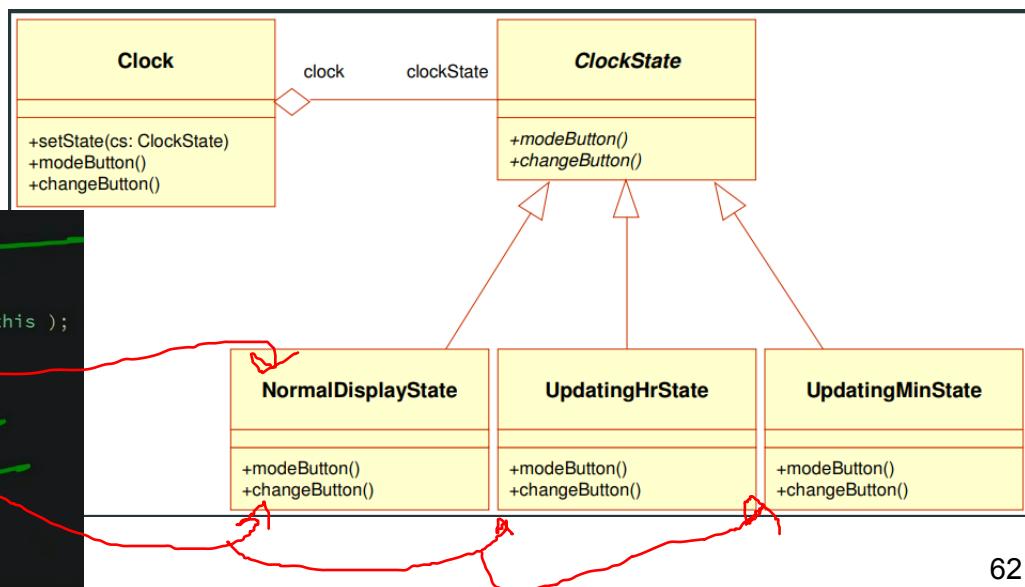
i suoi metodi delegano allo stato corrente

Struttura del pattern

Non devo cambiare il codice del clock !

```

public class Clock {
    private ClockState clockState;
    public int hr, min;
    public Clock() {
        clockState = new NormalDisplayState( this );
    }
    public void setState( ClockState cs ) {
        clockState = cs;
    }
    public void modeButton() {
        clockState.modeButton();
    }
    public void changeButton() {
        clockState.changeButton();
    }
    public void showTime() {
    }
}
  
```



Pattern Strategy

Nome:

Strategy (o 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. sono correlati da scopi comuni o obiettivi da raggiungere comuni, ad esempio il processo di pagamento con contanti o con carta di credito oppure gli algoritmi di ordinamento

Caratteristiche:

- 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; evita accoppiamento statico nel codice
- 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 interfaccia.

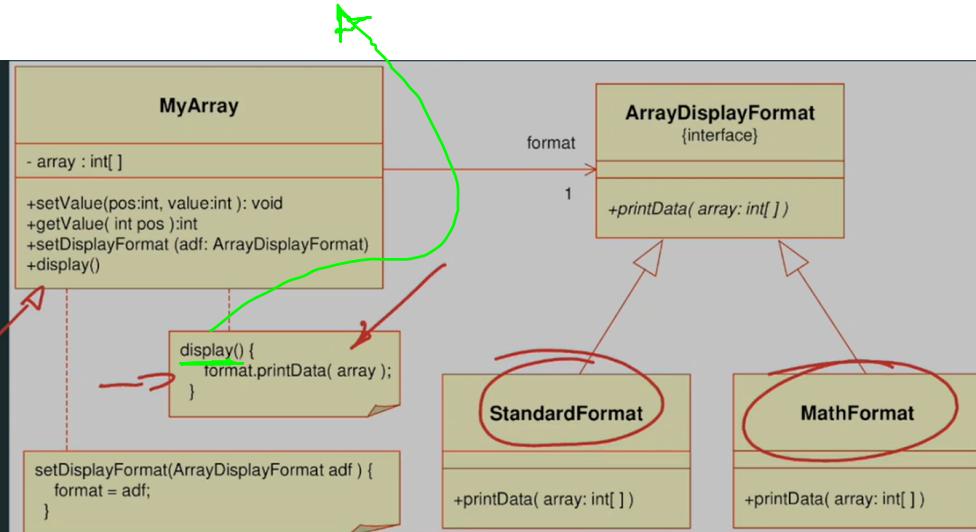
il binding dinamico mi permette di non scrivere gli if: if(carta di credito), if(contanti)

Otteniamo così generale è più riusabile

Struttura del pattern



display() rimane invariato e viene eseguita una print piuttosto che un'altra a seconda dell'algoritmo che ho passato facendo la setDisplayFormat()



vogliamo che lo stesso codice permetta di stampare in due modi l'array

```

MyArray m = new MyArray( 10 );
m.setValue( 1 , 6 );
m.setValue( 0 , 8 );
m.setValue( 4 , 1 );
m.setValue( 9 , 7 );
System.out.println("This is the array in 'standard' format");
m.setDisplayFormat( new StandardFormat() );
m.display();
System.out.println("This is the array in 'math' format:");
m.setDisplayFormat( new MathFormat() );
m.display();
  
```

Strategy vs State

incapsula un comportamento

- Il pattern **State** si occupa di che cosa (stato o tipo) un oggetto e (al suo interno) e incapsula un comportamento dipendente dallo stato. Fare cose diverse in base allo stato, lasciando il chiamante sollevato dall'onere di soddisfare ogni stato possibile;
- Il pattern **Strategy** si occupa del modo in cui un oggetto esegue un determinato compito: incapsula un algoritmo. Un'implementazione diversa che realizza (fondamentalmente) la stessa cosa, in modo che un'implementazione possa sostituire l'altra a seconda della strategia richiesta.

Pattern Visitor supponiamo che debba stampare un albero, sommare un albero, trovare il min, il max e

Nome: così via, come faccio? La struttura dati accetta un oggetto ConcreteVisitor che analizza
Visitor ogni nodo (oggetto) ricevuto e applica la funzione/algoritmo della famiglia

Problema:

Come separare l'operazione applicata su un contenitore complesso dalla struttura dati cui è applicata? Come 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 di elementi?

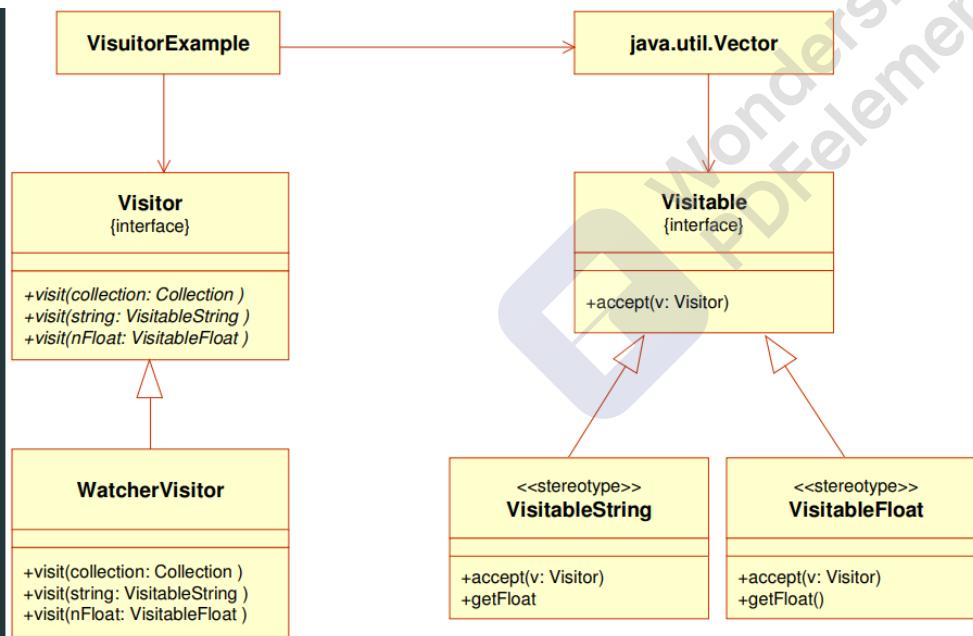
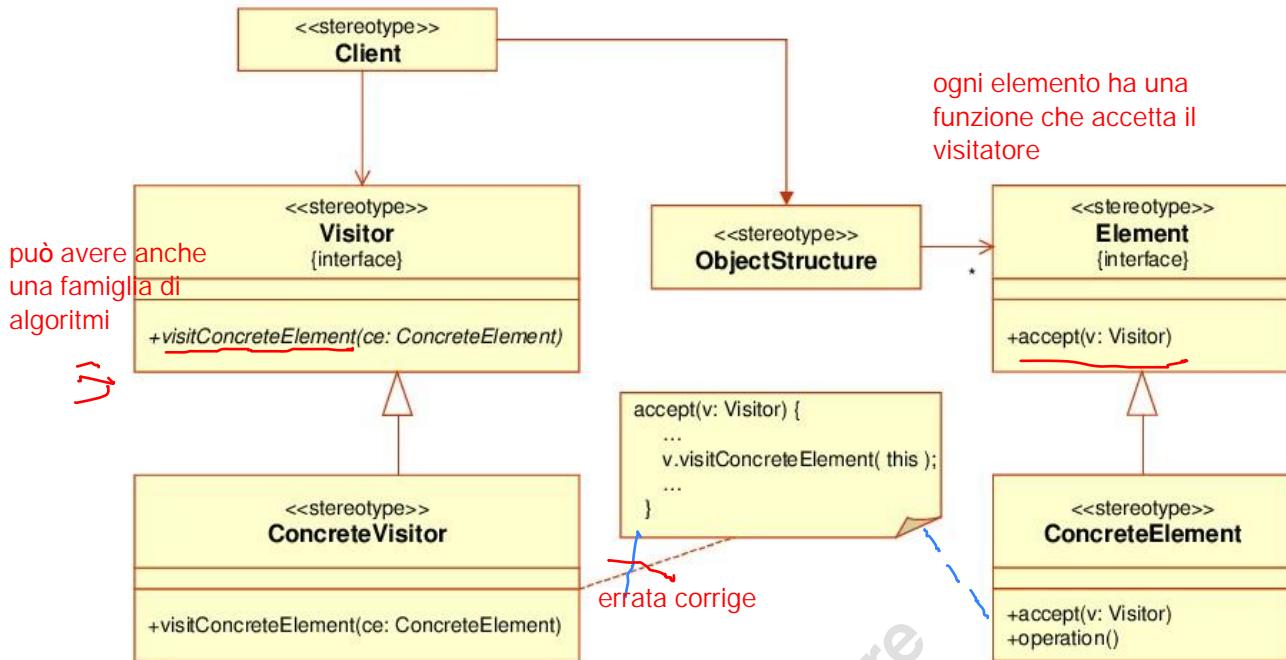
Soluzione: il visitatore contiene l'algoritmo da applicare al nodo

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.

Caratteristiche:

- 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.

Struttura del pattern



```

public static void main (String[] arg) {

    // Prepare a heterogeneous collection
    Vector untidyObjectCase = new Vector();
    untidyObjectCase.add( new VisitableString( "A string" ) );
    untidyObjectCase.add( new VisitableFloat( 1 ) );
    Vector aVector = new Vector();
    aVector.add( new VisitableString( "Another string" ) );
    aVector.add( new VisitableFloat( 2 ) );
    untidyObjectCase.add( aVector );
    untidyObjectCase.add( new VisitableFloat( 3 ) );
    untidyObjectCase.add( new Double( 4 ) );

    // Visit the collection
    Visitor browser = new WatcherVisitor();
    browser.visit( untidyObjectCase );
}

```

```

import java.util.Collection;
public interface Visitor {

    public void visit( Collection collection );
    public void visit( VisitableString string );
    public void visit( VisitableFloat nFloat );
}

```

```

public class VisitableString implements Visitable {
    private String value;

    public VisitableString(String string) {
        value = string;
    }

    public String getString() {
        return value;
    }

    public void accept( Visitor visitor ) {
        visitor.visit( this );
    }
}

public class VisitableFloat implements Visitable {
    private Float value;

    public VisitableFloat(float f) {
        value = new Float( f );
    }
}

```

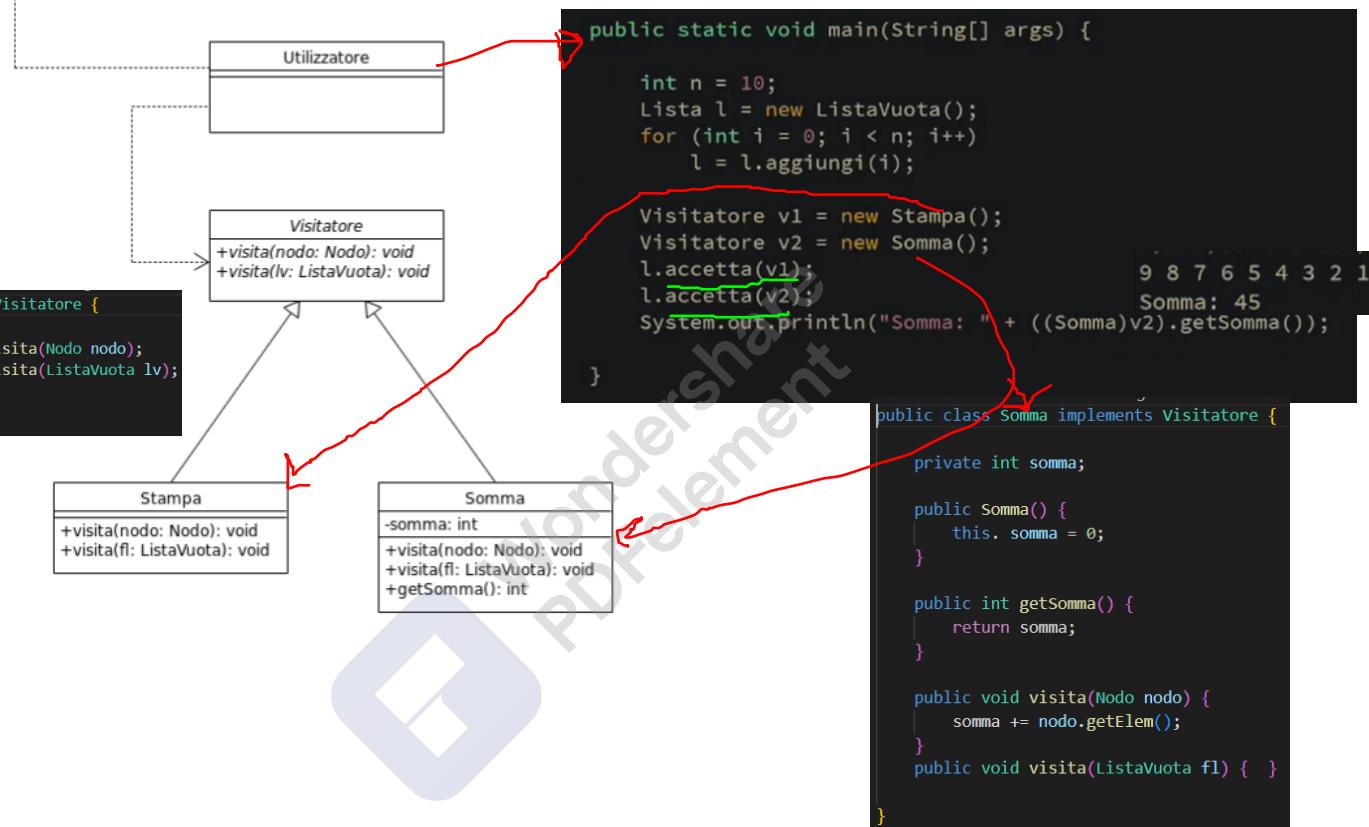
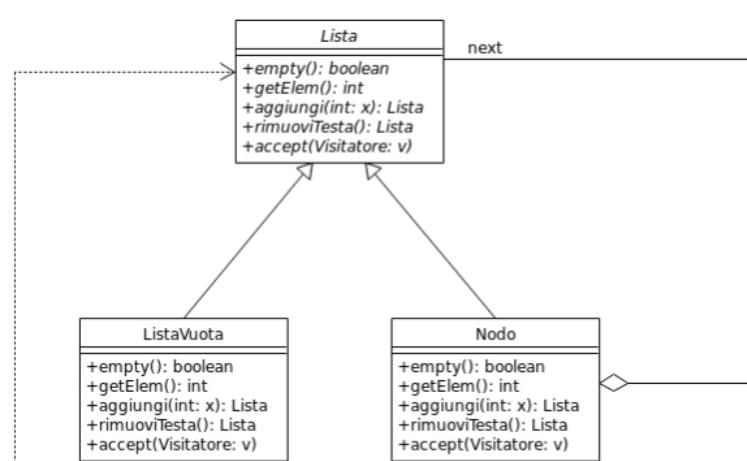
```

public class WatcherVisitor implements Visitor {
    public void visit(Collection collection) {
        Iterator iterator = collection.iterator();
        while (iterator.hasNext()) {
            Object o = iterator.next();
            if (o instanceof Visitable)
                ((Visitable)o).accept(this);
            else if (o instanceof Collection )
                visit( (Collection) o );
        }
    }

    public void visit(VisitableString vString) {
        System.out.println( "'" +vString.getString() +"' " );
    }

    public void visit(VisitableFloat vFloat) {
        System.out.println( vFloat.getFloat().toString()+"f" );
    }
}

```



```

public abstract class Lista {
    public abstract boolean empty();
    public abstract int getElem();
    public abstract Lista aggiungi(int x);
    public abstract Lista rimuoviTesta();
    public abstract void accetta(Visitatore visitatore);
}
  
```

```

public class ListaVuota extends Lista {
    public ListaVuota() { }

    public boolean empty() {
        return true;
    }

    public int getElem() {
        assert false; return 0;
    }

    public Lista aggiungi(int x) {
        return new Nodo(x, this);
    }

    public Lista rimuoviTesta() {
        assert false; return null;
    }

    public void accetta(Visitatore visitatore) {
        visitatore.visit(this);
    }
}
  
```

```

public class Nodo extends Lista {
    private int elem;
    private Lista next;

    public Nodo(int elem, Lista next) {
        this.elem = elem;
        this.next = next;
    }

    public boolean empty() {
        return false;
    }

    public int getElem() {
        return elem;
    }

    public Lista aggiungi(int x) {
        return new Nodo(x, this);
    }

    public Lista rimuoviTesta() {
        return next;
    }

    public void accetta(Visitatore visitatore) {
        visitatore.visit(this);
        next.accetta(visitatore);
    }
}
  
```

codice che percorre la struttura dati sarà utilizzato sia per stampare che per sommare

APPROFONDIMENTO PATTERN GoF vedere slide

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

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.

siamo ancora nella
prima iterazione!
3-4 settimane

DAL PROGETTO AL CODICE

L'**implementazione** è un processo di traduzione relativamente meccanico. Tuttavia 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.

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**. Se un oggetto implementa un'interfaccia, si dichiari la variabile in termini dell'interfaccia, non della classe concreta.

Un **metodo di una classe** può essere scritto per ispezione del **diagramma di collaborazione**.

Il **costruttore di una classe** è generato, per ispezione, da una versione parziale del diagramma di interazione della chiamata iniziale della classe.

Le **classi** possono essere **implementate** in modo e in ordine diverso, per esempio dalla meno accoppiata alla più accoppiata. va scelto un ordine

Extreme Programming (XP), test e refactoring

XP ha promosso:

- 1) **la pratica dei test**: scrivere i **test** per primi;
- 2) **il refactoring continuo del codice per migliorare la qualità**: meno duplicazioni, maggiore chiarezza, ecc.

Test Driven Development legati allo sviluppo incrementale, devono essere fatto ad ogni patch

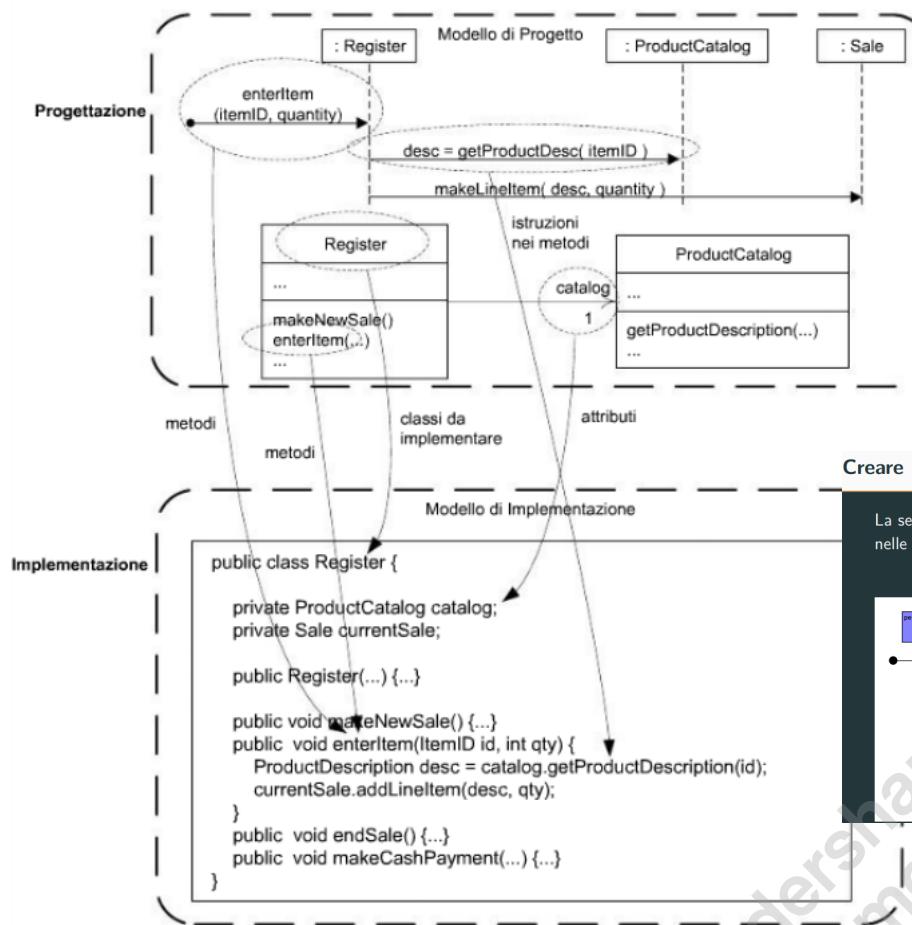
Una **pratica promossa** dal **metodo iterativo e agile XP** (applicabile a UP) è lo **sviluppo guidato dai test**, noto come **sviluppo preceduto dai test**.

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

Vantaggi di TDD

- i **test unitari** (ovvero i test **relativi a singole classi e metodi**) **vengono effettivamente scritti**; e così li hai disponibili
- 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**.

Alcune relazioni tra gli elaborati di UP

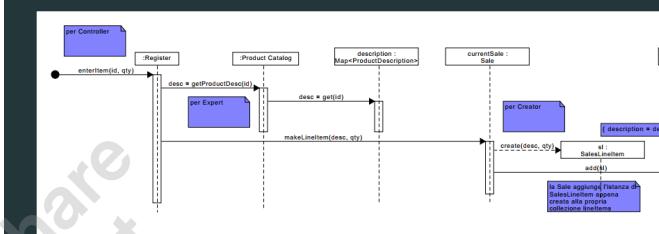


traduzione di classi e firme metodi e attributi

sequenza di messaggi
guardiamo gli ssd

Creare metodi dai diagrammi di interazione

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



Classe Store 7
public class Store {

```
private ProductCatalog catalog;
private Register register;
private List<Sale> completedSales;

public Store() {
    catalog = new ProductCatalog();
    register = new Register(this, catalog);
    completedSales = new ArrayList<>();
}

public Register getRegister() { return register; }

public void addSale(Sale s) {
    completedSales.add(s);
}
```

Classe ProductDescription 1
public class ProductDescription {

```
private ItemID id;
private Money price;
private String description;

public ProductDescription ( ItemID id, Money price, String desc ) {
    this.id = id;
    this.price = price;
    this.description = desc;
}

public ItemID getItemID() { return id; }
public Money getPrice() { return price; }
public String getDescription() { return description; }
```

Classe ProductCatalog 2
public class ProductCatalog {

```
private Map<ItemID, ProductDescription> descriptions;

public ProductCatalog() {
    descriptions = new HashMap<>();
    // carica dati di esempio
    loadProductDescriptions();
}

public ProductDescription getProductDescription(ItemID id) {
    return descriptions.get(id);
}

/* carica dati di prova */
private void loadProductDescriptions() {
    ItemID id;
    Money price;
    ProductDescription pd;

    id = new ItemID("100");
    price = new Money(3);
    pd = new ProductDescription(id, price, "pr. #100");
    descriptions.put(id, pd);

    id = new ItemID("200");
    price = new Money(4);
    pd = new ProductDescription(id, price, "pr. #200");
    descriptions.put(id, pd);
}
```

Classe Sale 4
public class Sale {

```
private List<SalesLineItem> lineItems;
private Date date;
private CashPayment payment;

public Sale() {
    lineItems = new ArrayList<>();
    date = new Date();
    payment = null;
}

public void makeLineItem(ProductDescription desc, int qty) {
    lineItems.add( new SalesLineItem(desc, qty) );
}
```

Classe SalesLineItem 5
public class SalesLineItem {

```
private int quantity;
private ProductDescription description;

public SalesLineItem(ProductDescription desc, int qty) {
    this.description = desc;
    this.quantity = qty;
}

public Money getSubtotal() {
    return description.getPrice().times(quantity);
}
```

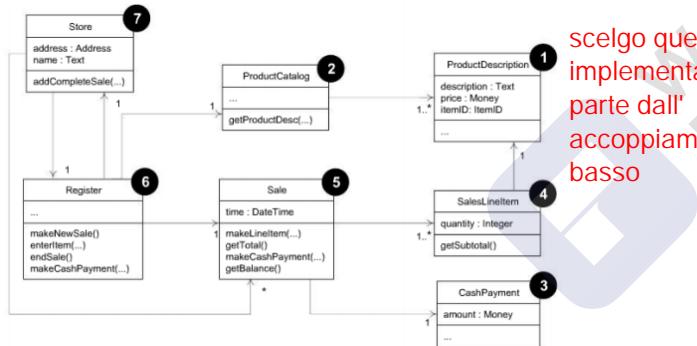
Classe CashPayment K
public class CashPayment {

```
private Money amount;

public CashPayment(Money cashTendered) {
    this.amount = cashTendered;
}

public Money getAmount() { return amount; }
```

scelgo quest'ordine di implementazione che parte dall'accoppiamento più basso



Classe ProductDescription 1

```
public class ProductDescription {
```

```
private ItemID id;
private Money price;
private String description;
```

```
public ProductDescription ( ItemID id, Money price, String desc ) {
    this.id = id;
    this.price = price;
    this.description = desc;
}
```

```
public ItemID getItemID() { return id; }
public Money getPrice() { return price; }
public String getDescription() { return description; }
```

Classe ProductCatalog 2

```
public class ProductCatalog {
```

```
private Map<ItemID, ProductDescription> descriptions;
```

```
public ProductCatalog() {
    descriptions = new HashMap<>();
    // carica dati di esempio
    loadProductDescriptions();
}
```

```
public ProductDescription getProductDescription(ItemID id) {
    return descriptions.get(id);
}
```

```
/* carica dati di prova */
private void loadProductDescriptions() {
    ItemID id;
    Money price;
    ProductDescription pd;
```

```
id = new ItemID("100");
price = new Money(3);
pd = new ProductDescription(id, price, "pr. #100");
descriptions.put(id, pd);

id = new ItemID("200");
price = new Money(4);
pd = new ProductDescription(id, price, "pr. #200");
descriptions.put(id, pd);
```

Classe Register 3

```
public class Register {
```

```
private Store store;
private ProductCatalog catalog;
private Sale currentSale;
```

```
public Register(Store store, ProductCatalog catalog) {
    this.store = store;
    this.catalog = catalog;
    this.currentSale = null;
}
```

```
public void makeNewSale() {
    currentSale = new Sale();
}
```

```
public void enterItem(ItemID id, int quantity) {
    ProductDescription desc =
        catalog.getProductDescription(id);
    currentSale.makeLineItem(desc, quantity);
}
```

```
public void endSale() {
    // niente da fare
}
```

```
public void makeCashPayment(Money cashTendered) {
    currentSale.makeCashPayment(cashTendered);
    store.addSale(currentSale);
}
```

```
}
```

Classe Sale 4

```
public class Sale {
```

```
private List<SalesLineItem> lineItems;
private Date date;
private CashPayment payment;
```

```
public Sale() {
    lineItems = new ArrayList<>();
    date = new Date();
    payment = null;
}
```

```
public void makeLineItem(ProductDescription desc, int qty) {
    lineItems.add( new SalesLineItem(desc, qty) );
}
```

K

Classe SalesLineItem

```
public class SalesLineItem {
```

```
private int quantity;
private ProductDescription description;
```

```
public SalesLineItem(ProductDescription desc, int qty) {
    this.description = desc;
    this.quantity = qty;
}
```

```
public Money getSubtotal() {
    return description.getPrice().times(quantity);
}
```

Classe CashPayment

```
public class CashPayment {
```

```
private Money amount;
```

```
public CashPayment(Money cashTendered) {
    this.amount = cashTendered;
}
```

```
public Money getAmount() { return amount; }
```

Il TDD prevede l'utilizzo di diversi tipi di test:

- 1) **test unitari**: hanno lo scopo di verificare il funzionamento delle piccole parti (unità) del sistema ma non di verificare il sistema nel suo complesso;
- 2) **test di integrazione**: per verificare la comunicazione tra specifiche parti (elementi strutturali) del sistema;
- 3) **test end-to-end**: per verificare il collegamento complessivo tra tutti gli elementi del sistema;
- 4) **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:

- 1) **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;
- 2) **esecuzione**: fa fare qualcosa alla fixture (per esempio, eseguire delle operazioni), viene richiesto lo specifico comportamento da verificare;
- 3) **verifica**: valuta che i risultati ottenuti corrispondano a quelli previsti;
- 4) **rilascio**: opzionalmente rilascia o ripulisce gli oggetti e/o le risorse utilizzate nel test (per evitare che altri test vengano corrotti).

Refactoring esiste per gli sviluppi di tipo incrementale, perché lo sviluppo degrada il codice

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.

La sua essenza è applicare piccole trasformazioni che preservano il comportamento. Dopo ogni trasformazione, i test unitari vengono eseguiti nuovamente per dimostrare che refactoring non abbia provocato una regressione (un fallimento).

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

Regole per il TDD e refactoring:

- scrivi un test unitario che fallisce, per dimostrare la mancanza di una funzionalità o di codice;
- scrivi il codice più semplice possibile per far passare il test;
- riscrivi o ristruttura (refactor) il codice, migliorandolo, oppure passa a scrivere il prossimo test unitario.

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.

Esempio di classe di test

```
@Test
// test per il metodo Sale.makeLineItem
public void testMakeLineItem() {
    // PREPARAZIONE
    // - crea la fixture, ovvero l'oggetto da testare
    // - un possibile idioma è chiamarlo 'fixture'
    // - spesso è definito come una variabile d'istanza
    // anziché come una variabile locale
    Sale sale = new Sale();
    // - crea degli oggetti di supporto per il test
    ProductDescription tofu =
        new ProductDescription( new ItemID( "1" ),
                               new Money( 2.50 ),
                               "Tofu" );
    ProductDescription burger =
        new ProductDescription( new ItemID( "2" ),
                               new Money( 1.50 ),
                               "VeggieBurger" );

    // ESECUZIONE
    // questo codice viene scritto **immaginando** che
    // ci sia già un metodo makeLineItem.
    // Questo atto di immaginazione mentre viene scritto
    // il test tende a migliorare o a chiarire la nostra
    // comprensione dell'interfaccia dettagliata dell'oggetto.
    // Pertanto il TDD ha il vantaggio collaterale di
    // chiarire la progettazione a oggetti dettagliata.
    sale.makeLineItem( tofu, 2 );
    sale.makeLineItem( burger, 1 );

    // VERIFICA
    // confronta il risultato atteso con quello effettivo
    assertEquals( new Money(6.50), sale.getTotal() );
}
```

Alcuni tipi di refactoring:

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

o introdurre dei pattern