

Sviluppo Applicazioni Software

Appunti

Luca Barra

Anno accademico 2023/2024



CAPITOLO 1	PREMESSA	PAGINA 1
1.1	Licenza	1
1.2	Formato utilizzato	1
CAPITOLO 2	PROCESSI PER LO SVILUPPO SOFTWARE	PAGINA 4
2.1	Introduzione Specifica dei requisiti — 5 • Sviluppo del software — 5 • Convalida del software — 5 • Evoluzione del software — 6	4
2.2	Modelli di processo software Modello a cascata — 7 • Modello incrementale — 7 • Integrazione e configurazione — 8 • Sviluppo incrementale, iterativo ed evolutivo — 8	6
2.3	Sviluppo agile I principi dello sviluppo agile — 9 • eXtreme Programming (XP) — 10 • Scrum — 10	9
CAPITOLO 3	UNIFIED PROCESS (UP)	PAGINA 11
3.1	OOA e OOD UML — 13	11
3.2	Unified Process Le discipline — 14 • Che cosa sono i requisiti? — 15	13
3.3	Ideazione Artefatti nell’Ideazione — 17 • Tipologie di documenti — 17	16
3.4	Casi d’Uso Disciplina dei requisiti — 18 • Capire il contesto del sistema e catturare i requisiti — 19 • Casi d’Uso e UP — 19 • Attori e tipi di attori — 20 • Formato di un Caso d’Uso — 21 • Come scrivere un Caso d’Uso — 21 • Trovare i Casi d’Uso — 24 • Livello dei Casi d’Uso — 26	18
3.5	Elaborazione Pianificazione dell’iterazione successiva — 27 • Artefatti dell’Elaborazione — 27	26
3.6	Modello di Dominio Classi concettuali — 28 • Trovare le classi concettuali — 29 • Associazioni — 30 • Composizione — 32 • Attributi — 32 • Verifiche del modello — 32	28
3.7	Diagrammi di sequenza del sistema (SSD) Notazione — 35	34
3.8	Contratti Pre-condizioni e Post-condizioni — 37 • Scrivere contratti — 38	37
CAPITOLO 4	ARCHITETTURA DEL SOFTWARE	PAGINA 39
4.1	Architettura logica	39
4.2	Strato del dominio	41
4.3	Separazione Modello-Vista	43

CAPITOLO 5	DIAGRAMMI DI INTERAZIONE E DI CLASSE UML	PAGINA 44
5.1	Verso la progettazione a oggetti Modellazione dinamica e statica — 44	44
5.2	Diagrammi di interazione Diagrammi di sequenza del design (DSD) — 46	45
5.3	Diagrammi delle classi Diagrammi di classe del design (DCD) — 53	53
CAPITOLO 6	PATTERN GRASP	PAGINA 57
6.1	Responsability-Driven Development Tipi di responsabilità — 57	57
6.2	General Responsibility Assignment Software Patterns	58
6.3	I pattern GRASP Creator — 59 • Information Expert — 61 • Low Coupling — 62 • High Cohesion — 63 • Controller — 64	59
CAPITOLO 7	PATTERN GoF	PAGINA 67
7.1	Design Pattern GoF	67
7.2	Pattern Creazionali Abstract Factory — 68 • Singleton — 70	68
7.3	Pattern Strutturali Adapter — 71 • Composite — 73 • Decorator — 75	71
7.4	Pattern Comportamentali Observer — 77 • State — 79 • Strategy — 80 • Visitor — 82	77
CAPITOLO 8	DAL PROGETTO AL CODICE	PAGINA 84
8.1	Trasformare i progetti in codice Collezioni e costruttori — 85 • Ordine di implementazione — 86	84
8.2	Sviluppo test-driven e refactoring I test — 87 • Il refactoring — 87	87

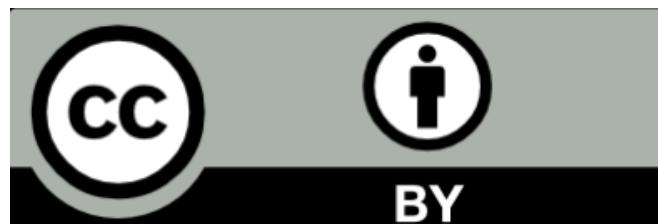
1

Premessa

1.1 Licenza

Questi appunti sono rilasciati sotto licenza Creative Commons Attribuzione 4.0 Internazionale (per maggiori informazioni consultare il link: <https://creativecommons.org/version4/>). Sono basati sulle slides del corso di "Sviluppo Applicazioni Software" del prof. Matteo Baldoni, contenenti riferimenti a:

1. C. Larman, "Applicare UML e i Pattern", Pearson, 2016;
2. F. Guidi Polanco, "GoF's Design Patterns in Java", Politecnico di Torino, 2002 (disponibile all'indirizzo <http://eii.pucv.cl/pers/guidi/designpatterns.htm>).



1.2 Formato utilizzato

In questi appunti vengono utilizzati molti *box*. Questa è una semplice rassegna che ne spiega l'utilizzo:

Box di "Concetto sbagliato":

Concetto sbagliato 1.1: Testo del concetto sbagliato

Testo contente il concetto giusto.

Box di "Corollario":

Corollario 1.2.1 Nome del corollario

Testo del corollario. Per corollario si intende una definizione minore, legata a un'altra definizione.

Box di "Definizione":

Definizione 1.2.1: Nome delle definizione

Testo della definizione.

Box di "Domanda":

Domanda 1.1

Testo della domanda. Le domande sono spesso utilizzate per far riflettere sulle definizioni o sui concetti.

Box di "Esempio":

Esempio 1.2.1 (Nome dell'esempio)

Testo dell'esempio. Gli esempi sono tratti dalle slides del corso.

Box di "Note":

Note:-

Testo della nota. Le note sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive.

Box di "Osservazioni":

Osservazioni 1.2.1

Testo delle osservazioni. Le osservazioni sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive. A differenza delle note le osservazioni sono più specifiche.

Box di "Pattern":

Pattern 1.2.1 (*Nome del pattern*):

Problema: Testo del problema.

Soluzione: Testo della soluzione.

Box di "Struttura del Pattern":

Struttura del Pattern

Questo box è utilizzato per mostrare la struttura di un pattern GoF.

2

Processi per lo sviluppo software

2.1 Introduzione

In questa sezione verranno mostrati, anche in chiave storica, i principali processi per lo *sviluppo software*, *modelli di processi software*, sviluppo *iterativo* ed evolutivo, sviluppo *agile*.

Definizione 2.1.1: Software di qualità

- Non è un semplice programma o gruppo di programmi;
- Include *documentazione*, *test*, *manutenzione*, *aggiornamenti*;

Corollario 2.1.1 Caratteristiche essenziali

- ⇒ *Mantenibilità*: il software deve evolversi in base alle necessità dei clienti^a;
- ⇒ *Fidatezza*: il software non dovrebbe causare danni fisici o economici;
- ⇒ *Efficienza*: il software deve fare un uso efficiente delle risorse;
- ⇒ *Accettabilità*: il software deve essere comprensibile, usabile e compatibile con altri sistemi.

^aDa questo si hanno i maggiori introiti.

Note:-

A volte può convenire vendere il software "sottoprezzo" per poi guadagnare con la manutenzione.

Domanda 2.1

Cosa descrive un processo software?

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

Definizione 2.1.2: Un processo per lo sviluppo software

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

Si possono distinguere quattro attività di processo comuni:

- ⇒ *Specifiche del software*: clienti e sviluppatori definiscono le funzionalità del software (e i relativi vincoli);
- ⇒ *Sviluppo del software*: il software viene progettato e sviluppato;
- ⇒ *Convalida del software*: il software viene convalidato per garantire che soddisfi le specifiche del cliente;
- ⇒ *Evoluzione del software*: il software viene modificato per riflettere i cambiamenti nei requisiti del cliente e del mercato.

2.1.1 Specifica dei requisiti

Anche detta "*ingegneria dei requisiti*", è l'attività per capire e definire quali sono i requisiti richiesti dal sistema e identificare i vincoli all'operabilità e allo sviluppo del sistema.

Le fasi principali di questa attività sono:

- ⇒ *Deduzione e analisi dei requisiti*: osservazione di sistemi esistenti, discussioni con possibili utenti, analisi, etc.
- ⇒ *Specificazione dei requisiti*: si traducono le informazioni raccolte in un *documento*;
- ⇒ *Convalida dei requisiti*: si controlla che i requisiti siano realistici, coerenti e completi.

2.1.2 Sviluppo del software

Anche detta "*progettazione e implementazione del software*", è l'attività di conversione delle specifiche del software in un sistema da consegnare al cliente. Nelle *metodologie agili* la progettazione e l'implementazione sono spesso *integrate* e, tipicamente, non producono documenti formali.

Le fasi principali di questa attività sono:

- ⇒ *Progettazione dell'architettura*: identifica la struttura complessiva del sistema, dei componenti, delle loro relazioni e della loro distribuzione;
- ⇒ *Progettazione del database*: si progetta la rappresentazione delle strutture dati che verranno utilizzate e la loro rappresentazione in un database¹;
- ⇒ *Progettazione dell'interfaccia*: definisce l'interfaccia utente e le modalità di interazione con il sistema²;
- ⇒ *Progettazione e scelta dei componenti*: si ricercano i componenti riutilizzabili o vengono progettati nuovi componenti.

Note:-

La scelta dei componenti è particolarmente facile nel caso di linguaggi object-oriented.

2.1.3 Convalida del software

L'attività di verifica e convalida serve a dimostrare che un sistema sia *conforme* alle specifiche e che *soddisfi* le esigenze del cliente. La convalida richiede anche attività di *controllo*, *ispezione* e *revisione* a ogni stadio del processo di sviluppo. In alcune metodologie agili si scrivono i test prima di scrivere il codice (eXtreme Programming).

Note:-

In questo corso ci si concentrerà sul processo di testing. Per una modo formale di verificare la correttezza di un sistema si può fare riferimento al corso "Metodi formali dell'informatica".

¹Non verrà trattata in questo corso. È stata parzialmente trattata nel corso "Basi di dati".

²Non verrà trattato lo sviluppo di un'interfaccia. È stato parzialmente trattato nel corso "Programmazione III".

I test possono essere:

- ⇒ *Test di unità (o dei componenti)*: i componenti vengono testato singolarmente³;
- ⇒ *Test del sistema*: si testa il sistema nel suo complesso;
- ⇒ *Test del cliente*: il sistema viene testato dal cliente con i propri dati.

2.1.4 Evoluzione del software

Anche detto "*manutenzione del software*", è l'attività di modifica durante o dopo lo sviluppo di un sistema software. La distinzione (storica) tra sviluppo e manutenzione è sempre più irrilevante. L'ingegneria del software è un unico processo evolutivo.

Note:-

Può capitare che si debba far fronte a cambiamenti improvvisi per esigenze di mercato o per incomprensioni con il cliente.

Bisogna *ridurre* i costi di rilavorazione:

- ⇒ *Anticipazione dei cambiamenti*: si possono prevedere o anticipare eventuali cambiamenti prima di una richiesta di rilavorazione;
- ⇒ *Tolleranza ai cambiamenti*: si progetta il sistema in modo da rendere facili eventuali cambiamenti.

Ci sono due metodi per far fronte ai cambiamenti:

- ⇒ *Prototipazione del sistema*: il sistema viene sviluppato rapidamente per verificare i requisiti del cliente. Ciò consente eventuali modifiche prima di sviluppare il sistema completo;
- ⇒ *Consegna incrementale*: vengono consegnati al cliente parti del sistema in modo incrementale in modo che il cliente possa provarlo e commentarlo.

Note:-

Il refactoring è un importante meccanismo per supportare la tolleranza ai cambiamenti

2.2 Modelli di processo software

Esistono veri modelli di processo software: *cascata*, *Unified Process*, *Scrum*, *XP*, *RUP*, *RAD*, *Spirale*, etc. Le quattro attività fondamentali sono organizzate in modo diverso in ciascun modello: in sequenza nel modello a cascata e intrecciate negli altri (modelli incrementali). Un ulteriore modello è il modello a integrazione e configurazione che però è poco trattato a livello ingegneristico.

Definizione 2.2.1: Paradigma di processo

Il modello di processo software è una rappresentazione semplificata di un processo software. Sono strutture di processo da *estendere* e *adattare* per soddisfare le esigenze specifiche di un progetto.

Osservazioni 2.2.1

Non esiste un modello di processo software "universale", ma la scelta del modello dipende dai requisiti del cliente:

- ⇒ i software a sicurezza critica richiedono un modello a cascata per via delle analisi e della documentazione;
- ⇒ i software per il mercato richiedono un modello incrementale;
- ⇒ i sistemi aziendali richiedono un modello a configurazione e integrazione.

³Visti nel corso "Algoritmi e strutture dati".

Inoltre, in grandi sistemi, si possono combinare più modelli.

2.2.1 Modello a cascata

Definizione 2.2.2: Modello a cascata

Il modello a cascata è un modello di processo software in cui le fasi di sviluppo sono viste come *fasi distinte* e *non sovrapposte*.

Questo modello era l'unico modello utilizzato fino agli anni '80.

Note:-

Si contrappone ai modelli incrementali in cui le fasi di sviluppo sono sovrapposte e iterate.

Corollario 2.2.1 Fasi del modello a cascata

- ⇒ All'inizio si definiscono i requisiti;
- ⇒ All'inizio si definisce un piano temporale;
- ⇒ Si progetta e modella il sistema;
- ⇒ Si crea un progetto completo del software;
- ⇒ Si inizia la programmazione del sistema;
- ⇒ Si testa il sistema, si rilascia e si prosegue con la manutenzione.

Il modello a cascata:

- ⇒ Non è adatto allo sviluppo in team;
- ⇒ Si dovevano definire spesso modelli matematici;
- ⇒ Costava molto in termini di tempo e denaro.

2.2.2 Modello incrementale

Definizione 2.2.3: Modello incrementale

Il modello incrementale è un modello di processo software in cui il sistema viene sviluppato in *incrementi* (o *iterazioni*). Si effettuano *feedback veloci* e *rilasci*.

Note:-

Negli anni '80 e '90 molte persone si avvicinano al mondo della progettazione e nasce la necessità di sviluppare software in modo incrementale.

Corollario 2.2.2 I casi d'uso

I casi d'uso sono il modo migliore per definire i requisiti: il cliente racconta una storia e il programmatore la traduce in un caso d'uso.

Lo sviluppo incrementale:

- ⇒ È un approccio *plan-driven*, *agile* o una combinazione di questi approcci;
- ⇒ Se *plan-driven*, si pianificano in anticipo gli incrementi;

- ⇒ Se *agile*, si identificano gli incrementi iniziali ma si dà priorità al rilascio di incrementi che soddisfano i requisiti più importanti;
- ⇒ Il costo di implementazione di modifiche è ridotto;
- ⇒ È più facile ottenere un feedback dal cliente;

• **Note:-**

Tuttavia si devono avere consegne regolari e frequenti, la struttura dei sistemi tende a degradarsi e richiede pianificazione in anticipo per grandi team.

2.2.3 Integrazione e configurazione

Definizione 2.2.4: Riutilizzo del software

- ⇒ Dagli anni 2000 si sono diffusi software che riutilizzano software già esistente;
- ⇒ Collezioni di oggetti che sono sviluppati come un componente o un pacchetto da integrare tramite framework;
- ⇒ Servizi web che possono essere integrati in un sistema.

Le fasi principali sono:

- ⇒ *Specifica dei requisiti*;
- ⇒ *Ricerca e valutazione del software*: se esiste un software che soddisfa i requisiti;
- ⇒ *Perfezionamento dei requisiti*: utilizzando le informazioni trovate nella ricerca;
- ⇒ *Configurazione del sistema di applicazioni*;
- ⇒ *Adattamento e integrazione*: si integra il sistema con i componenti riutilizzabili.

Osservazioni 2.2.2

Questo approccio riduce la quantità di software da sviluppare, riducendo i costi e i rischi. Però bisogna scendere a compromessi con i requisiti e si perde il controllo sull'evoluzione del sistema.

2.2.4 Sviluppo incrementale, iterativo ed evolutivo

Questo modello è:

- **Incrementale**: si incrementa il codice man mano che si sviluppa;
- **Iterativo**: si sviluppa il software in cicli (iterazioni);
- **Evolutivo**: si sviluppa il software in modo che possa evolvere a ogni iterazione richiedendo un feedback.

Definizione 2.2.5: Approccio iterativo

Nell'approccio iterativo:

- ⇒ lo sviluppo è organizzato in mini-progetti brevi (le iterazioni);
- ⇒ il risultato di ogni iterazione è un sistema parzialmente funzionante (testato e integrato);
- ⇒ ogni iterazione dura poche settimane^a e comprende le proprie attività di analisi, sviluppo, etc.;
- ⇒ si ottiene un feedback a ogni iterazione.

^aUn'iterazione di lunghezza fissata è detta *timeboxed*.

Note:-

Git supporta lo sviluppo incrementale, iterativo ed evolutivo.

2.3 Sviluppo agile

Definizione 2.3.1: Sviluppo agile

Lo sviluppo *agile* è un insieme di metodi di sviluppo software.

Contesto:

- ⇒ Il software è parte essenziale delle operazioni aziendali;
- ⇒ La rapidità della consegna è un fattore critico;
- ⇒ Spesso non si possono ottenere requisiti stabili;
- ⇒ I requisiti diventano chiari solo dopo che il sistema è stato consegnato e utilizzato;
- ⇒ In successive iterazioni si possono ottenere requisiti più chiari.

2.3.1 I principi dello sviluppo agile

Definizione 2.3.2: Agile Modelling

Lo scopo della modellazione (UML) è principalmente quello di *comprendere* e di agevolare la *comunicazione*, non di documentare.

- ⇒ Adottare un metodo agile non significa evitare del tutto la modellazione;
- ⇒ Non si deve applicare UML per eseguire per intero o per la maggior parte la progettazione software;
- ⇒ Va utilizzato l'approccio più semplice e che comporta il minor dispendio di energie. Esempio: abbozzo di UML su una lavagna;
- ⇒ La modellazione non va fatta da soli ma in coppie o in gruppo;
- ⇒ Solo il codice verificato dimostra il vero progetto, i diagrammi precedenti sono suggerimenti incompleti (usa e getta);
- ⇒ La modellazione per OO dovrebbe essere eseguita dagli stessi sviluppatori che andranno effettivamente a scrivere il codice.

Pratiche innovative:

- ⇒ *Storie utente*: scenari d'uso in cui potrebbe trovarsi un utente. Il cliente lavora a stretto contatto con il team di sviluppo e discute di possibili scenari;
- ⇒ *Refactoring*: il codice va costantemente rifattorizzato per proteggerlo dal deterioramento causato dallo sviluppo incrementale;
- ⇒ *Sviluppo con test iniziali*: lo sviluppo non può procedere finché tutti i test non sono stati superati;
- ⇒ *Programmazione a coppie*: i programmatore lavorano a coppie nella stessa postazione per sviluppare il software.

2.3.2 eXtreme Programming (XP)

Definizione 2.3.3: eXtreme Programming

eXtreme Programming (XP) è un metodo di sviluppo software che si basa su valori e principi di base:

- ⇒ sviluppo incrementale attraverso piccole e frequenti release;
- ⇒ il cliente è parte attiva dello sviluppo;
- ⇒ il progetto è supportato da test, refactoring e integrazione continua;
- ⇒ si punta a mantenere la semplicità.

2.3.3 Scrum

Definizione 2.3.4: Scrum

Scrum offre un framework per organizzare progetti agili e fornire una visibilità esterna su ciò che sta accadendo, ossia si occupa dell'organizzazione del lavoro e della gestione dei progetti.

Scrum è un approccio iterativo e incrementale in cui ciascuna iterazione ha una durata fissata denominata Sprint (non si hanno estensioni).

Sono presenti tre ruoli:

- ⇒ *Product Owner*: rappresenta il cliente, definisce i requisiti e specifica le priorità attraverso il *Product Backlog*⁴;
- ⇒ *Development Team*: le persone che sviluppano il software;
- ⇒ *Scrum Master*: garantisce che il team segua le regole di Scrum.

Gestione agile della progettazione:

- ⇒ Il Development Team seleziona dal Product Backlog un insieme di voci da sviluppare durante quell'iterazione (*Sprint Goal*), compila lo *Sprint Backlog* (ossia i compiti dettagliati per raggiungere il goal);
- ⇒ Il risultato di ciascuno Sprint è un prodotto software funzionante chiamato "incremento di prodotto potenzialmente rilasciabile" (integrato, verificato e documentato);
- ⇒ Nello *Sprint Review* il Product Owner e il Development Team presentano le parti coinvolte dall'incremento, ne fanno la dimostrazione, ottengono un feedback e decidono cosa fare nello Sprint successivo;
- ⇒ Si dà enfasi all'adozione di Team auto-organizzati e auto-gestiti.

⁴Un elenco di voci, funzionalità e requisiti.

3

Unified Process (UP)

3.1 OOA e OOD

Definizione 3.1.1: OOA/D

- ⇒ **OOA** (Object Oriented Analysis): studio dei requisiti e delle specifiche del sistema;
- ⇒ **OOD** (Object Oriented Design): progettazione del sistema.

Per studiare OOA/D si utilizza Unified Process, un processo di sviluppo software orientato agli oggetti.

Note:-

UP può essere applicato usando un approccio agile come Scrum o XP.

Corollario 3.1.1 UML

UP utilizza UML come linguaggio di modellazione. UML è un linguaggio di modellazione grafico e testuale per la specifica, la costruzione e la documentazione di sistemi software orientati agli oggetti.

Concetto sbagliato 3.1: UML descrive il software

UML non è nato per descrivere software, ma per descrivere *concetti*^a.

^aSimile a ER, visto nel corso "Basi di dati".

OOD è guidata dalle responsabilità (si vedano i pattern GRASP):

- ⇒ Quali sono gli oggetti? Quali sono le classi?
- ⇒ Cosa deve conoscere un oggetto? Cosa deve saper fare?
- ⇒ Come collaborano gli oggetti?

Definizione 3.1.2: Pattern

I pattern sono euristiche, best practice, che aiutano a codificare principi di soluzioni.

ODD è correlata all'analisi dei requisiti:

- ⇒ *Casi d'uso*;
- ⇒ *Storie utente*.

Esempio 3.1.1 (Gioca una *partita a dadi*)

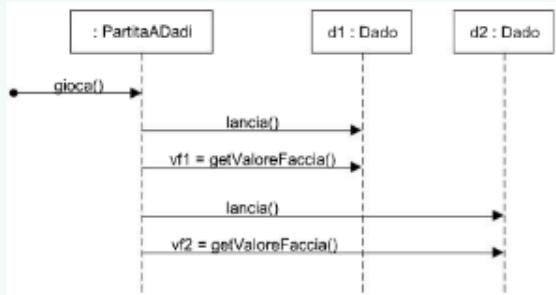
Definizione dei casi d'uso: storie scritte.

Il *Giocatore* chiede di *lanciare* i *dadi*. Il Sistema presenta il *risultato*: se *il valore totale* delle facce dei dadi è sette, il giocatore ha vinto; altrimenti ha perso.

Definizione di un modello di dominio: *i concetti o gli oggetti significativi*.



Assegnare responsabilità agli oggetti e disegnare diagrammi di interazione: *responsabilità e collaborazioni*.



Definizione dei diagrammi delle classi di progetto.



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

- ⇒ Sviluppo iterativo;
- ⇒ Approccio agile;
- ⇒ Unified Process (UP).

Note:-

ER e UML non sono pienamente adatti a possibili incrementi.

3.1.1 UML

Definizione 3.1.3: UML

UML è un linguaggio *visuale* per la specifica, la costruzione e la documentazione degli elaborati di un sistema software.

UML è uno standard per la notazione di diagrammi per disegnare o rappresentare figure relative al software (specialmente OO).

UML è un *abbozzo* o un *progetto* per aiutare la comprensione nei team di sviluppo. Il termine abbozzo indica che può essere soggetto a correzione, ma se non ci sono feedback a tal proposito deve essere trattato come un dizionario.

Uso di UML:

- ⇒ Punto di vista *concettuale*: modello di dominio, per visualizzare concetti del mondo reale;
- ⇒ Punto di vista *software*: diagramma delle classi di progetto, utilizzata per visualizzare elementi software.

Brevi note storiche:

- ⇒ Anni '60 e '70: nascita dei linguaggi OO (Simula e Smalltalk);
- ⇒ 1988: Bertrand Meyer, "Object-Oriented Software";
- ⇒ 1991: Jim Rumbaugh, "Object-Oriented Modelling and Design" (OOA/D);
- ⇒ 1991, Grady Booch, "Object-Oriented Software Engineering" (OOA/D e Casi d'Uso);
- ⇒ 1994, Rumbaugh e Booch fanno le prime proposte di UML;
- ⇒ Rational Corporation fondata dai "tre amigos" (Jacobson, Booch e Rumbaugh);
- ⇒ 1997 UML 1;
- ⇒ 2004 UML 2 (usato attualmente).

3.2 Unified Process

Definizione 3.2.1: Unified Process

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*.

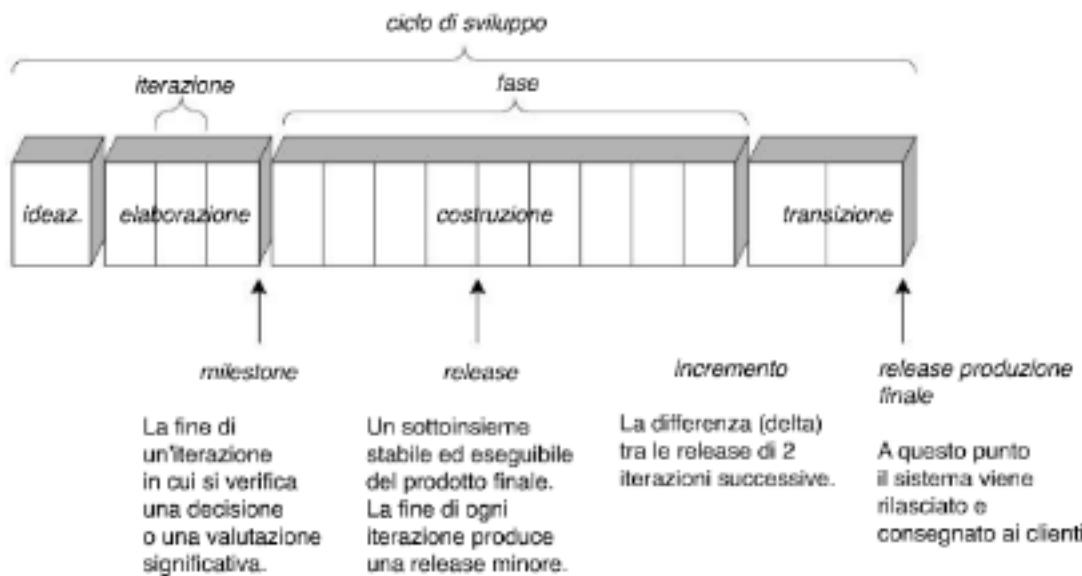
Domanda 3.1

Cosa c'è in UP?

- ⇒ Un'organizzazione del piano di progetto per fasi sequenziali;
- ⇒ Indicazioni sulle attività da svolgere nell'ambito di discipline e sulle loro inter-relazioni;
- ⇒ Un insieme di ruoli predefiniti;
- ⇒ Un insieme di artefatti da produrre.

Un progetto UP è organizzato in 4 fasi:

- ⇒ *Ideazione* (inception): visione approssimativa, studio economico, portata, stime approssimative di costi e tempi. Milestone: *Obiettivi*;
- ⇒ *Elaborazione* (elaboration): visione raffinata, è un'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. Milestone: *Architetture*;
- ⇒ *Costruzione* (construction): implementazione iterativa degli elementi rimanenti, più facili e a rischio minore, preparazione al rilascio. Milestone: *Capacità operazionale*;
- ⇒ *Transizione* (transition): beta test, rilascio. Milestone: *Rilascio prodotto*.



Concetto sbagliato 3.2: L'ideazione e l'elaborazione sono fasi di requisiti

- ⇒ L'Ideazione non è una fase di requisiti, ma di fattibilità;
- ⇒ L'Elaborazione non è una fase di requisiti o di progettazione, ma una fase in cui si implementa in modo iterativo l'architettura del sistema e vengono ridotti i rischi maggiori.

3.2.1 Le discipline

Definizione 3.2.2: Discipline

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

Corollario 3.2.1 Elaborato

Un elaborato (artefatto o work product) è il termine generico che indica un qualsiasi prodotto di lavoro: codice, schemi di basi di dati, documenti di testo, diagrammi, modelli, etc.

Discipline ingegneristiche di UP:

- ⇒ *Modellazione del business*: attività che modellano il dominio del problema e il suo ambito;
- ⇒ *Requisiti*: attività di raccolta dei requisiti;

- ⇒ *Progettazione* (analysis and design): attività di analisi dei requisiti e progetto architettonale;
- ⇒ *Implementazione*: attività di progetto dettagliato e codifica del sistema, test sui componenti;
- ⇒ *Test*: attività di controllo di qualità, test di integrazione e di sistema;
- ⇒ *Rilascio*: attività di consegna e messa in opera.

Discipline di supporto di UP:

- ⇒ *Gestione delle configurazioni e del cambiamento*: attività di manutenzione durante il progetto;
- ⇒ *Gestione progetto*: attività di pianificazione e governo del progetto;
- ⇒ *Infrastruttura* (environment): attività che supportano il team di progetto, riguardo ai processi e strumenti utilizzati.

Note:-

Nonostante le fasi siano *sequenziali*, le discipline non lo sono (perchè si eseguono in ogni iterazione). Il numero di iterazioni dipende dal Project Manager.

Uso di UML in UP:

- ⇒ UP usa solo UML come linguaggio di modellazione;
- ⇒ I diagrammi UML si usano con variabilità, bisogna *personalizzare* UP;
- ⇒ I diagrammi si usano in UP seguendo le iterazioni e gli incrementi;
- ⇒ UP dice *quando* usare un diagramma;
- ⇒ In UP quasi tutto è *opzionale* eccetto che lo sviluppo iterativo e guidato dal rischio, la verifica continua della qualità e il codice;
- ⇒ La scelta delle pratiche e degli artefatti UP si riassume in un documento (*scenario di sviluppo*).

3.2.2 Che cosa sono i requisiti?

Definizione 3.2.3: Requisito

Un requisito è una *capacità* o una condizione a cui il sistema deve essere *conforme*.

Corollario 3.2.2 Sorgenti dei requisiti

I requisiti derivano da richieste degli utenti del sistema per risolvere dei problemi e raggiungere degli obiettivi. Possono essere:

- ⇒ *Requisiti funzionali*: descrivono il comportamento del sistema in termini di funzionalità offerte;
- ⇒ *Requisiti non funzionali*: le proprietà del sistema nel suo complesso (sicurezza, prestazioni, etc.).

Osservazioni 3.2.1

In UP bisogna gestire i requisiti: si utilizza un approccio sistematico per trovare, documentare, organizzare e tracciare i requisiti che cambiano di un sistema. Si inizia a programmare quando sono stati specificati il 10% o il 20% dei requisiti significativi.

Acquisizione sistematica dei requisiti:

- ⇒ Scrivere i Casi d'Uso con i clienti;
- ⇒ Workshop dei requisiti con sviluppatori e clienti;
- ⇒ Gruppi di lavoro con rappresentanti dei clienti;
- ⇒ Dimostrazione ai clienti dei risultati di ciascuna iterazione, per favorire un feedback.

Modello FURPS+:

- ⇒ *Funzionali* (F): requisiti funzionali e di sicurezza;
- ⇒ *Usabilità* (U): facilità d'uso del sistema;
- ⇒ *Affidabilità* (R - Reliability): disponibilità del sistema, capacità di tollerare guasti o di essere ripristinato;
- ⇒ *Prestazioni* (P): tempi di risposta, throughput, capacità e uso delle risorse;
- ⇒ *Sostenibilità* (S): facilità di modifica per riparazioni e miglioramenti, adattabilità, manutenibilità, localizzazione, configurazione, compatibilità;
- ⇒ *+*: vincoli di progetto, interoperabilità, operazionali, fisici, legali, etc.

Elaborati:

- ⇒ *Modello dei Casi d'Uso*: scenari tipi dell'utilizzo di un sistema;
- ⇒ *Specifiche supplementari*: ciò che non rientra nei Casi d'Uso, requisiti non funzionali o funzionali non esprimibili attraverso i Casi d'Uso;
- ⇒ *Glossario*: termini significativi, dizionario dei dati;
- ⇒ *Visione*: riassume i requisiti di 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 un sistema deve conformarsi.

3.3 Ideazione

Domanda 3.2

Che cos'è l'ideazione?

Risposta: l'ideazione permette di stabilire una visione completa e la portata del progetto (*studio di fattibilità*).

Durante l'ideazione:

- ⇒ Si analizzano il 10% dei Casi d'Uso;
- ⇒ Si analizzano i requisiti non funzionali più importanti;
- ⇒ Si realizza una stima dei costi;
- ⇒ Si prepara l'ambiente di sviluppo;
- ⇒ *Durata*: breve.

Osservazioni 3.3.1

Lo scopo dell'Ideazione non è di raccogliere tutti i requisiti, né di generare una stima o un piano di progetto

affidabile. Durante l'ideazione si cerca di capire se il progetto è fattibile e se ha senso.

Elaborato	Commento
Visione e studio economico	Describe obiettivi e vincoli di alto livello, fornisce un sommario del progetto.
Modello dei Casi d'Uso	Describe i requisiti funzionali del sistema. Vengono identificati i nomi della maggior parte dei Casi d'Uso.
Specifiche supplementari	Describe i requisiti non funzionali e i requisiti funzionali non esprimibili attraverso i Casi d'Uso.
Glossario	Definisce i termini significativi del dominio.
Lista dei Rischi e Piano di Gestione dei Rischi	Identifica i rischi principali e come affrontarli.
Prototipi e proof of concept	Dimostrano la fattibilità tecnica e la comprensione dei requisiti.
Piano dell'Iterazione	Fornisce una descrizione di cosa fare nella prima iterazione dell'elaborazione.
Piano delle Fasi e Piano di Sviluppo del Software	Ipotesi (poco precise) riguardo la fase di elaborazione.
Scenario di Sviluppo	Describe le pratiche e gli artefatti UP da usare.

3.3.1 Artefatti nell'Ideazione

Domanda 3.3

La documentazione non è troppa?

Risposta: lo scopo della documentazione non è nel documento in sè, ma nel pensare:

- ⇒ gli artefatti sono quelli che aggiungono valore;
- ⇒ sono parzialmente completati;
- ⇒ sono preliminari e approssimativi.

Note:-

Nessun documento è definitivo.

Definizione 3.3.1: Specifiche supplementari

Le *specifiche supplementari* raccolgono altri requisiti, informazioni e vincoli che non sono espressi nei Casi d'Uso o nel Glossario. Si deve mettere anche la cronologia delle versioni.

3.3.2 Tipologie di documenti

Definizione 3.3.2: Visione

Il documento *Visione* riassume alcune informazioni contenute nel modello dei Casi d'Uso e nelle Specifiche supplementari. Inoltre descrive brevemente il progetto ai partecipanti per stabilire una visione comune.

- ⇒ Obiettivi e problemi fondamentali ad alto livello^a;
- ⇒ Riepilogo delle caratteristiche di sistema.

^aSoprattutto per i requisiti non funzionali

Note:-

Spesso è utile iniziare da un Glossario.

Definizione 3.3.3: Glossario e dizionario dei dati

Il *Glossario* è un documento che definisce i termini significativi del dominio e le relazioni tra di essi. Si devono eliminare eventuali discrepanze per ridurre problemi di comunicazione e di ambiguità.

In UP il Glossario svolge anche il ruolo di *dizionario dei dati*: un documento di dati che si riferiscono ad altri dati^a, per esempio le regole di validazione.

^aMetadati.

Corollario 3.3.1 Regole di dominio

Le *regole di dominio* (o regole di Business^a) stabiliscono come può funzionare un dominio o un business.

^aViste in "Basi di dati".

3.4 Casi d'Uso

3.4.1 Disciplina dei requisiti

Definizione 3.4.1: Disciplina dei requisiti

La *disciplina dei requisiti* è il processo per scoprire cosa deve essere costruito e orientare lo sviluppo verso il sistema corretto.

Corollario 3.4.1 Requisiti di sistema

I requisiti di sistema sono le capacità e le condizioni a cui il sistema deve essere conforme.

Note:-

I requisiti di sistema sono scritti nel "linguaggio del cliente".

Passi principali:

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

Ogni requisito è caratterizzato da:

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

Definizione 3.4.2: Lista dei requisiti

La *lista dei requisiti* è usata per stimare la taglia del progetto e per decidere come suddividere il lavoro in sequenze di iterazioni.

3.4.2 Capire il contesto del sistema e catturare i requisiti

Ci sono due modi per capire il contesto del sistema:

- ⇒ **Modello di dominio**: descrive i concetti significativi del sistema come oggetti del dominio e relaziona i concetti con associazioni (usa UML);
- ⇒ **Modello di business**: un super-insieme del modello di dominio, descrive i processi di business. È un prodotto dell'ingegneria del business e ha lo scopo di migliorare i processi di business.

Note:-

Il contesto del sistema è catturato dal *diagramma UML dei Casi d'Uso*.

Catturare i requisiti funzionali:

- ⇒ In UP vengono usati i Casi d'Uso;
- ⇒ Un Caso d'Uso rappresenta un possibile utilizzo del sistema da parte di un utente;
- ⇒ Sono descrizioni testuali.

Catturare i requisiti non funzionali:

- ⇒ Si utilizzano le Specifiche supplementari;
- ⇒ Possono anche essere catturati nei Casi d'Uso.

3.4.3 Casi d'Uso e UP

UP è una metodologia "use-case driven":

- ⇒ I Casi d'Uso si usano per pianificare le iterazioni;
- ⇒ L'analisi e la progettazione si basano sui Casi d'Uso;
- ⇒ I Casi d'Uso sono usati per definire i test;
- ⇒ I Casi d'Uso influiscono nella redazione dei manuali utente e della Visione;
- ⇒ **Attori**: qualcosa o qualcuno dotato di *comportamento*;
- ⇒ **Scenario** (istanza di Caso d'Uso): *sequenza specifica di azioni e interazioni* tra il sistema e alcuni attori. Descrive una particolare storia;
- ⇒ **Caso d'Uso**: *insieme di scenari* che descrivono un attore che usa il sistema per *raggiungere un obiettivo specifico*.

Note:-

I Casi d'Uso possono essere di successo o di fallimento.

Concetto sbagliato 3.3: I Casi d'Uso sono diagrammi

I Casi d'Uso sono documenti di testo, non diagrammi.

Esempio 3.4.1 (Caso d'Uso: breve descrizione)

Gestisci Restituzione (Handle Returns)

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

Scenari alternativi:

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

Se il codice identificativo dell'articolo non viene trovato nel sistema, il sistema avvisa il cassiere e suggerisce l'inserimento manuale del codice (può darsi che sia danneggiato).

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

Definizione 3.4.3: Modello dei Casi d'Uso

Il *modello dei Casi d'Uso* è un modello delle funzionalità del sistema. Include un diagramma UML dei Casi d'Uso che funge da *modello di contesto* del sistema e da *indice dei nomi* di Caso d'Uso.

Note:-

I Casi d'Uso sono utili per rappresentare i requisiti come OOA/D.

I Casi d'Uso definiscono i *contratti* (vedi sezione 3.8) in relazione al comportamento del sistema.

L'enfasi è posta sull'utente:

- ⇒ Chi utilizza il sistema?
- ⇒ Quali sono i loro Scenari d'Uso tipici?
- ⇒ Quali sono i loro obiettivi?
- ⇒ Non sono caratteristiche del sistema (il "come" si vedrà nella progettazione).

3.4.4 Attori e tipi di attori

Definizione 3.4.4: Attore

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

Note:-

Il sistema stesso è considerato un attore.

Gli attori sono **ruoli** svolti da persone, organizzazioni, software e macchine:

⇒ *Attore primario:*

- Raggiunge gli obiettivi utente utilizzando il sistema;
- Utile per trovare gli obiettivi utente.

⇒ *Attore di supporto:*

- Offre un servizio al sistema;
- Utile per chiarire le interfacce esterne e i protocolli.

⇒ *Fuori scena*:

- Ha un interesse nel comportamento del Caso d'Uso;
- Utile per garantire che tutti gli interessi necessari vengano soddisfatti.

3.4.5 Formato di un Caso d'Uso

Un Caso d'Uso può avere tre formati distinti:

- ⇒ *Breve*: un paragrafo, relativo allo scenario principale di successo. Serve a capire rapidamente l'*argomento* e la *portata*;
- ⇒ *Informale*: più paragrafi, scritti in modo informale, relativi a vari scenari. Si ha un maggior livello di dettaglio;
- ⇒ *Dettagliato*: tutti i passi e le variazioni sono scritti in dettaglio, include *pre-condizioni* e *garanzie di successo*. Si scrivono a partire da un formato breve o informale.

Note:-

Durante l'Ideazione il 10% dei Casi d'Uso è riportato in formato dettagliato utilizzando appositi *template* (e.g. Cockburn).

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 fornire i suoi servizi per raggiungere un obiettivo.
Parti interessate e interessi	Chiunque abbia un interesse nel comportamento del Caso d'Uso e che cosa desidera.
Pre-condizioni	Stato del sistema prima che il Caso d'Uso inizi. Ciò che vale la pena di dire al lettore.
Garanzia di successo	Che cosa deve essere vero se il Caso d'Uso viene completato con successo. Ciò che 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 o di fallimento.
Requisiti speciali	Requisiti non funzionali correlati.
Elenco delle varianti tecnologiche e dei dati	Varianti dei metodi di I/O e nel formato dei dati.
Frequenza di ripetizione	Quanto spesso si prevede che il Caso d'Uso venga eseguito.
Varie	Altri aspetti.

3.4.6 Come scrivere un Caso d'Uso

Sezioni del Caso d'Uso:

- ⇒ *Portata*: descrive i confini del sistema;
- ⇒ *Livello*: "obiettivo utente" o "sottofunzione";
- ⇒ *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;
- ⇒ *Parti interessate*: chiunque abbia un interesse nel comportamento del Caso d'Uso;
- ⇒ *Pre-condizioni*: lo stato del sistema prima che il Caso d'Uso inizi. Non vengono verificate all'interno del Caso d'Uso;
- ⇒ *Garanzie di successo* (post-condizioni): ciò che deve essere vero se il Caso d'Uso viene completato con successo.

Caratteristiche:

- ⇒ Lo scenario principale viene anche chiamato "*percorso felice*", "flusso di base" o "flusso tipico";
- ⇒ Lo scenario principale è costituito da una sequenza di passi, che può contenere passi da ripetere, ma che non comprende nessuna diramazione¹;
- ⇒ La gestione del comportamento condizionale e delle alternative viene descritta nelle "*estensioni*".

I passi possono essere di 3 tipi:

- ⇒ *Un'interazione tra attori*:
 - Un attore chiede qualcosa al sistema o inserisce dei dati;
 - Il sistema interagisce con l'attore, rispondendo o comunicando dei dati;
 - Il sistema interagisce con altri sistemi.
- ⇒ *Un cambiamento di stato da parte del sistema*;
- ⇒ *Una validazione*: normalmente fatta dal sistema.

Note:-

Il primo passo indica l'evento *trigger* che scatena l'esecuzione dello scenario.

Estensioni:

- ⇒ Descrivono tutti gli altri scenari;
- ⇒ Sono descritte per differenza rispetto allo scenario principale;
- ⇒ Vengono indicate con riferimento a un passo dello scenario principale;
- ⇒ Sono composte da:
 - *Condizione*: che scatena l'estensione;
 - *Gestione*: come viene gestita l'estensione. Può essere di un unico passo o di più passi.

Note:-

Le condizioni vanno scritte, se possibile, come qualcosa che può essere rilevato da un attore.

Corollario 3.4.2 Utilizzo delle estensioni

- ⇒ L'attore vuole che l'esecuzione principale del Caso d'Uso proceda in modo diverso da quanto previsto nel percorso felice;
- ⇒ Il Caso d'Uso deve procedere diversamente da quanto previsto ed è il sistema che se ne accorge (durante un'azione o una validazione);
- ⇒ Un passo dello scenario principale descrive un'azione generica o astratta, mentre le estensioni descrivono azioni specifiche o concrete.

¹Non è un algoritmo.

Notazione:

⇒ Si può utilizzare il formato a due colonne. Questo enfatizza la *conversazione* tra gli attori e il sistema².

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.

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

6. Il Cassiere riferisce il totale al Cliente e richiede il pagamento.
7. Il Cliente paga.

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

Risponsabilità 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.
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.

Scenario principale di successo

#	Attore	Sistema
1	Decide di creare un nuovo menu	
2	Specifica un titolo per il menu.	Mostra i dettagli (titolo) del menu creato
3	Definisce una sezione del menu assegnandole un nome.	Mostra la sezione con il suo nome
4	Inserisce una voce nel menu 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 menu è a suo avviso completo e quindi utilizzabile.	Segnala che il menu è ora completo.
6	Conclude il lavoro su questo menu.	

(a) Rappresentazione come conversazione

(b) Relativa tabella

⇒ Le estensioni vanno indicate con riferimento al passo dello scenario principale.

Estensione 1a: lavora su un menu esistente

#	Attore	Sistema
1a.1	Sceglie di lavorare su un menu precedentemente creato.	Mostra i dettagli (titolo, sezioni, voci) del menu scelto
	<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 1b: crea un menu come copia di un altro

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

Estensione 3b: elimina una sezione esistente

#	Attore	Sistema
3b.1	Elimina una sezione precedentemente creata.	Mostra il menu aggiornato (senza la sezione eliminata). Se sono state eliminate tutte le sezioni e il menu era indicato come completo, segnala che non lo è più.
	<i>Se vuole lavorare su un'altra sezione torna al passo 3 se ne prosegue con il passo 5.</i>	

(a) Estensione del punto 1

(b) Estensione del punto 3

⇒ Ci possono essere alternative che possono occorrere in più passi.

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

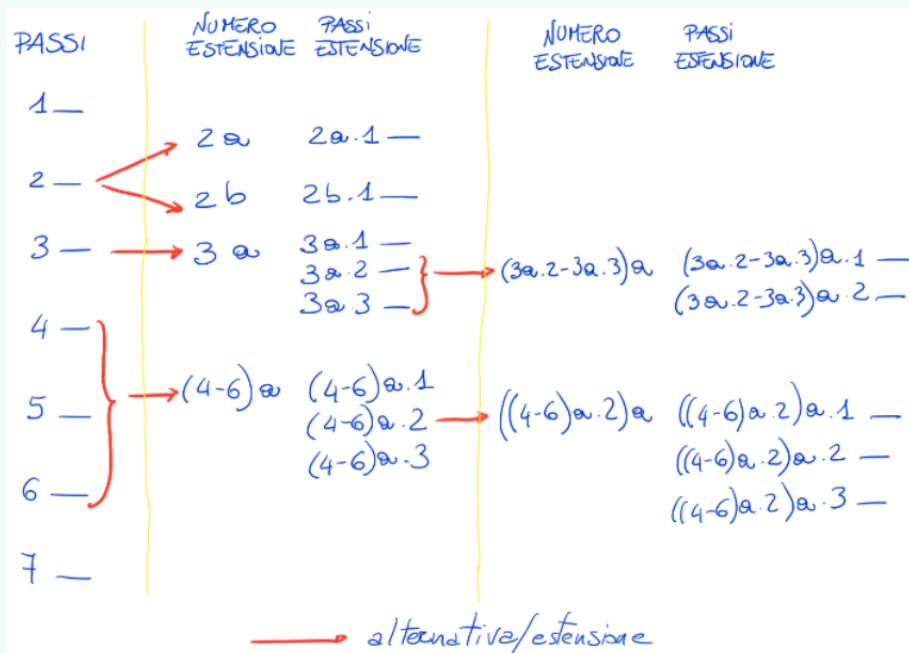
#	Attore	Sistema
(3-4)a.1	Specifica un nuovo titolo per il menu.	Mostra i dettagli del menu aggiornati (con il nuovo titolo)

Estensione (3-5)a: conclude anticipatamente

#	Attore	Sistema
	<i>Va al passo 6 dello scenario principale</i>	

²È il formato scelto per il laboratorio del corso.

Esempio 3.4.2 (Passi ed estensioni)



Note:-

Lo stile deve essere *essenziale* e *coinciso*. Si ignora l'interfaccia utente, ci si concentra sull'Obiettivo Utente.

Definizione 3.4.5: Stile essenziale

La narrativa è espressa a livello di *intenzioni* e *responsabilità*, non con riferimento ad azioni concrete. Le intenzioni e le responsabilità devono rimanere indipendenti dai dettagli tecnologici e dagli attori.

Esempio 3.4.3 (Stile)

- ✓ L'Amministratore si identifica.
- ✓ Il Sistema autentica l'identità.
- ✗ L'Amministratore inserisce ID e password nella finestra di dialogo.
- ✗ Il sistema autentica l'Amministratore.
- ✗ Il sistema visualizza la finestra "edit users".

Osservazioni 3.4.1

Durante l'analisi dei requisiti bisogna specificare il comportamento esterno del Sistema, considerato a "scatola nera". Non bisogna prendere decisioni sul "come".

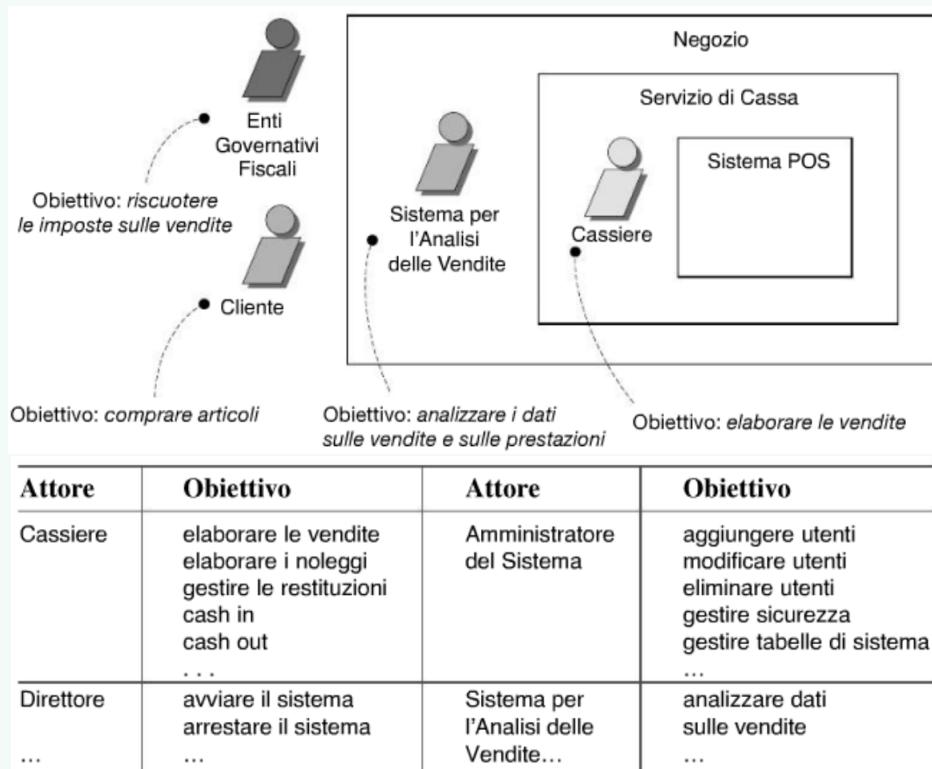
- ✗ Il Sistema memorizza la vendita in una base di dati.
- ✗ Il Sistema esegue un'istruzione SQL INSERT per la vendita.

3.4.7 Trovare i Casi d'Uso

1. Scegliere i confini del Sistema;

2. Identificare gli attori primari;
3. Identificare gli obiettivi di ciascun attore primario;
4. Definire i Casi d'Uso che soddisfano gli obiettivi degli utenti.

Esempio 3.4.4 (Identificare gli attori primari)



Domanda 3.4

Qual è un livello utile per esprimere i Casi d'Uso nell'analisi dei requisiti di un'applicazione software?

- ⇒ Il test del *capo*: il capo sarà felice?
- ⇒ Il test *EBP* (Elementary Business Process): un processo di Business è un'attività che aggiunge un valore;
- ⇒ Il test della *dimensione*: un Caso d'Uso raramente richiede una singola azione o passo. Normalmente, nella forma dettagliata, richiede dalle 3 alle 10 pagine.

Esempio 3.4.5 (Verificare l'utilità dei Casi d'Uso)

- ✗ Negoziare un contratto con un fornitore.
⇒ Troppo grande.
- ✓ Gestire una restituzione.
⇒ D'accordo con il capo, simile a EBP, dimensioni adeguate.
- ✗ Effettuire il login.
⇒ Il capo non è contento se ci si limita a fare solo quello tutta la giornata.

- ✗ Spostare una pedina sul tabellone da gioco.
⇒ Passo singolo, non supera il test della dimensione.

3.4.8 Livello dei Casi d'Uso

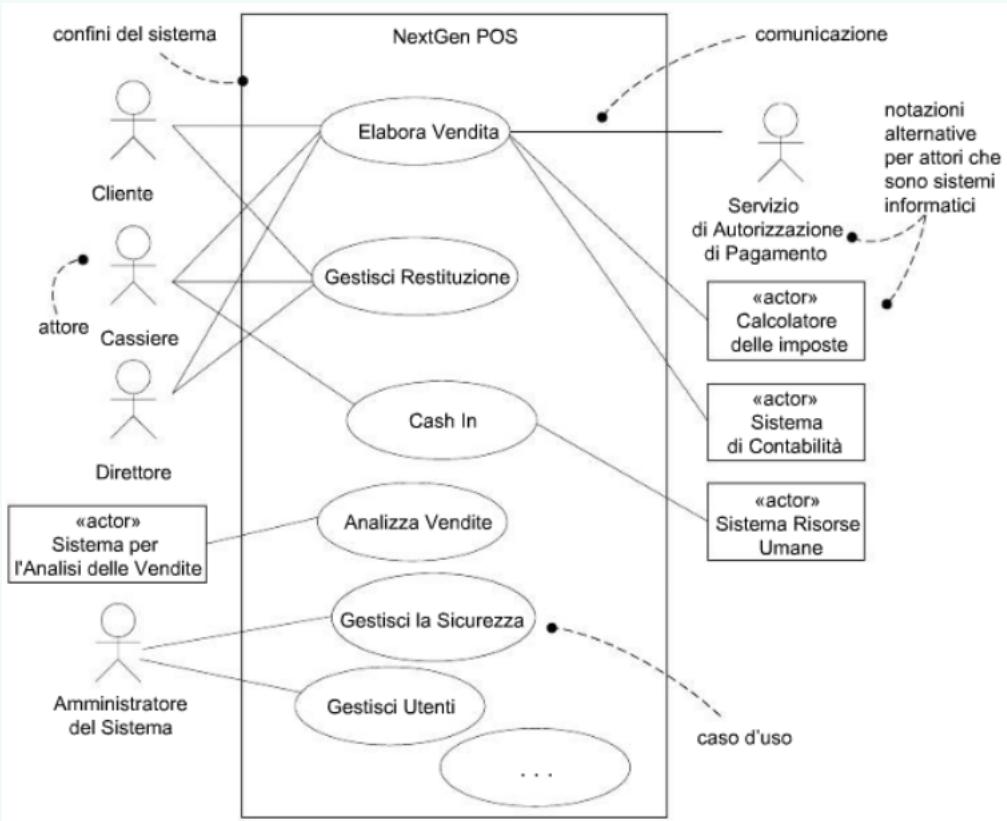
Definizione 3.4.6: Livello di obiettivo utente

Nell'analisi dei requisiti è utile concentrarsi sui casi utente EBP.

Definizione 3.4.7: Livello di sotto-funzione

Rappresenta una funzionalità nell'uso del sistema. Utile per mettere a fattor comune sequenze di passi condivise da più Casi d'Uso, evitando duplicazione di testo.

Esempio 3.4.6 (Diagramma dei Casi d'Uso (UCD))



3.5 Elaborazione

Definizione 3.5.1: Elaborazione

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

Note:-

Durante questa fase vengono creati prototipi "usa e getta" ma codice e progettazione sono parti di qualità-produzione del sistema finale.

3.5.1 Pianificazione dell'iterazione successiva

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;
- ⇒ **Criticità**: le funzioni che il cliente ritiene più importanti.

Osservazioni 3.5.1

La classifica viene stilata prima dell'iterazione 1, poi prima dell'iterazione 2 e così via. La lista non è definitiva.

Voto	Requisito (Caso d'uso o Caratteristica)	Commento
Alto	Elabora Vendita Logging ...	Ottiene voti alti per tutti i criteri. Pervasivo. Difficile da aggiungere in un secondo momento. ...
Medio	Gestire utenti ...	Influisce sul sottodominio di sicurezza. ...
Basso

Definizione 3.5.2: Iterazione 1

Nell'iterazione 1 si implementa un sottoinsieme dei requisiti o dei Casi d'Uso completi.

3.5.2 Artefatti dell'Elaborazione

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

3.6 Modello di Dominio

Definizione 3.6.1: Modello di Dominio

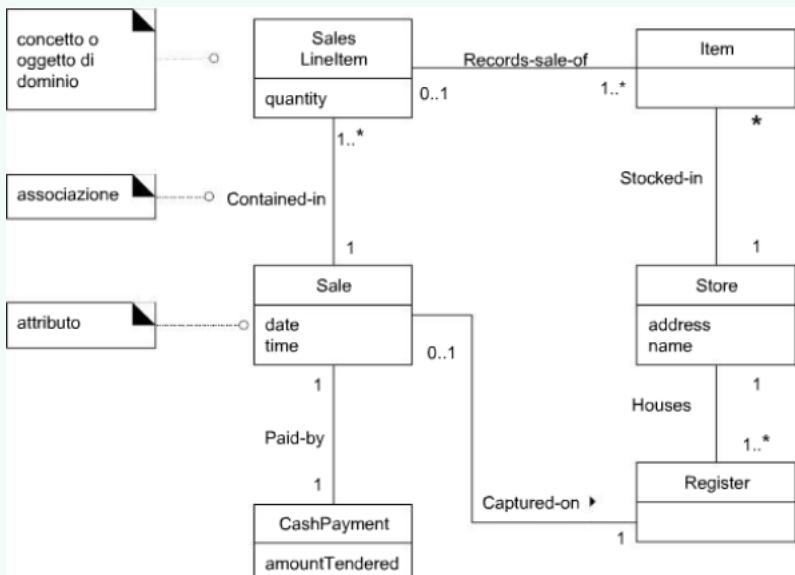
Il *Modello di Dominio* è una rappresentazione visuale delle classi concettuali (oggetti del dominio). Include:

- ✓ *Oggetti* di dominio.
- ✓ *Associazioni* tra classi concettuali.
- ✓ *Attributi* di classi concettuali.
- ✗ *Operazioni*.

Note:-

Non è un modello di dati.

Esempio 3.6.1 (Modello di dominio)



3.6.1 Classi concettuali

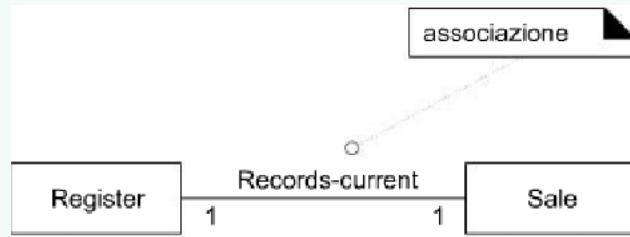
Definizione 3.6.2: Classi concettuali

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

- ⇒ Il *simbolo* è una parola o un'immagine usata per rappresentare la classe concettuale;
- ⇒ L'*intensione* è la definizione della classe concettuale;
- ⇒ L'*estensione* è l'insieme di oggetti che la classe concettuale rappresenta.

Corollario 3.6.1 Associazioni

Un'associazione è una relazione tra istanze di classi che indica una connessione significativa e interessante.

Esempio 3.6.2 (Associazione)**Corollario 3.6.2 Attributi**

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

Esempio 3.6.3 (Attributo)

Riduzione del "gap di rappresentazione":

- ⇒ *Comprendere* il dominio del sistema da realizzare e il suo vocabolario;
- ⇒ *Definire* un *linguaggio comune* che abiliti la comunicazione tra le diverse parti;
- ⇒ Come *fonte di ispirazione* per la progettazione dello strato di dominio.

Domanda 3.5

Come creare un Modello di Dominio

- ⇒ Trovare le classi concettuali;
- ⇒ Disegnarle come classi in UML;
- ⇒ Aggiungere le associazioni;
- ⇒ Aggiungere gli attributi.

3.6.2 Trovare le classi concettuali

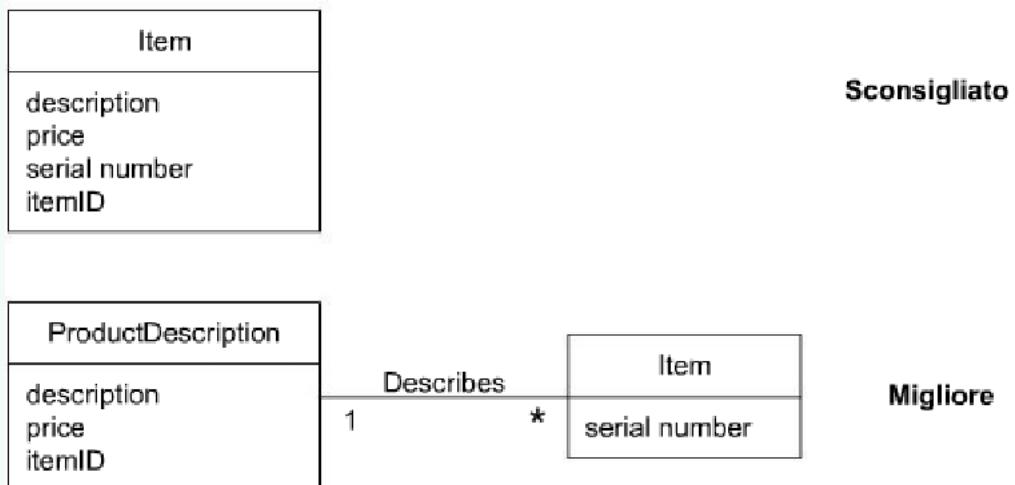
- ⇒ Riuso-modifica di modelli esistenti (pattern);
- ⇒ Utilizzo di elenchi di categorie;
- ⇒ Analisi linguistica delle descrizioni testuali di un dominio. Si utilizzano i Casi d'Uso dettagliati.

Definizione 3.6.3: Classi Descrizione

Una *classe descrizione* contiene informazioni che descrivono qualcos'altro. È utile avere un'associazione che collega la classe descrizione alla classe descritta^a.

^aPattern Item-Descriptor.

Esempio 3.6.4 (Classe descrizione)



3.6.3 Associazioni

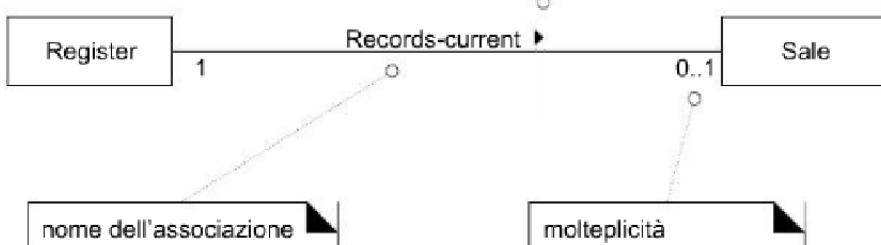
È utile includere:

- ⇒ Associazioni la cui conoscenza della relazione deve essere mantenuta dal sistema;
- ⇒ Associazioni derivate dall'elenco di associazioni comuni.

Note:-

Un'associazione è per natura *bidirezionale*.

► è la "freccia della direzione di lettura".
Questa freccia **non** ha significato, tranne che indicare la direzione di lettura del nome dell'associazione. Spesso viene esclusa.



Caratteristiche delle associazioni:

⇒ *Nome significativo*: NomeClasse-FraseVerbale-NomeClasse;

⇒ *Molteplicità e direzione di lettura*.

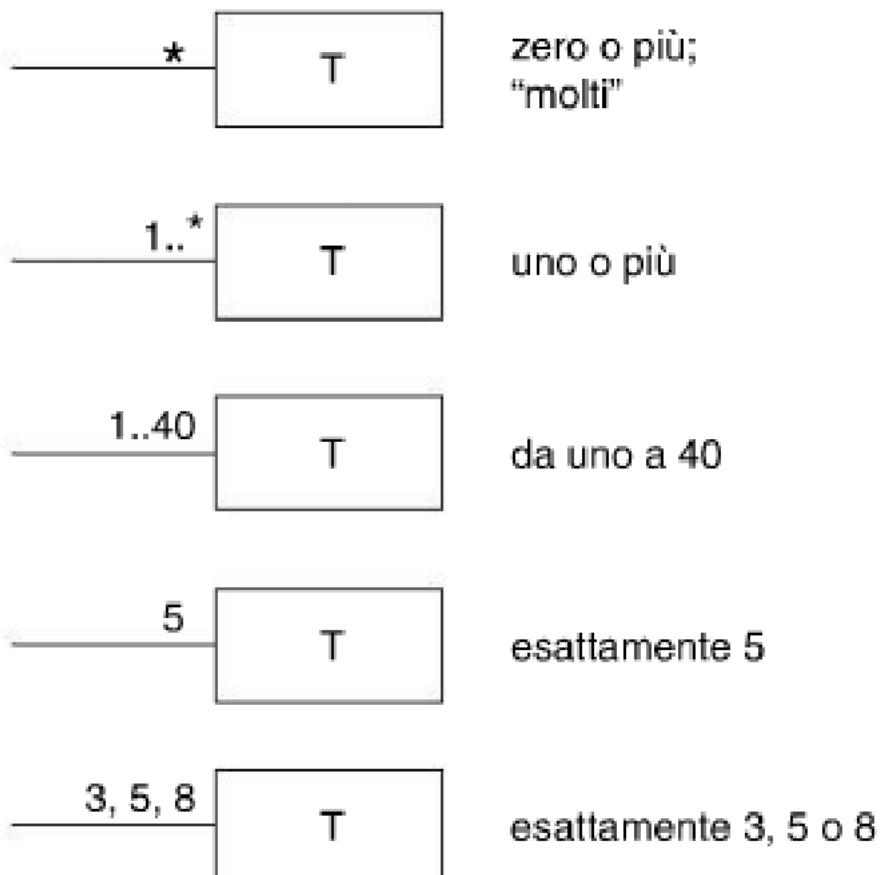
Definizione 3.6.4: Ruoli

Un *ruolo* è l'estremità di un'associazione. I ruoli possono avere:

- ⇒ *Nome*;
- ⇒ *Molteplicità*;
- ⇒ *Navigabilità*.

Corollario 3.6.3 Molteplicità

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

Esempio 3.6.5 (Molteplicità)

3.6.4 Composizione

Definizione 3.6.5: Composizione

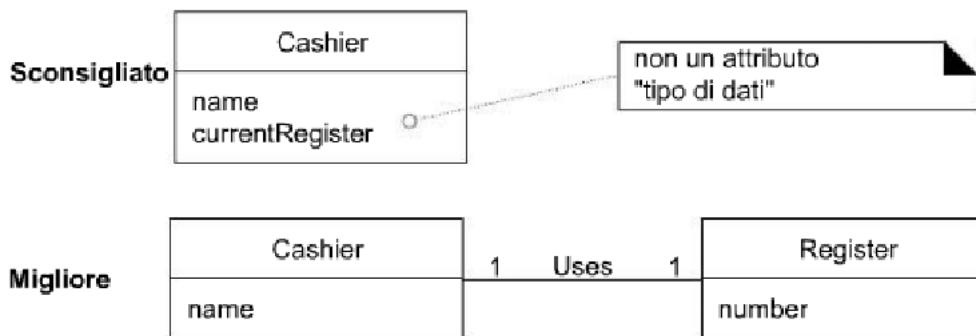
La *composizione*, o aggregazione composta, è un tipo forte di aggregazione intero-parte:

- ⇒ Ciascuna istanza della parte appartiene a una sola istanza del composto alla volta;
- ⇒ La parte non può esistere senza il composto;
- ⇒ La vita delle parti è limitata a quella del composto.

3.6.5 Attributi

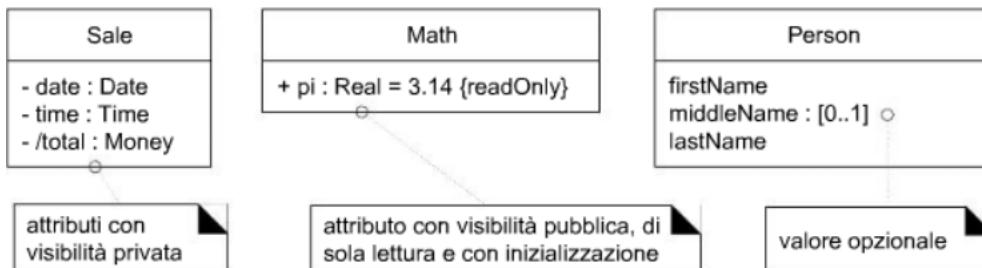
Caratteristiche degli attributi:

- ⇒ *Origine*: derivati/non derivati;
- ⇒ *Tipo di dato*: vincolo sui valori del dominio.



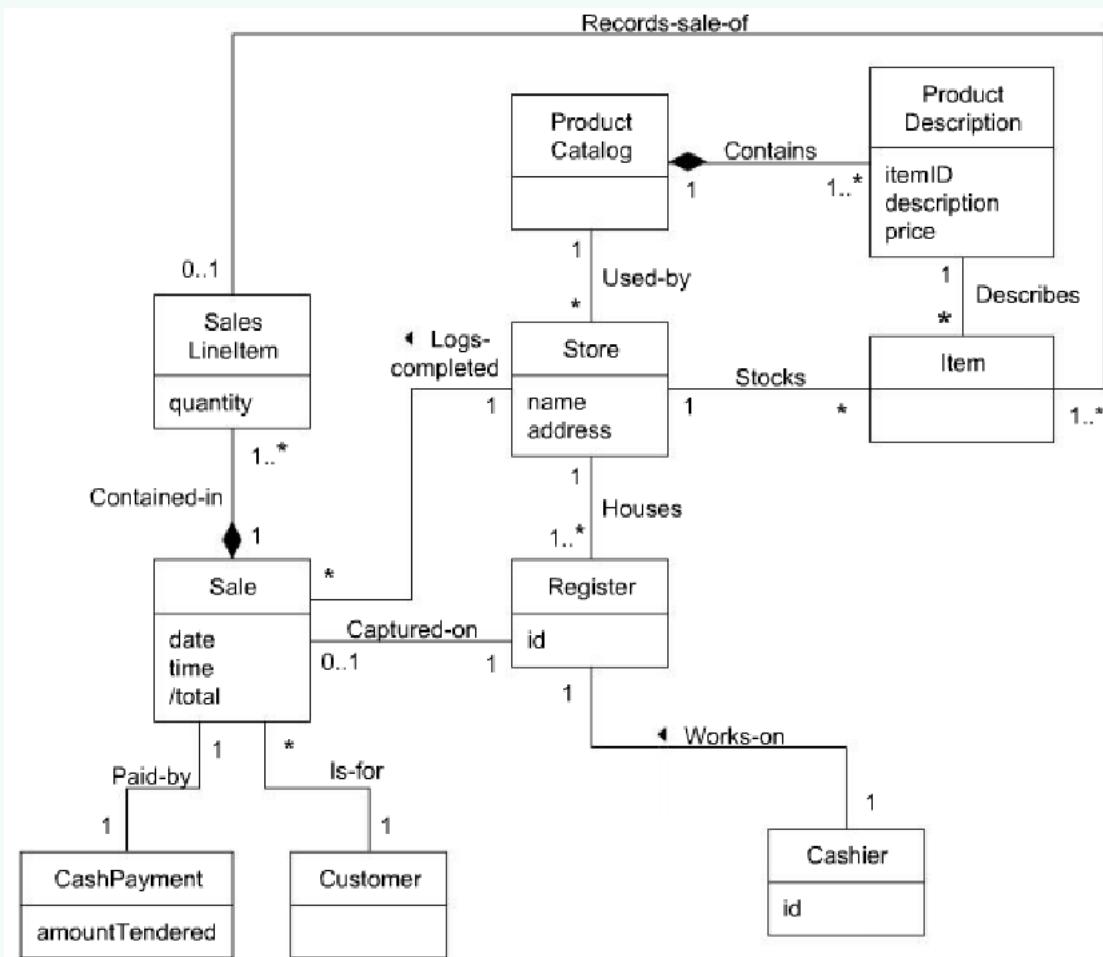
Note:-

Se non si pensa a un concetto come a un numero, un testo o un valore allora probabilmente è una classe, non un attributo.

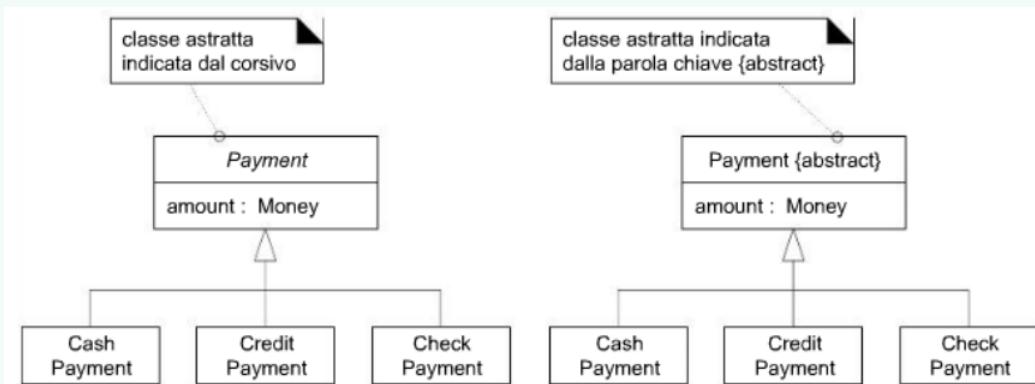


3.6.6 Verifiche del modello

- ⇒ Verificare le classi concettuali introdotte:
 - Alternativa classe-attributo attributo;
 - Classi descrizione.
- ⇒ Verificare le associazioni:
 - Indipendenza delle associazioni diverse relative alle stesse classi.
- ⇒ Verificare gli attributi:
 - Non introdurre attributi per riferirsi ad altre classi (chiavi esterne).

Esempio 3.6.6 (Modello di Dominio (parziale))**Definizione 3.6.6: Generalizzazione**

La *generalizzazione* è un’astrazione basata sull’identificazione di caratteristiche comuni tra concetti, che porta a definire un concetto più generale (superclasse) e concetti più specifici (sottoclassi). Vale il principio di sostituibilità.

Esempio 3.6.7 (Generalizzazione)

3.7 Diagrammi di sequenza del sistema (SSD)

Definizione 3.7.1: SSD

Il *Diagramma di Sequenza del Sistema* è un elaborato della disciplina dei requisiti che illustra eventi di input e di output relativi ai sistemi in discussione.

È una figura che mostra, per un particolare Caso d'Uso, gli eventi generati dagli attori esterni al sistema, il loro ordine e gli eventi inter-sistema.

Note:-

Di solito si usa un SSD per ogni Caso d'Uso.

Corollario 3.7.1 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.

- ⇒ Le operazioni di sistema sono operazioni che il sistema deve definire proprio per gestire tali eventi;
- ⇒ Un evento deve essere degno di nota;
- ⇒ Un evento di sistema è un evento esterno al sistema generato da un attore per interagire con il sistema.

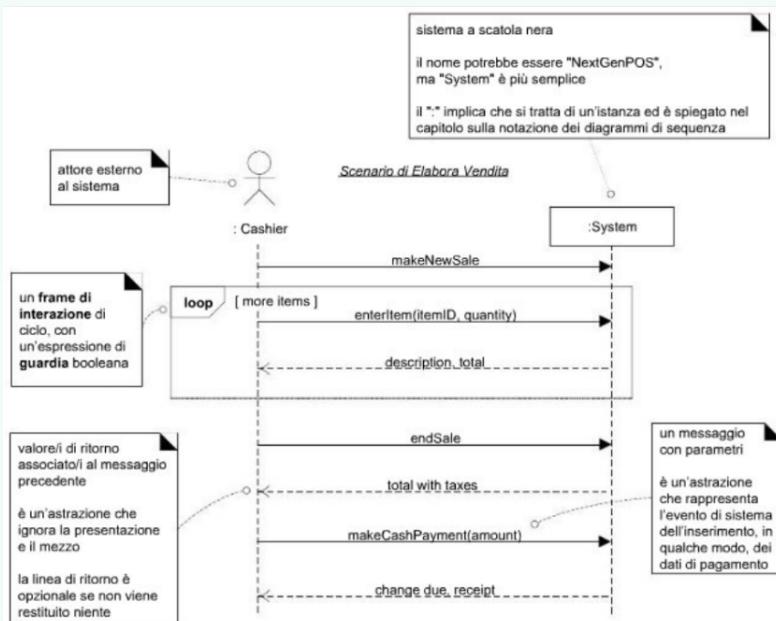
Un sistema reagisce a:

- ⇒ *Eventi esterni*: generati da attori umani o sistemi informatici;
- ⇒ *Eventi temporali*;
- ⇒ *Guasti o eccezioni*.

Note:-

Il software deve essere progettato per gestire questi eventi.

Esempio 3.7.1 (SSD)



- ⇒ *makeNewSale*: il cassiere inizia una nuova vendita;
- ⇒ *enterItem*: il cassiere inserisce il codice identificativo di un articolo;
- ⇒ *endSale*: il cassiere indica di aver terminato l'inserimento degli articoli acquistati;
- ⇒ *makeCashPayment*: il cassiere indica che il cliente sta pagando in contanti e inserisce l'importo offerto dal cliente.

Un SSD mostra:

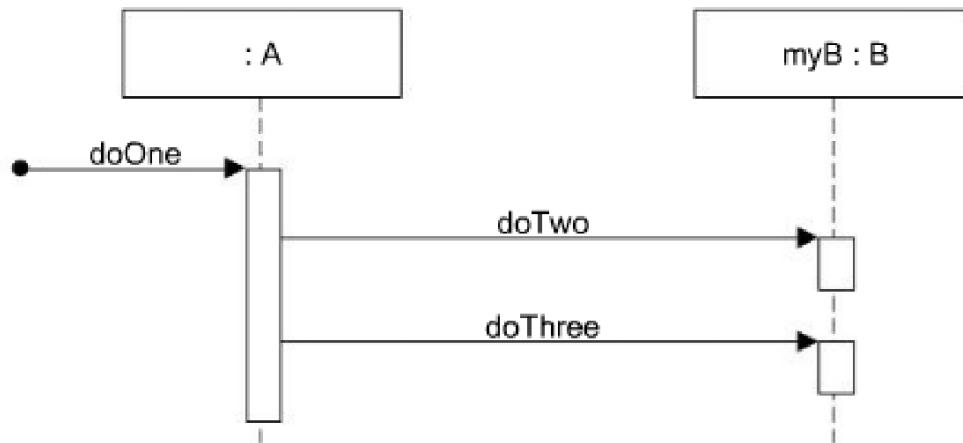
- ⇒ L'attore primario del Caso d'Uso;
- ⇒ Il sistema in discussione;
- ⇒ I passi che rappresentano le interazioni tra il sistema e l'attore.

Note:-

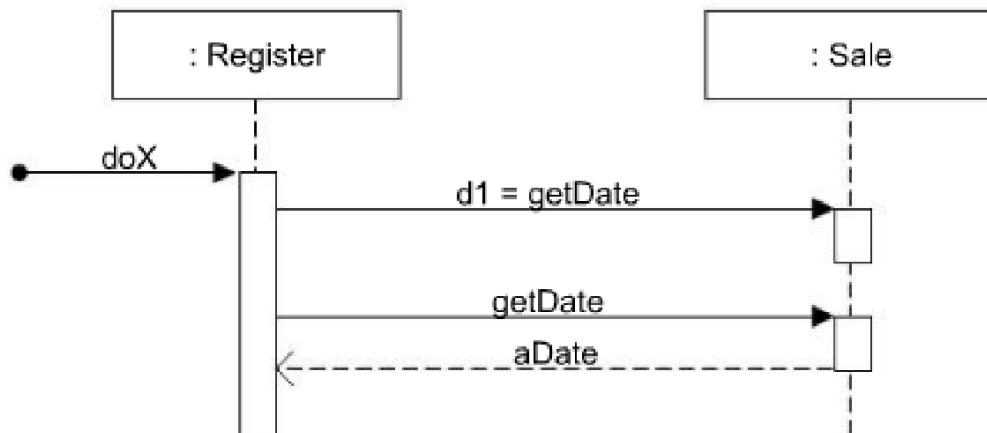
Le interazioni sono mostrate come messaggi con parametri.

3.7.1 Notazione

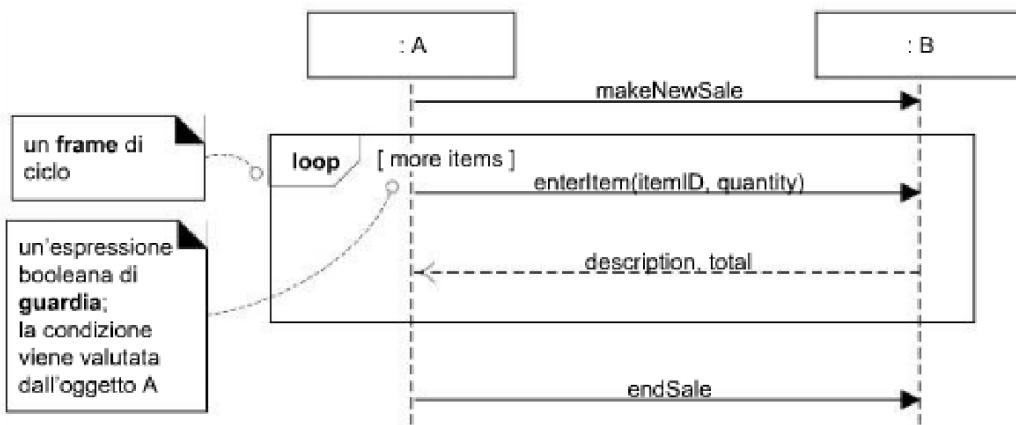
Semplice diagramma di sequenza:



Mostrare il risultato di ritorno:



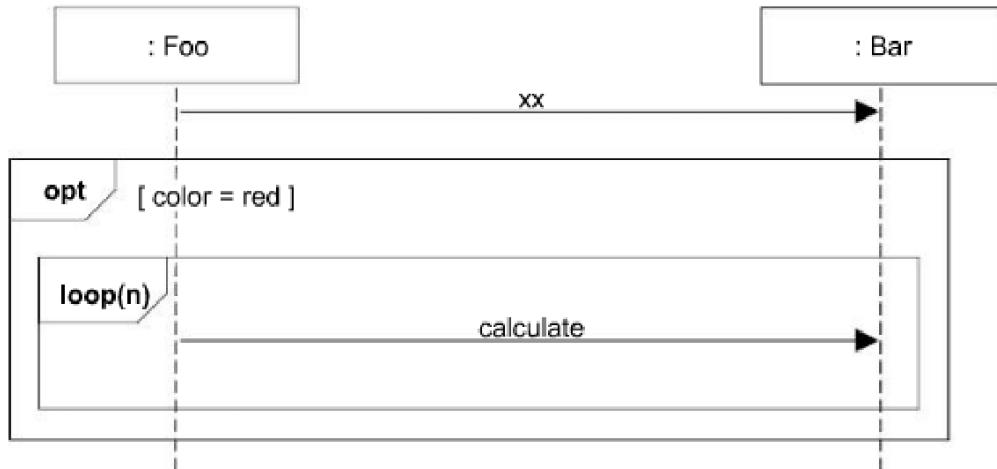
Frame di UML:



Operatori comuni per i frame UML:

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

Annidamento di frame:



3.8 Contratti

3.8.1 Pre-condizioni e Post-condizioni

Definizione 3.8.1: I contratti delle operazioni

I contratti usano *pre-condizioni* e *post-condizioni* per descrivere nel dettaglio i cambiamenti agli oggetti in un Modello di Dominio.

Punto di vista:

- ✓ Concettuale.
- ✗ Implementativo.

Note:-

Possono essere considerati parte del Modello dei Casi d'Uso, ma non sono menzionati esplicitamente in UP.

Esempio 3.8.1 (Contratto)

Contratto CO2: enterItem

Operazione: enterItem(itemID: ItemID, quantity: integer)
Riferimenti: casi d'uso: Elabora Vendita
Pre-condizioni: è in corso una vendita s

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

Template di un contratto:

- ⇒ *Operazione*: Nome e parametri dell'operazione;
- ⇒ *Riferimenti*: Casi d'Uso in cui può verificarsi l'operazione;
- ⇒ *Pre-condizioni*: Stato del sistema prima dell'operazione. Sono ipotesi non banali;
- ⇒ *Post-condizioni*: Stato del sistema dopo l'operazione.

Definizione 3.8.2: Post-condizioni

Le *post-condizioni* descrivono i cambiamenti nello stato degli oggetti del Modello di Dominio. I cambiamenti comprendono:

- ⇒ Oggetti creati;
- ⇒ Collegamenti formati;
- ⇒ Collegamenti rotti;
- ⇒ Attributi modificati.

Note:-

Le post-condizioni sono osservazioni al termine dell'operazione. Ci si concentra sul "cosa", ma non sul "come".

Definizione 3.8.3: Pre-condizioni

Le *pre-condizioni* sono ipotesi sullo stato del sistema prima dell'operazione. Sono condizioni che devono essere vere prima che l'operazione possa essere eseguita.

Note:-

Le pre-condizioni sono una descrizione sintetica sullo stato di avanzamento del Caso d'Uso.

3.8.2 Scrivere contratti

Si procede così:

1. Si identificano le operazioni dagli SSD;
2. Si crea un contratto per le operazioni complesse o i cui effetti sono sottili (non chiari dai Casi d'Uso);
3. Si scrivono le pre-condizioni e le post-condizioni:
 - ⇒ creazione o cancellazione di oggetti;
 - ⇒ formazione o rottura di collegamenti;
 - ⇒ modifica di attributi.

Note:-

Ogni operazione può avere una componente di:

- ⇒ *Trasformazione*: modifica lo stato del sistema;
- ⇒ *Interrogazione*: non modifica lo stato del sistema, vengono restituiti valori.

Un'operazione ha post-condizioni solo se è di trasformazione, mentre non le ha se è di interrogazione.

Domanda 3.6

Bisogna scrivere un contratto per ogni evento di sistema trovato nel SSD?

Risposta: Non è necessario, si considerano solo quelli più complessi.

Domanda 3.7

Se si scoprono nuove classi, attributi, si possono aggiungere nel modello di dominio?

Risposta: Sì, UP è incrementale.

Domanda 3.8

Le post-condizioni devono essere in ogni momento le più complete possibili?

Risposta: Non è necessario.

4

Architettura del software

4.1 Architettura logica

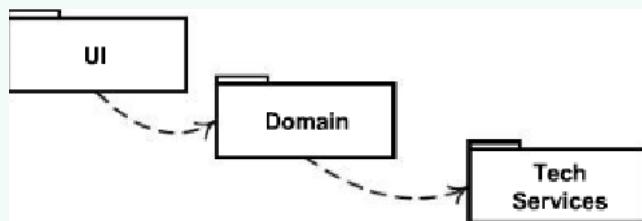
Definizione 4.1.1: Architettura

La progettazione OO è basata su diversi strati architetturali, come uno strato UI, uno strato di logica applicativa (o "del dominio") e così via.

Note:-

L'architettura può essere mostrata sotto forma di diagrammi UML.

Esempio 4.1.1 (Package UML)



Definizione 4.1.2: Architettura logica

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

Note:-

Non vengono prese decisioni su come gli elementi sono distribuiti sui processi o sui vari computer fisici di una rete (*platform independent architecture*).

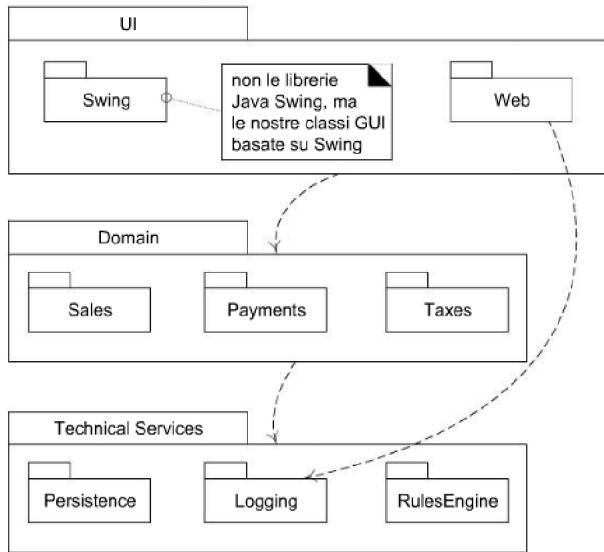
Corollario 4.1.1 Layer (o Strato)

Un layer è un gruppo di classi software, packages, sottosistemi con responsabilità condivisa su un aspetto importante del sistema.

Gli strati comprendono:

- ⇒ *UI (User Interface)*: strato che si occupa dell'interazione con l'utente;
- ⇒ *Application logic* (o "domain object"): oggetti software che rappresentano concetti di dominio;
- ⇒ *Technical services*: oggetti e sottosistemi che forniscono servizi tecnici (es. persistenza, sicurezza, ecc.);

Rappresentazioni di strati:



(a) Packages UML

```

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

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

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

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

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

(b) Relativo codice

Definizione 4.1.3: Architettura a strati

L'obiettivo di un'*architettura a strati* è la suddivisione di un sistema complesso in un insieme di elementi software che possano essere sviluppati e modificati in modo indipendente.

Corollario 4.1.2 Separation of concerns

La separazione degli interessi:

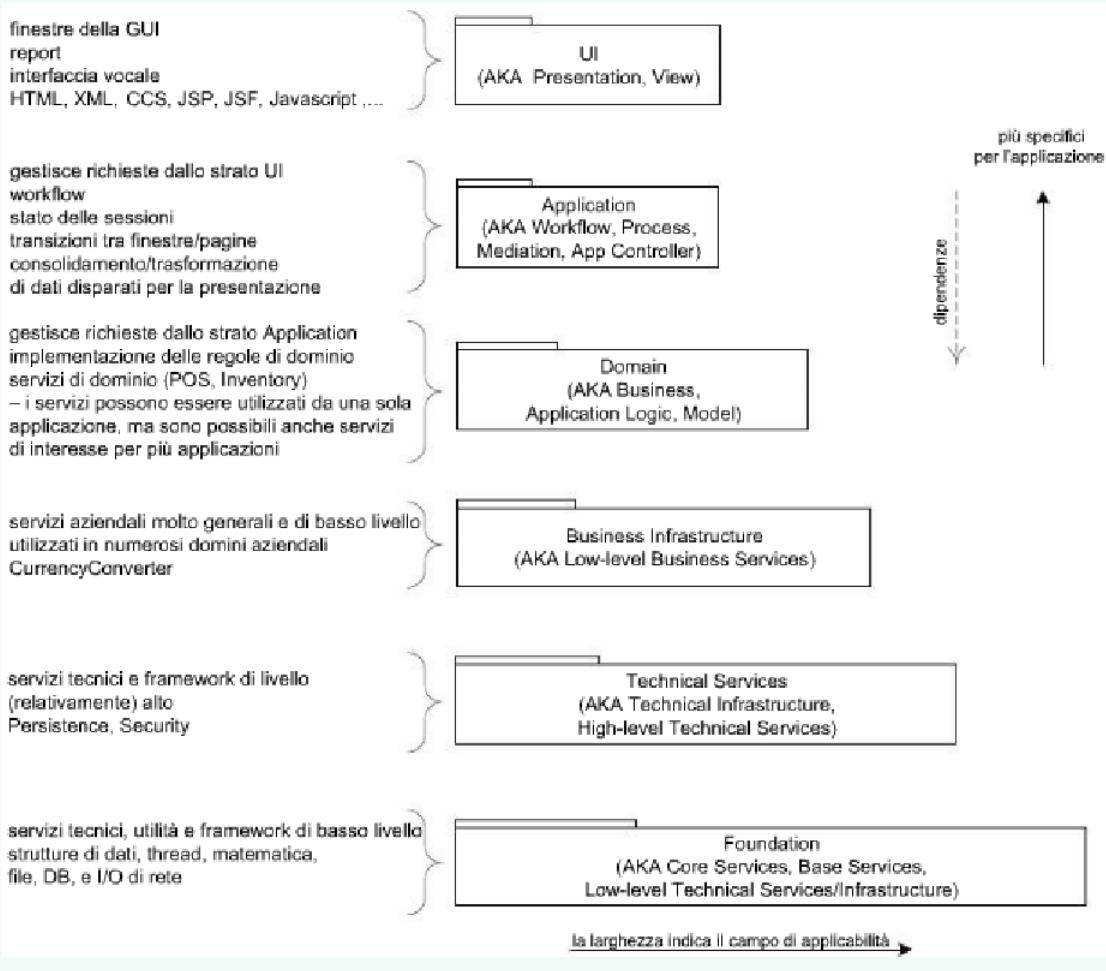
- ⇒ Riduce l'accoppiamento e le dipendenze;
- ⇒ Aumenta la possibilità di riuso;
- ⇒ Facilita la manutenzione e l'evoluzione del sistema;
- ⇒ Aumenta la chiarezza.

Corollario 4.1.3 Alta coesione

In uno strato le responsabilità degli oggetti devono essere fortemente collegate tra loro, ma non essere mischiare con quelle di altri strati.

Note:-

Questi due principi verranno ripresi in GRASP, vedi capitolo 6.

Esempio 4.1.2 (Architettura a strati)

4.2 Strato del dominio

Domanda 4.1

Come va progettata la logica applicativa con gli oggetti?

Risposta: Si creano oggetti software con nomi e informazioni simili al Dominio del mondo reale e assegnare a essi le responsabilità della logica applicativa.

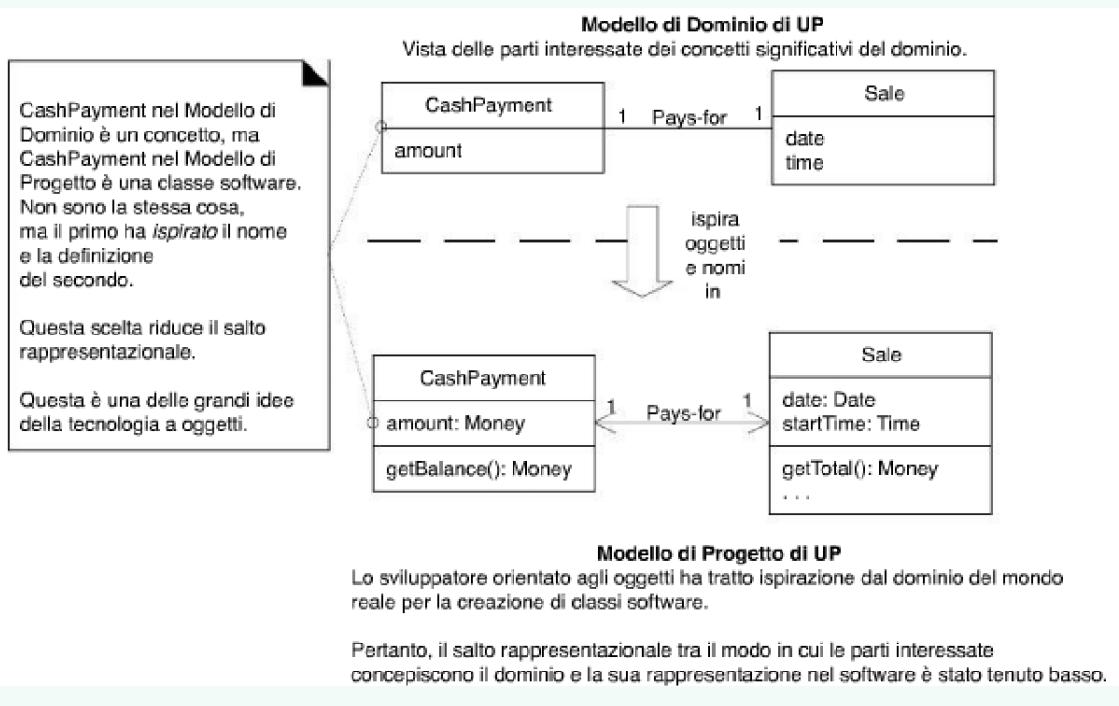
Definizione 4.2.1: Oggetto del Dominio

Un *oggetto del dominio* è un oggetto software che rappresenta un concetto del dominio del problema. Ha una logica applicativa o di business correlata.

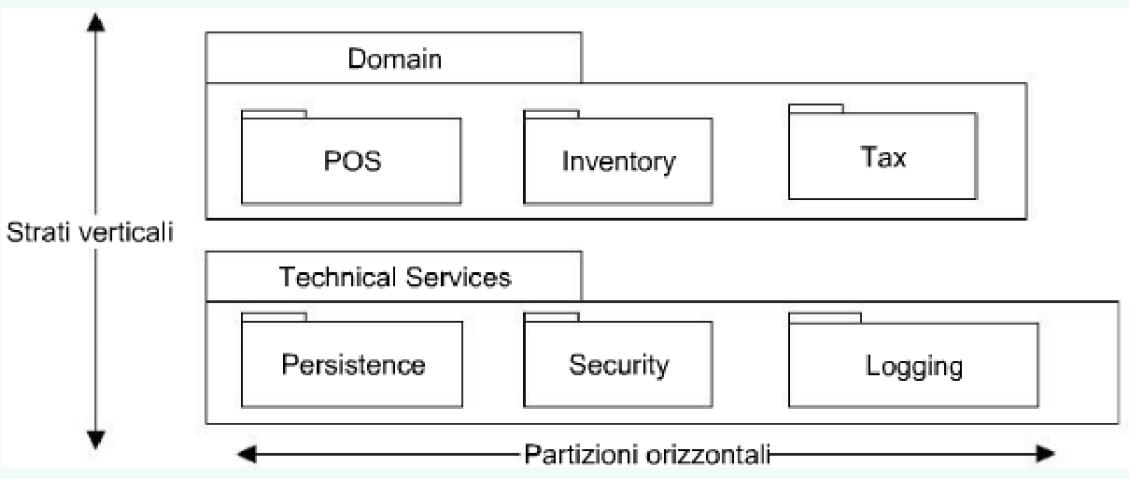
Note:-

Ispirandosi al Modello del Dominio si ha un basso salto rappresentazionale che rende più facile e veloce l'implementazione.

Esempio 4.2.1 (Strato di Dominio e Modello di Dominio)



Esempio 4.2.2 (Strati e Partizioni)



4.3 Separazione Modello-Vista

Definizione 4.3.1: Principio di separazione Modello-Vista

- ✗ Non relazionare o accoppiare oggetti non UI con oggetti UI.
- ✗ Non encapsulare la logica applicativa in metodi UI.
- ✓ Le finestre appartengono a una specifica applicazione, mentre gli oggetti non UI possono venire riutilizzati in nuove applicazioni o associati a nuove interfacce.
- ✓ Gli oggetti UI inizializzano elementi UI, ricevono eventi UI e delegano le richieste della logica dell'applicazione agli oggetti non UI.

Note:-

- ⇒ Il Modello è lo strato degli oggetti del dominio;
- ⇒ La Vista è lo strato UI.

Questo principio è alla base del pattern MVC^a.

^aVisto nel corso "Programmazione III".

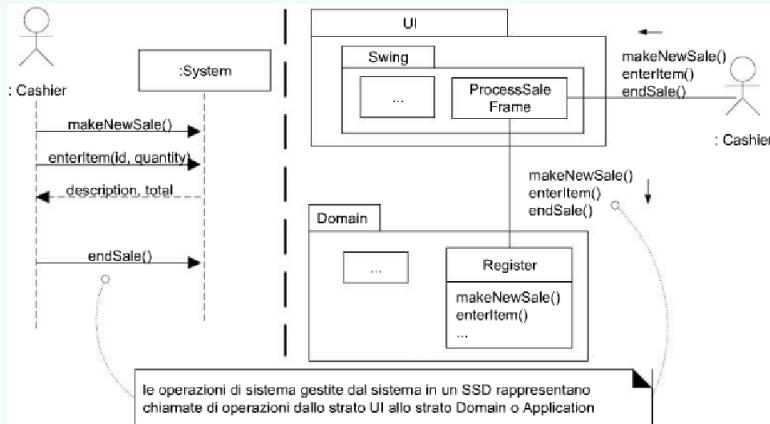
Ulteriori considerazioni:

- ⇒ Le classi di dominio encapsulano le informazioni e il comportamento della logica applicativa;
- ⇒ Le classi della vista sono leggere, sono responsabili dell'input e dell'output, ma non mantengono dati.

Vantaggi:

- ⇒ Favorire la definizione coesa dei modelli;
- ⇒ Consentire lo sviluppo separato;
- ⇒ Minimizzare l'impatto sullo strato del dominio;
- ⇒ Consentire di connettere facilmente nuove viste;
- ⇒ Consentire l'esecuzione dello strato di modello indipendentemente dallo strato UI;
- ⇒ Consentire un *porting* facile dello strato di modello a un altro framework UI.

Esempio 4.3.1 (SSD e UI)



5

Diagrammi di interazione e di classe UML

5.1 Verso la progettazione a oggetti

Domanda 5.1

Come si progetta a oggetti?

⇒ *Codifica*: si progetta mentre si codifica;

⇒ *Disegno, poi codifica*: si disegnano i diagrammi UML e poi si codifica;

⇒ *Solo disegno*: si disegnano i diagrammi UML e lo strumento genera ogni cosa da essi.

Note:-

Soltamente si sceglie di disegnare e poi codificare.

5.1.1 Modellazione dinamica e statica

Definizione 5.1.1: Modelli dinamici

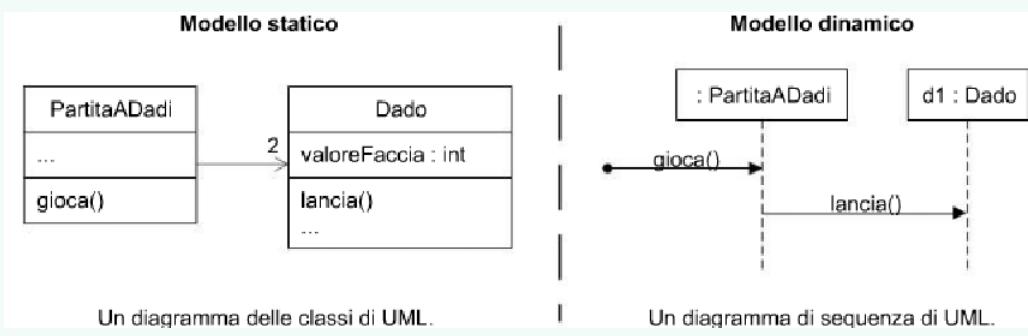
Rappresentano il comportamento del sistema, la collaborazione tra oggetti per realizzare dei Casi d'Uso.
Di solito si utilizzano i diagrammi di interazione UML.

Definizione 5.1.2: Modelli statici

Servono per definire package, nomi delle classi, attributi e firme di operazioni. Di solito si utilizzano i diagrammi delle classi UML.

Note:-

Soltamente si creano in parallelo sia i modelli dinamici che statici.

Esempio 5.1.1 (Modelli statici e dinamici)

- ⇒ La maggior parte del lavoro utile e difficile avviene nel disegno dei diagrammi di interazione (o di sequenza);
- ⇒ Durante la modellazione dinamica si pensa a quali oggetti devono esistere e a come collaborano tra loro;
- ⇒ Per la modellazione dinamica si fa riferimento ai principi GRASP¹.

Le domande che ci si deve porre sono:

- ⇒ *Quali* sono le responsabilità dell'oggetto?
- ⇒ *Chi* collabora con chi?
- ⇒ *Quali* design pattern devono essere applicati?

La progettazione a oggetti richiede la conoscenza di:

- ⇒ *Design pattern*;
- ⇒ *Principi di assegnazione delle responsabilità*.

5.2 Diagrammi di interazione

Definizione 5.2.1: Interazione

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

Note:-

Il termine diagramma di interazione si riferisce a due tipi di diagrammi:

- ⇒ *Diagrammi di sequenza*;
- ⇒ *Diagrammi di comunicazione*.

In questo corso ci concentreremo sui diagrammi di sequenza.

¹Vedi capitolo 6.

Caratteristiche dei diagrammi di sequenza:

- ⇒ Un' *interazione* è motivata dalla necessità di eseguire un *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* del compito;
- ⇒ L'oggetto *responsabile* collabora con altri oggetti (*partecipanti*) per eseguire il compito;
- ⇒ Ciascun *partecipante* svolge un proprio *ruolo* nell'interazione;
- ⇒ La *collaborazione* avviene mediante *messaggi* scambiati tra gli oggetti;
- ⇒ Ciascun messaggio è una richiesta che un oggetto fa a un altro oggetto di eseguire un' *operazione*.

Vantaggi e svantaggi dei diagrammi di sequenza:

- ✓ Mostrano chiaramente la sequenza dell'ordinamento temporale dei messaggi.
- ✗ Costringono a estendersi verso destra quando si aggiungono nuovi oggetti.

Esempio 5.2.1 (Diagrammi di sequenza)



5.2.1 Diagrammi di sequenza del design (DSD)

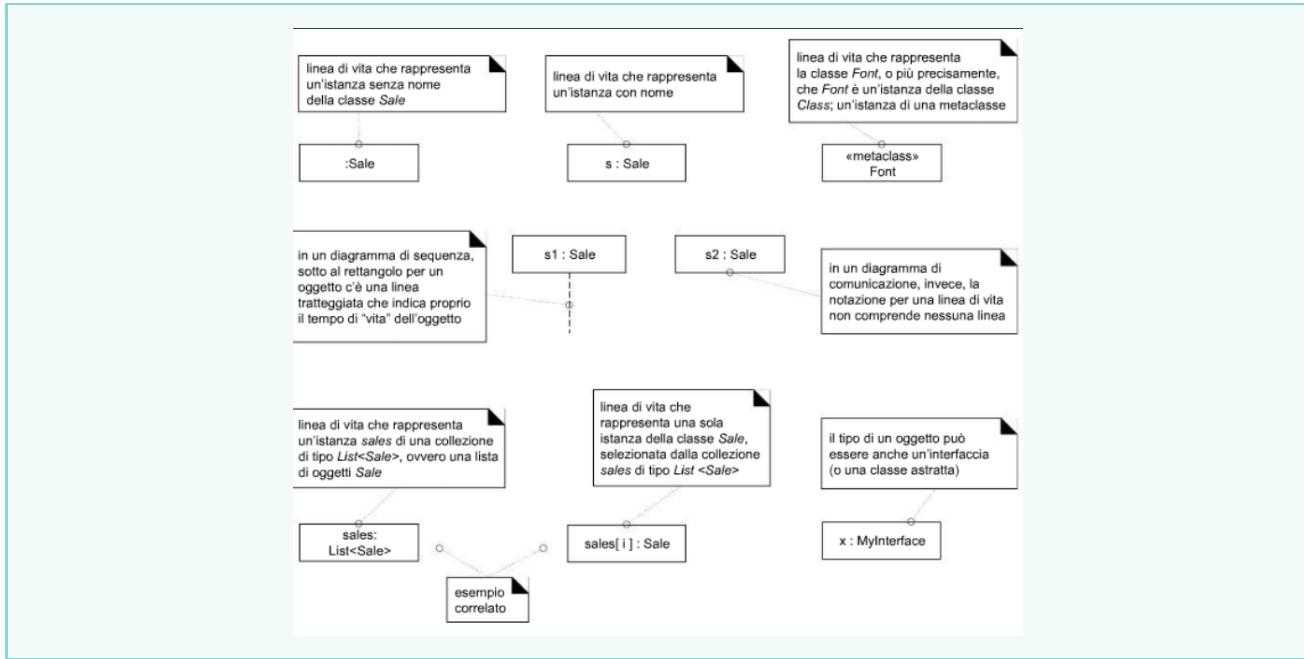
Definizione 5.2.2: Design Sequence Diagram (DSD)

Un *diagramma di sequenza* di progetto è un diagramma di sequenza utilizzato da un punto di vista software o di progetto.

Note:-

In UP l'insieme di tutti i DSD fa parte del modello di progetto.

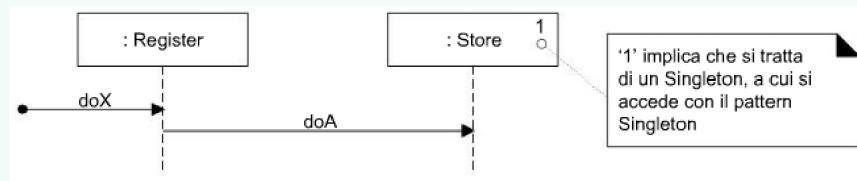
Esempio 5.2.2 (Notazione di DSD: partecipanti)



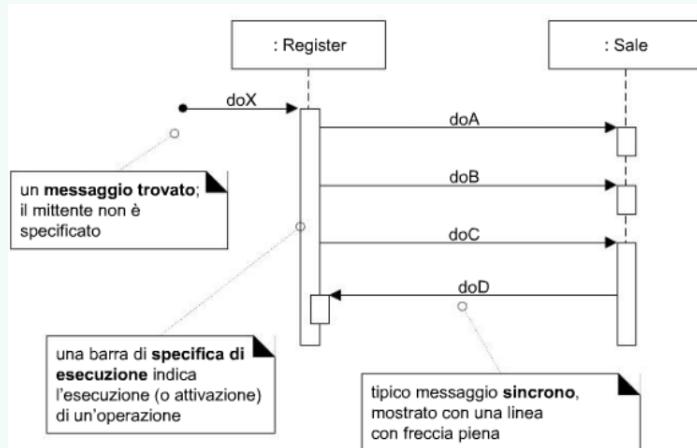
Definizione 5.2.3: Singleton

Un *singleton* è un pattern nel quale da una classe viene istanziata una sola istanza.

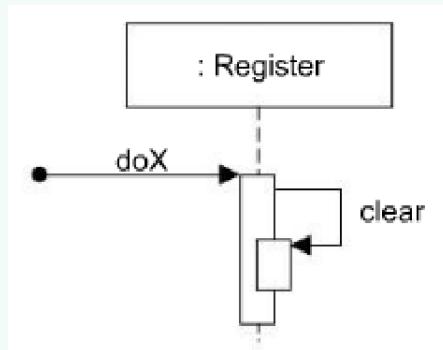
Esempio 5.2.3 (Notazione di DSD: singleton)



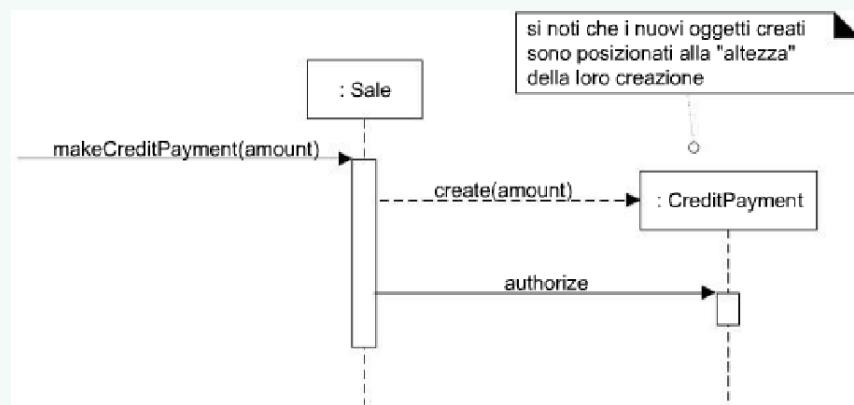
Esempio 5.2.4 (Notazione di DSD: messaggi)



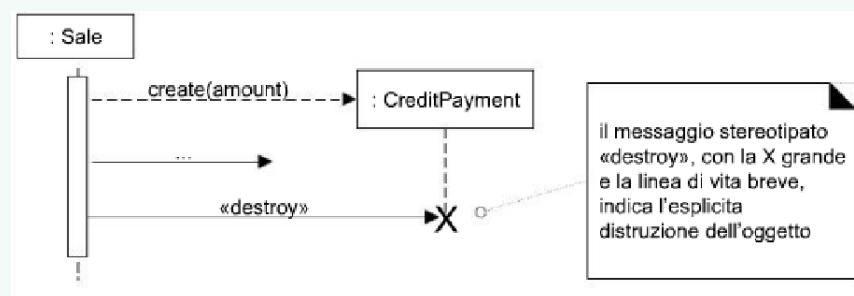
Esempio 5.2.5 (Notazione di DSD: self (this))

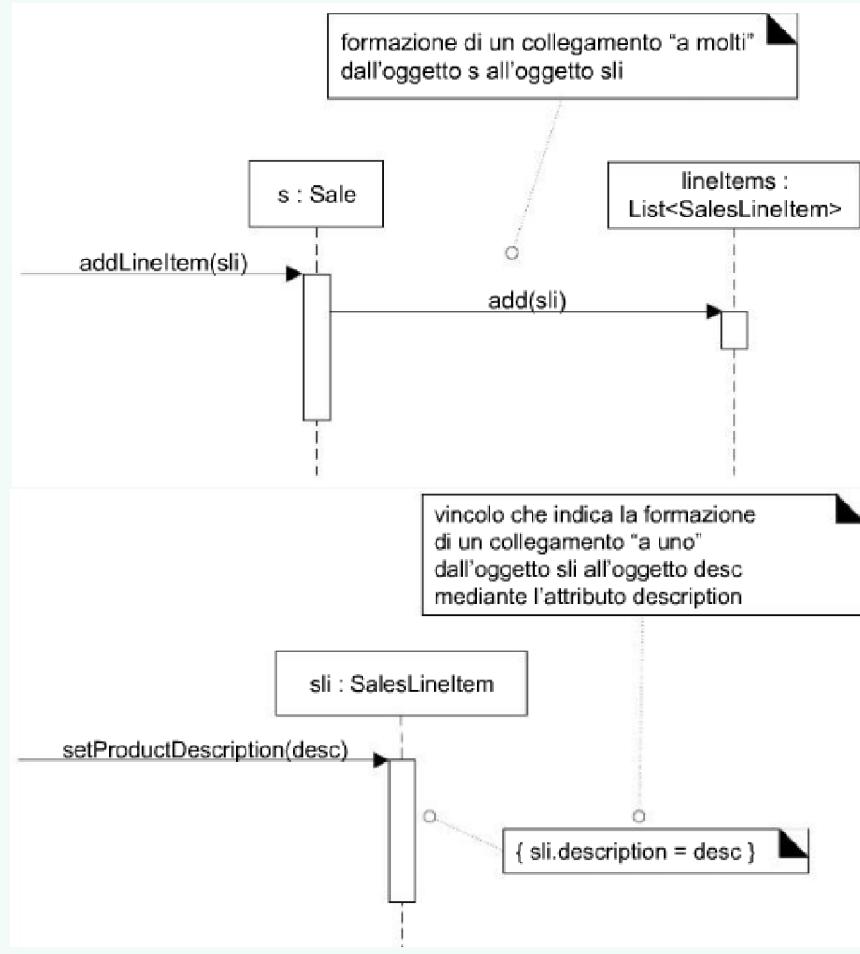
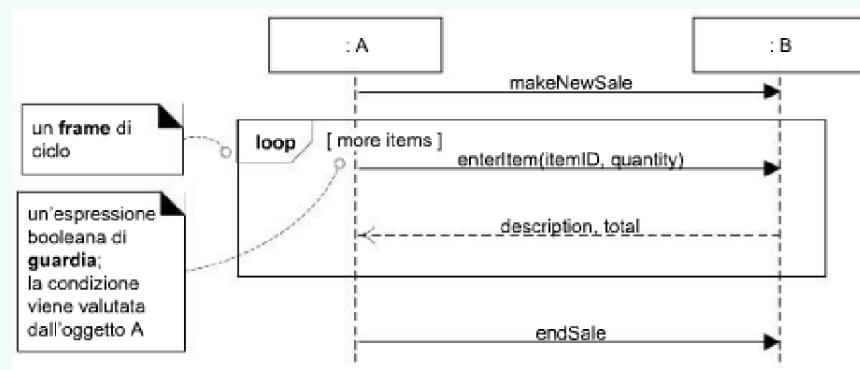


Esempio 5.2.6 (Notazione di DSD: creazione di istanze)

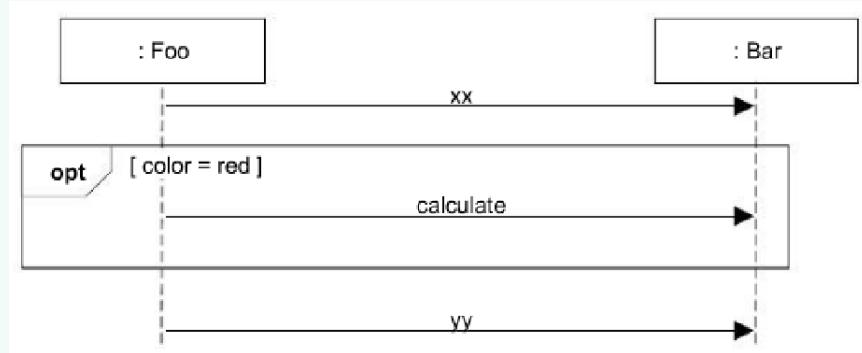


Esempio 5.2.7 (Notazione di DSD: distruzione di oggetti)

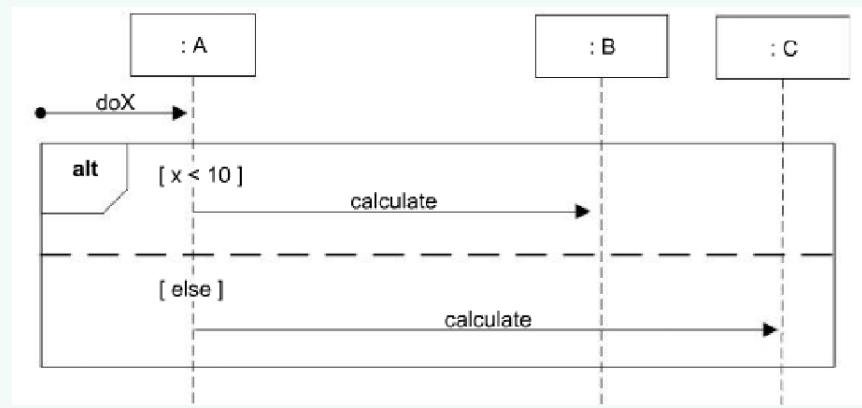


Esempio 5.2.8 (Notazione di DSD: collegamenti)**Esempio 5.2.9** (Notazione di DSD: frame)

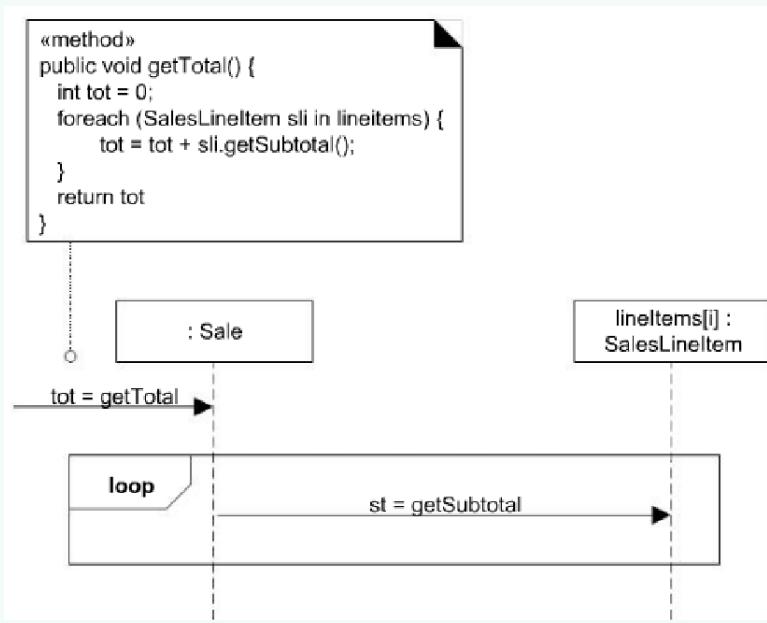
Esempio 5.2.10 (Notazione di DSD: condizionale)

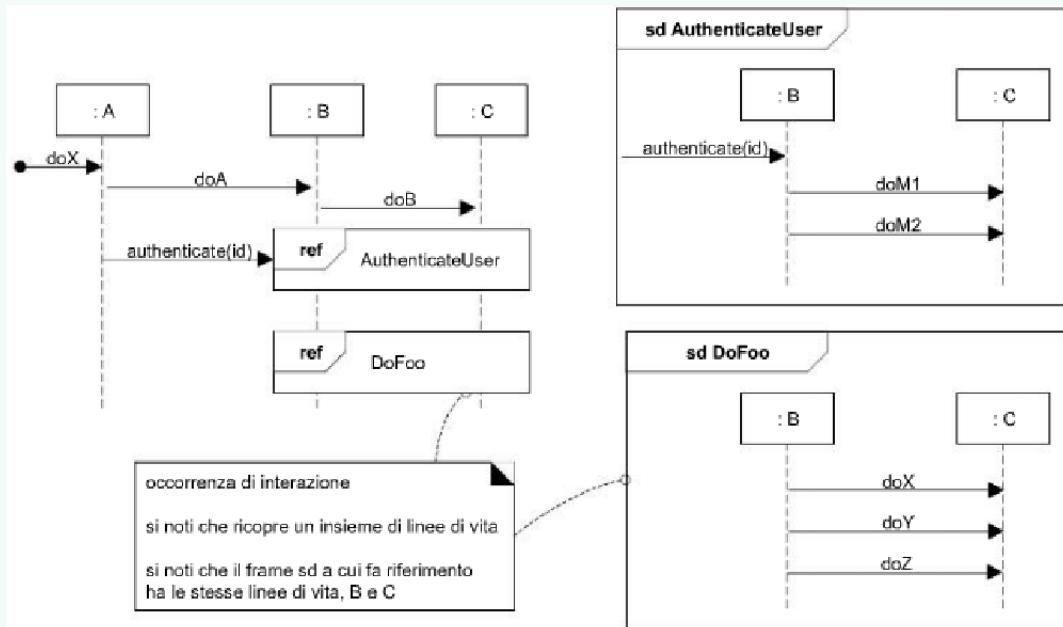
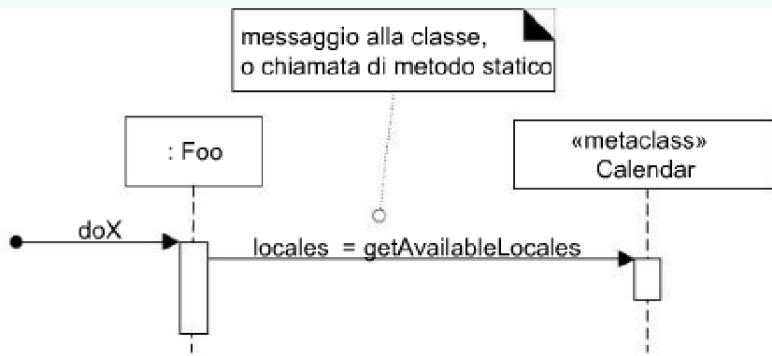


Esempio 5.2.11 (Notazione di DSD: condizionale mutualmente esclusivo)



Esempio 5.2.12 (Notazione di DSD: iterazione su una collezione)



Esempio 5.2.13 (Notazione di DSD: correlare diagrammi di interazione)**Esempio 5.2.14** (Notazione di DSD: invocare metodi statici)**Note:**

Nel caso di invocazione di metodi statici l'oggetto ricevente è un'istanza di una meta-classe.

Esempio 5.2.15 (Notazione di DSD: chiamate sincrone e asincrone)

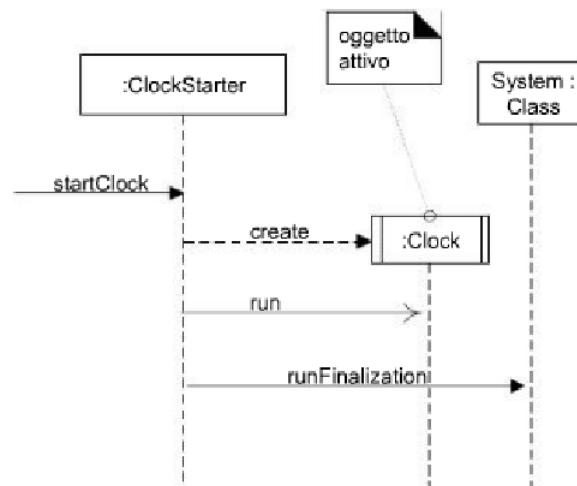
una freccia non piena indica una **chiamata asincrona** (diversamente da una freccia piena, che invece rappresenta una più comune chiamata sincrona)

In Java, per esempio, una chiamata asincrona può avvenire come segue:

```
// Clock implementa l'interfaccia Runnable
Thread t = new Thread( new Clock() );
t.start();
```

la chiamata di *start* implica poi una chiamata asincrona del metodo *run* dell'oggetto *Runnable* (*Clock*)

per semplificare il diagramma, l'oggetto *Thread* e il messaggio *start* si possono evitare (sono un "overhead" standard); al contrario, la chiamata asincrona è implicata dai dettagli essenziali della creazione di *Clock* e dal messaggio *run*



5.3 Diagrammi delle classi

5.3.1 Diagrammi di classe del design (DCD)

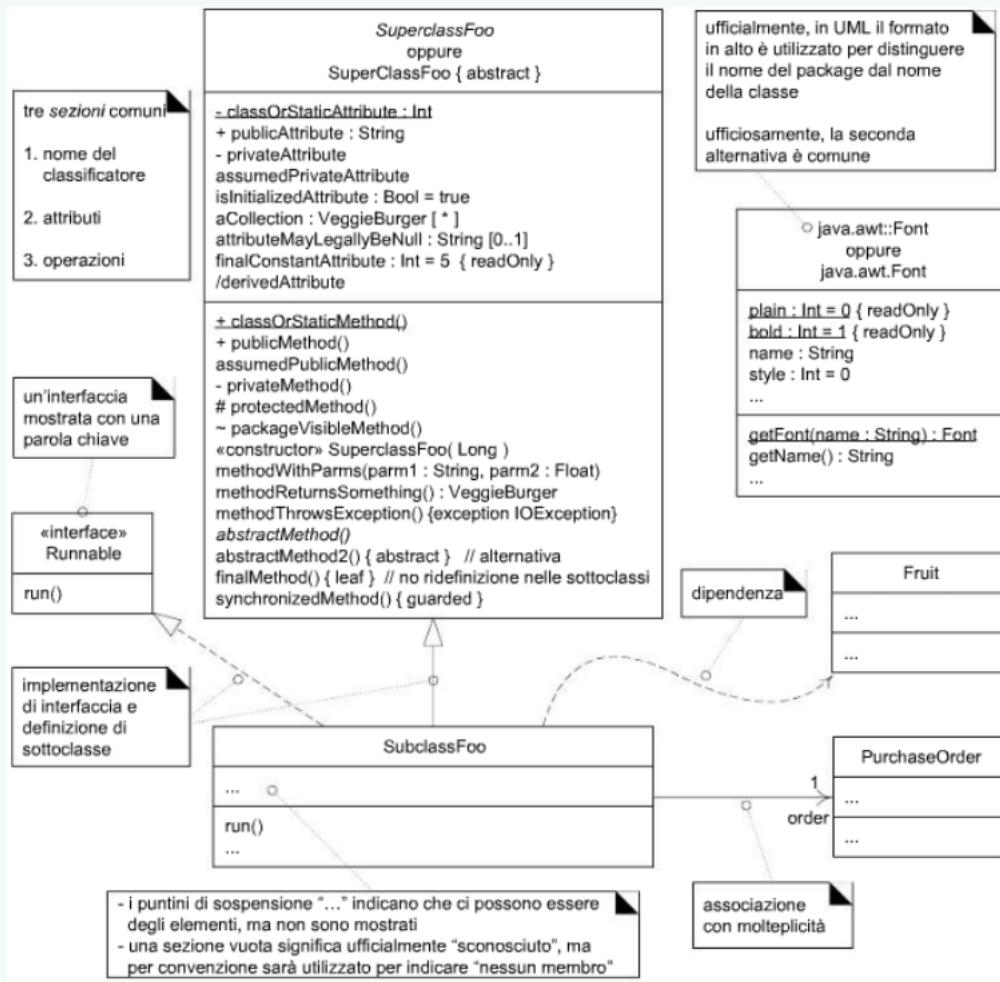
Definizione 5.3.1: Design Class Diagram (DCD)

Un *diagramma delle classi* di progetto è un diagramma delle classi utilizzato da un punto di vista software o di progetto.

Note:-

In UP l'insieme di tutti i DCD fa parte del modello di progetto.

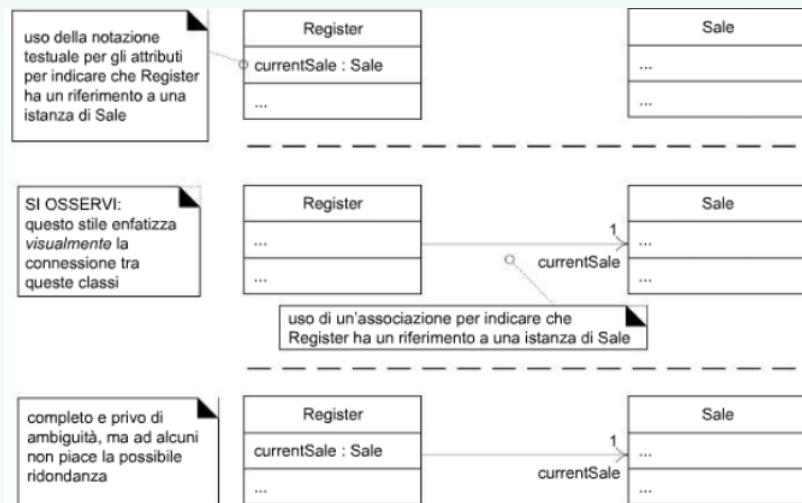
Esempio 5.3.1 (Notazione generale di DCD)



Note:-

- ⇒ + indica un attributo pubblico;
- ⇒ - indica un attributo privato;
- ⇒ # indica un attributo protetto;
- ⇒ ~ indica un attributo package.

Esempio 5.3.2 (Notazione di DCD: attributi)



Note:-

Se non viene specificata la visibilità di un attributo si intende che esso sia privato.

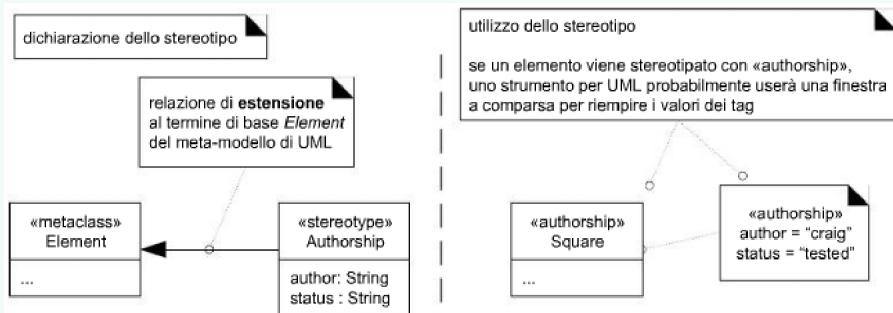
Esempio 5.3.3 (Notazione di DCD: parole chiave)

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

Definizione 5.3.2: Stereotipi

Gli *stereotipi* sono un raffinamento di un concetto di modellazione esistente.

Esempio 5.3.4 (Notazione di DCD: stereotipi)



Definizione 5.3.3: Generalizzazione

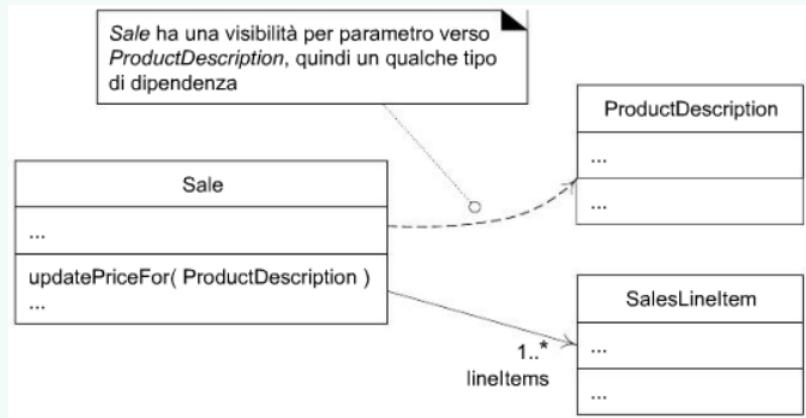
La **generalizzazione** è una relazione tassonomica tra una classe più generale e una classe più specifica. Ogni istanza della classe specifica è anche un'istanza della classe generale.

Note:-

La generalizzazione implica **ereditarietà** nei linguaggi OO.

Definizione 5.3.4: Linee di dipendenza

Una **linea di dipendenza** è una relazione tra due classi che indica che un cambiamento nella classe dipendente può influenzare la classe dipendente.

Esempio 5.3.5 (Notazione di DCD: dipendenza)**Note:-**

È importante mantenere un basso accoppiamento tra le classi.

Esistono varie tipologie di dipendenze:

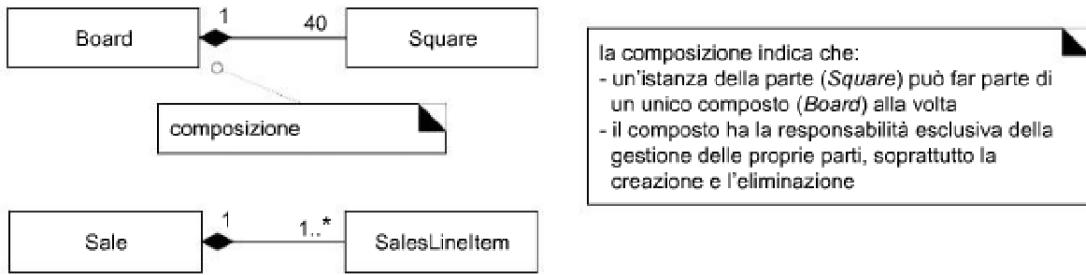
- ⇒ Avere un attributo del tipo del fornitore;
- ⇒ Inviare un messaggio al fornitore;
- ⇒ Ricevere un parametro del tipo del fornitore;
- ⇒ Il fornitore è una superclasse o un'interfaccia implementata.

Definizione 5.3.5: Interfacce

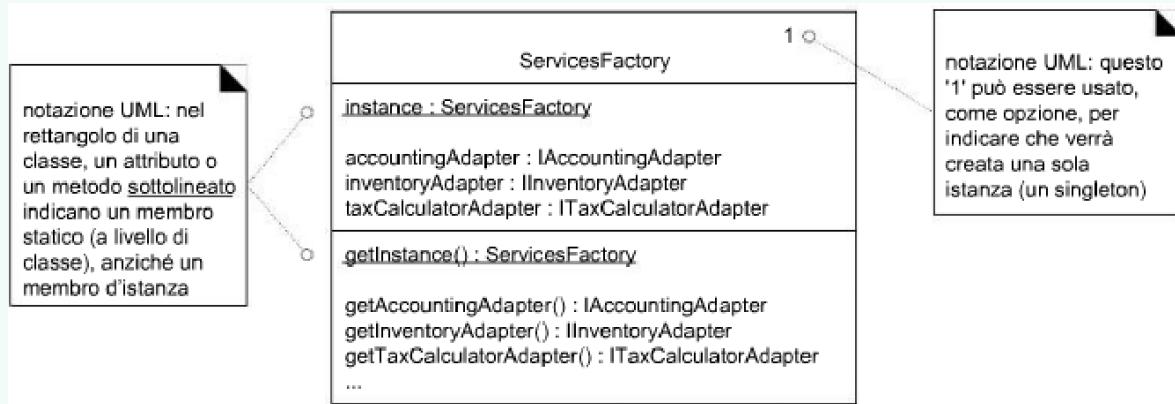
Un'**interfaccia** è un contratto tra un fornitore e un cliente. Un cliente può invocare i metodi definiti nell'interfaccia. Si hanno due notazioni:

- ⇒ **Notazione a pallina** (lollipop), indica che una classe X implementa (fornisce) un'interfaccia Y;
- ⇒ **Notazione a semicerchio** (socket), indica che una classe X richiede (usa) un'interfaccia Y.

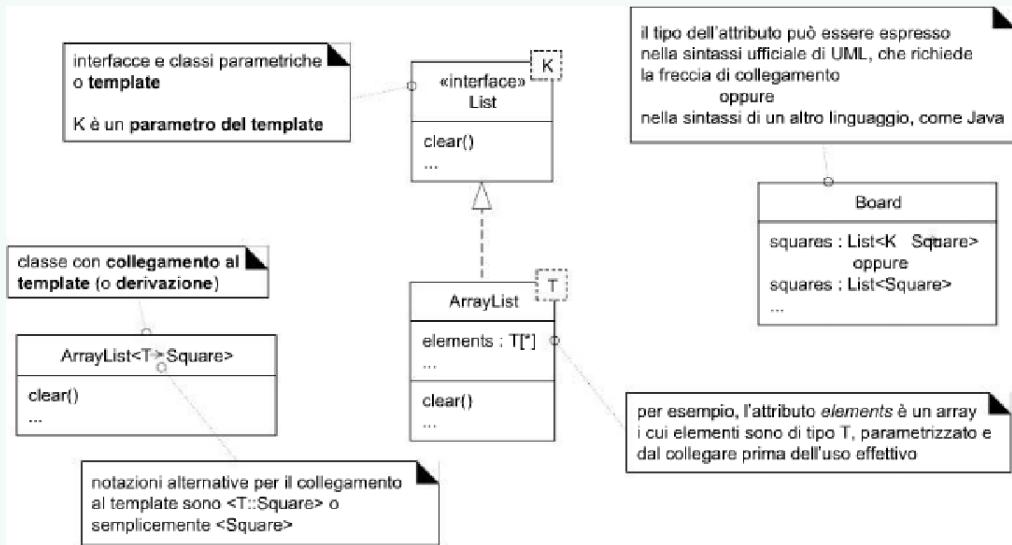
Esempio 5.3.6 (Notazione di DCD: composizione)



Esempio 5.3.7 (Notazione di DCD: singleton)



Esempio 5.3.8 (Notazione di DCD: tipi a template)



6

Pattern GRASP

6.1 Responsability-Driven Development

Definizione 6.1.1: Responsability-Driven Development (RDD)

L'approccio complessivo al fare la modellazione per la programmazione OO si basa sulla metafore della *progettazione guidata dalle responsabilità* (RDD), che è un modo di pensare a come assegnare le responsabilità agli oggetti che collaborano.

Corollario 6.1.1 responsabilità

Per responsabilità si intende un'astrazione di ciò che fa o rappresenta un oggetto o un componente software. Le responsabilità sono di due tipi:

- ⇒ Di fare;
- ⇒ Di conoscere.

Note:-

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

6.1.1 Tipi di responsabilità

Le responsabilità di fare comprendono:

- ✓ un oggetto fa qualcosa per sè stesso;
- ✓ un oggetto chiede ad altri oggetti di fare qualcosa;
- ✓ un oggetto controlla e coordina le attività di altri oggetti.

Le responsabilità di conoscere comprendono:

- ✓ un oggetto conosce i propri dati privati;
- ✓ un oggetto conosce le informazioni di oggetti correlati;
- ✓ un oggetto conosce informazioni che può ricavare o calcolare.

Esempio 6.1.1 (Responsabilità)

Si può dichiarare che la classe *Sale* è responsabile di:

- ⇒ creare oggetti *SaleLineItems* (*responsabilità di fare*);
- ⇒ conoscere il suo totale (*responsabilità di conoscere*).

Note:-

Le responsabilità più grandi coinvolgono centinaia di classi e di metodi, mentre le responsabilità più piccole coinvolgono poche classi e pochi metodi (anche solo un metodo).

La progettazione guidata dalle responsabilità:

1. Identificare le responsabilità, considerandole una alla volta;
2. Decidere a quali oggetti assegnare le responsabilità. Potrebbero essere oggetti già esistenti o nuovi oggetti;
3. Capire come fa un oggetto a soddisfare le proprie responsabilità. Potrebbe farlo da solo o collaborando con altri oggetti.

6.2 General Responsibility Assignment Software Patterns

Definizione 6.2.1: GRASP

GRASP è l'acronimo di General Responsibility Assignment Software Patterns. Si tratta di un insieme di linee guida per la progettazione orientata agli oggetti. I pattern GRASP sono dei principi generali che possono essere applicati in modo flessibile e adattato a seconda del contesto. Sono un aiuto per acquisire padronanza dell'OOD.

Note:-

Sono principi di buona programmazione, ma adeguatamente motivati.

Domanda 6.1

Cosa sono i pattern?

Definizione 6.2.2: Pattern

I principi e gli idiomis se codificati in un linguaggio strutturato, che descrive il *problema* e la *soluzione* e a cui è assegnato un nome, diventano un *pattern*.

Un *pattern* è una coppia problema/soluzione ben conosciuta che può essere applicata in *nuovi contesti* con compromessi, implementazioni, variazioni, etc.

Domanda 6.2

Da dove arrivano i pattern?

- ⇒ La nozione di pattern fu introdotta da Christopher Alexander nei "pattern architettonici";
- ⇒ I pattern software furono introdotti da Kent Beck e sviluppati da Beck con Ward Cunningham;
- ⇒ Nel 1994 viene pubblicato il libro "Design Patterns" di Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. vengono descritti 23 pattern software, noti come i *Gang of Four* (GoF).

Definizione 6.2.3: Low Representational Gap (LGR)

Quando si progetta vale *LRG*: si cerca di ridurre il divario tra il mondo reale e il mondo del software.

Corollario 6.2.1 Progettazione modulare

Comprensibilità, modificabilità, impatto nei cambiamenti basso, flessibilità, riuso, semplicità sono garantiti dal principio di *progettazione modulare*: il software viene diviso in moduli coesi (High Cohesion) e debolmente accoppiati (Low Coupling).

Note:-

I pattern High Cohesion e Low Coupling sarebbero sufficienti per garantire una corretta assegnazione delle responsabilità, ma in pratica sono difficili da applicare direttamente (perchè sono pattern valutativi).

6.3 I pattern GRASP

Dei 9 pattern GRASP si andranno ad analizzare solamente:

- ⇒ Creator;
- ⇒ Information Expert;
- ⇒ Low Coupling;
- ⇒ Controller;
- ⇒ High Cohesion.

6.3.1 Creator

Pattern 6.3.1 (Creator - Creatore):

Problema: Chi crea un oggetto A? Chi deve essere responsabile della creazione di una nuova istanza di una classe?

Soluzione: Assegnare alla classe B la responsabilità di creare un'istanza della classe A se una delle seguenti condizioni è vera:

- ⇒ B **aggredisce o contiene** A;
- ⇒ B **registra** A^a;
- ⇒ B **utilizza** strettamente A;
- ⇒ B ha i dati necessari per **inizializzare** A.

^aRegistrare significa salvare un riferimento.

Note:-

Più condizioni sono vere, meglio è.

Esempio 6.3.1 (Monopoly)**7.21 Esempio: il gioco del Monopoly**

L'unico caso d'uso significativo nel sistema software del Monopoly è *Gioca una Partita a Monopoly* (*Play Monopoly Game*, nelle figure), anche se non supera il test del capo. Poiché la partita viene eseguita come una simulazione che viene semplicemente osservata da una persona, si può dire che questa persona è un osservatore, non un giocatore.

Questo studio di caso mostra che i casi d'uso non sono sempre la soluzione migliore per i requisiti comportamentali. Cercare di descrivere tutte le regole del gioco nella forma di casi d'uso è inopportuno e poco naturale. Di che cosa fanno parte le regole del gioco? Innanzitutto, in generale, si tratta di **regole di dominio** (chiamate talvolta "regole di business"). In UP, le regole di dominio possono far parte delle Specifiche Supplementari (SS). Le SS probabilmente conterranno, in una sezione di "regole di dominio", un riferimento al regolamento ufficiale del gioco o a un sito web che lo descrive. Anche nel testo dei casi d'uso potrebbe esserci un riferimento a questo regolamento, come mostrato di seguito.

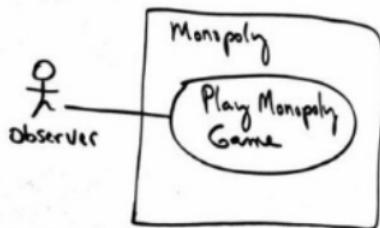


Figura 7.6 Diagramma dei casi d'uso ("diagramma di contesto") per il sistema Monopoly.

Il testo di questo caso d'uso è molto diverso da quello per il sistema POS NextGen, poiché si tratta di una semplice simulazione, e le diverse possibili azioni del giocatore (simulato) sono raccolte nelle regole di dominio anziché nella sezione delle Estensioni.

Caso d'uso UC1: Gioca una Partita a Monopoly (*Play Monopoly Game*)

Portata: Applicazione Monopoly
Livello: Obiettivo utente
Attore primario: Osservatore (Observer, nelle figure)

Parti interessate e interessi:

- Osservatore (Observer): Vuole osservare facilmente il risultato della simulazione della partita.

Scenario principale di successo:

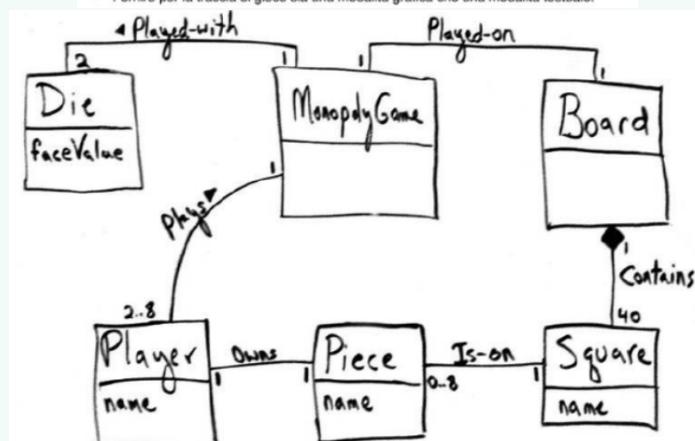
1. L'Osservatore richiede l'inizio di una nuova partita, e inserisce il numero dei giocatori.
 2. L'Osservatore avvia la partita.
 3. Il Sistema visualizza la traccia di gioco per la successiva mossa di un giocatore (vedere regole di dominio, e "traccia di gioco" nel glossario per i dettagli sulla traccia).
- Ripetere il passo 3 fino a che un giocatore vince oppure l'Osservatore annulla la simulazione.*

Estensioni:

- a. In qualsiasi momento, il Sistema fallisce: (per consentire il ripristino, il Sistema registra ogni mossa completata)
 1. L'Osservatore riavvia il Sistema.
 2. Il Sistema rileva il guasto precedente, ricostruisce lo stato ed è pronto a continuare.
 3. L'Osservatore decide di continuare (dall'ultimo turno di gioco completato).

Requisiti speciali:

- Fornire per la traccia di gioco sia una modalità grafica che una modalità testuale.



Osservazioni 6.3.1

- ⇒ Trovare un creatore che abbia effettivamente bisogno di essere collegato a un oggetto (Low Coupling);
- ⇒ Si devono usare classi di supporto (pattern non-GRASP, esempio: Factory Method) se la creazione può essere in alternativa a "riciclo";
- ⇒ Creator è correlato a Low Coupling. Creator favorisce il basso accoppiamento, riduce le dipendenze e favorisce il riuso. Solitamente la classe creata deve essere già visibile a chi la crea.

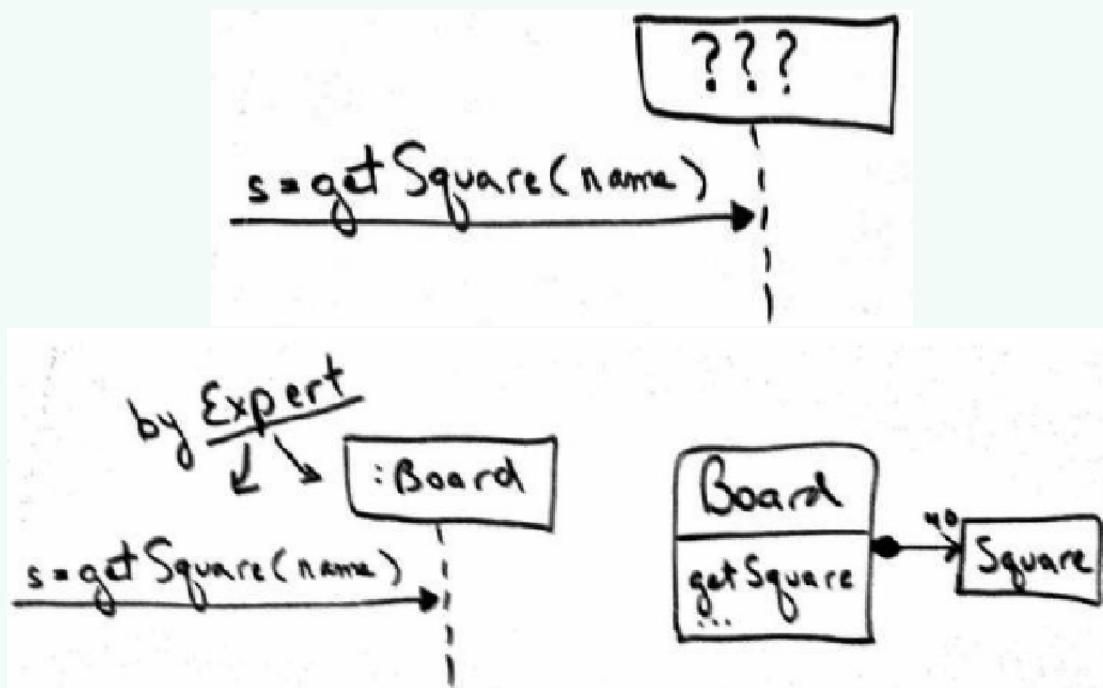
6.3.2 Information Expert**Pattern 6.3.2 (Information Expert - Esperto delle informazioni):**

Problema: Qual è un principio di base, generale, per l'assegnazione delle responsabilità agli oggetti?

Soluzione: Assegnare la responsabilità a un oggetto che ha le informazioni necessarie per soddisfarla.

Esempio 6.3.2 (Monopoly)

Riferendosi all'esempio del Monopoly, si supponga che alcuni oggetti debbano essere in grado di trovare una particolare Square, dato il suo nome unico. Chi deve essere responsabile di conoscere una Square, dato il suo identificatore?

**Osservazioni 6.3.2**

- ⇒ Si individuano informazioni parziali di cui diverse classi sono esperte: queste classi possono collaborare per soddisfare la responsabilità;
- ⇒ Gli oggetti software, a differenza degli oggetti *reali*, hanno la responsabilità di compiere azioni sulle cose che conoscono.

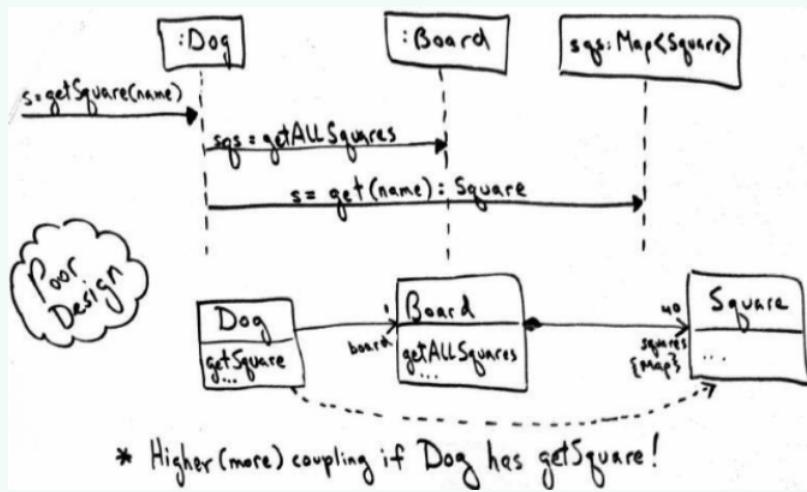
6.3.3 Low Coupling

Pattern 6.3.3 (Low Coupling - Accoppiamento basso):

Problema: Come ridurre l'impatto dei cambiamenti? Come sostenere una dipendenza bassa, un impatto dei cambiamenti basso e una maggiore opportunità di riuso?

Soluzione: Assegnare le responsabilità in modo tale che l'accoppiamento (non necessario) rimanga basso.

Esempio 6.3.3 (Monopoly)



Osservazioni 6.3.3

- ⇒ Low Coupling aumenta la riusabilità e la comprensione;
- ⇒ Low Coupling è un pattern valutativo (come High Cohesion);
- ⇒ Le classi più generiche devono avere un accoppiamento basso;
- ⇒ È normale avere accoppiamento tra le classi, ma bisogna evitare collaborazioni inutili;
- ⇒ Una sottoclasse è fortemente accoppiata alla sua superclasse ^a;
- ⇒ Codice duplicato corrisponde ad alto accoppiamento;
- ⇒ Il problema non è l'accoppiamento alto in sè, ma l'accoppiamento alto con classi instabili.

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

- ⇒ La classe X ha un attributo di tipo Y o referenzia un'istanza di tipo Y o una collezione di oggetti Y;
- ⇒ Un oggetto di tipo X chiama un metodo di un oggetto Y;
- ⇒ Un oggetto di tipo X crea un oggetto di tipo Y;
- ⇒ Il tipo X ha un metodo che contiene un elemento di tipo Y o che referenzia un oggetto di tipo Y;
- ⇒ Una classe X è sottoclasse (diretta o indiretta) di una classe Y;
- ⇒ Y è un'interfaccia implementata da X.

^aSi vedano i pattern GoF, capitolo 7.

6.3.4 High Cohesion

Pattern 6.3.4 (High Cohesion - Coesione alta):

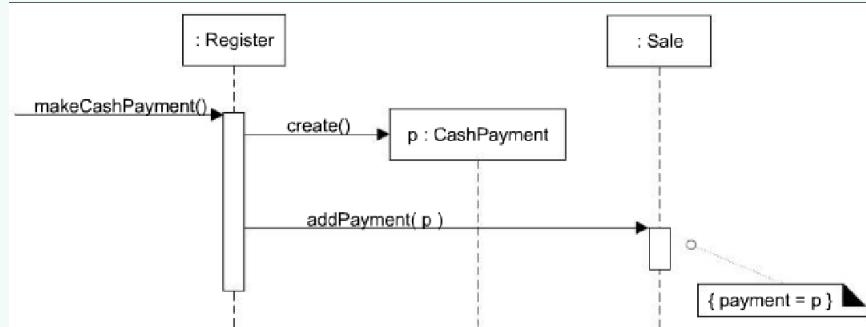
Problema: Come mantenere gli oggetti focalizzati, comprensibili e gestibili e, come effetto collaterale, sostenere Low Coupling?

Soluzione: Assegnare le responsabilità in modo tale che la coesione rimanga alta.

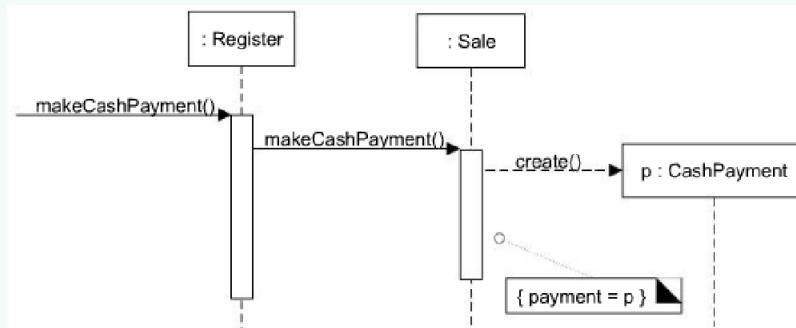
Esempio 6.3.4 (POS NextGen)

Siano CashPayment, Register e Sale tre classi di progetto dell'applicazione POS NextGen.

- ⇒ con il pattern Creator si sceglie Register come creatore di Payment, suggerito dalle responsabilità nel "mondo reale" (registra i pagamenti);
- ⇒ Si usa un metodo addPayment(p) per comunicare con Sale.



Ma questo significa che Register ha una responsabilità di ricevere l'operazione di sistema makeCashPayment e contemporaneamente soddisfarla (anzichè delegarla). Così facendo si andrà a caricare la classe Register di compiti rendendola meno coesa.



In questo modo la responsabilità della creazione del pagamento è delegata a Sale, il che rende favorisce maggiore coesione (e anche minore accoppiamento).

Osservazioni 6.3.4

- ⇒ High Cohesion è un pattern valutativo, come Low Coupling;
- ⇒ Un elemento con responsabilità altamente correlate che non esegue una quantità eccessiva di compiti è altamente coeso;
- ⇒ *Coesione di dati*: una classe implementa un tipo di dati (*molto buona*);

- ⇒ *Coesione funzionale*: gli elementi svolgono una singola funzione (*buona* o *molto buona*);
- ⇒ *Coesione sequenziale*: gli elementi sono raggruppati perchè usati nello stesso tempo (a volte *buona*, a volte *meno molto buona*, dipende dall'utilizzo^a);
- ⇒ *Coesione per pura coincidenza*: per esempio una classe che raggruppa tutti i metodi che iniziano con una certa lettera dell'alfabeto (*molto cattiva*).
- ⇒ High Cohesion rappresenta la coesione funzionale.

Problemi della bassa coesione:

- ⇒ Difficoltà di comprensione;
- ⇒ Difficoltà di manutenzione;
- ⇒ Difficoltà di riuso;
- ⇒ Continuamente cambiante.

In alcuni casi è accettabile avere bassa coesione:

- ⇒ Se per un compito di programmazione/manutenzione ci vuole un esperto;
- ⇒ Per gli oggetti distribuiti lato server.

^aPer esempio il Controller.

6.3.5 Controller

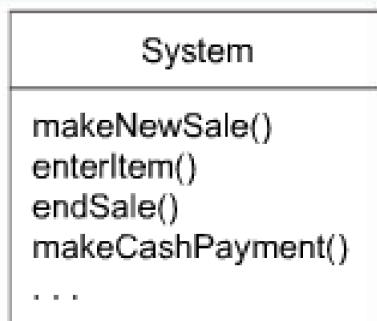
Pattern 6.3.5 (Controller - Controllore):

Problema: Qual è il primo oggetto oltre lo strato UI che riceve e coordina (“controlla”) un’operazione di sistema?

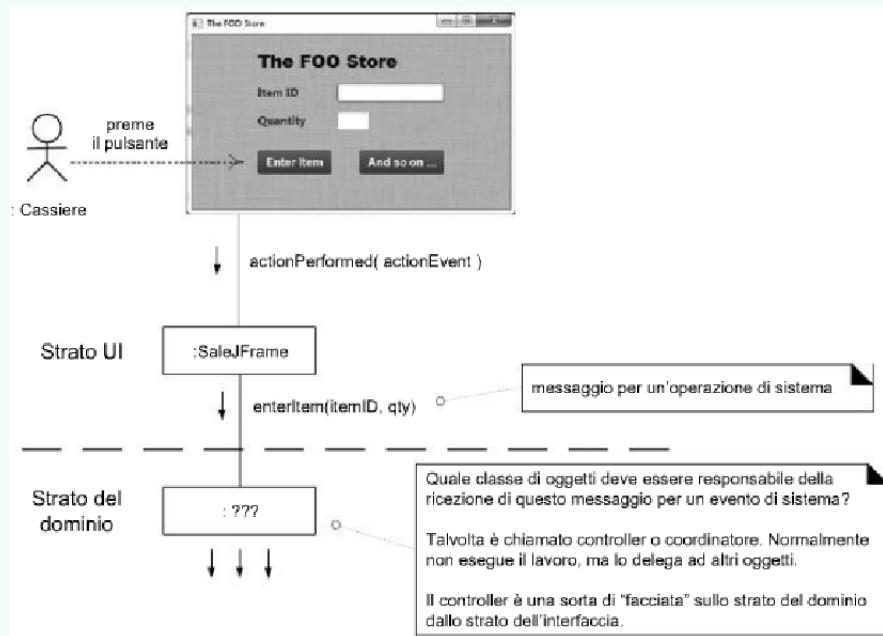
Soluzione: Assegnare le responsabilità a un oggetto che rappresenta una delle seguenti scelte:

- ⇒ Un oggetto che rappresenta il sistema (facede controller);
- ⇒ Un oggetto che rappresenta uno scenario di un Caso d’Uso (controller di Caso d’Uso o controller di sessione).

Esempio 6.3.5 (Controller)

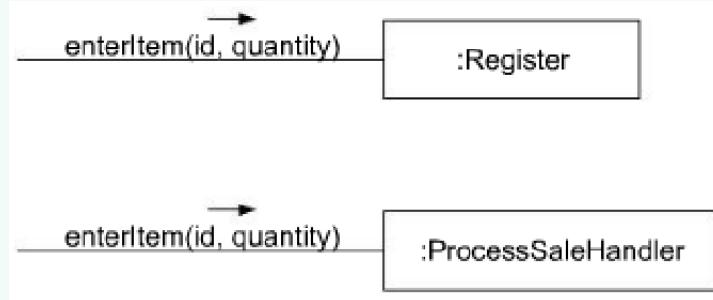


Queste operazioni andranno distribuite dal pattern Controller.



Ci sono possibili soluzioni:

- ⇒ Un oggetto che rappresenta il sistema, per esempio Register;
- ⇒ Un oggetto che rappresenta uno scenario di un Caso d'Uso, per esempio ProcessSaleHandler.



Osservazioni 6.3.5

- ⇒ Il controller è un pattern di delega;
- ⇒ Gli oggetti dell'UI catturano gli eventi e delegano le richieste;
- ⇒ Il controller permette di progettare gli oggetti di dominio in modo indipendente dall'interfaccia utente;
- ⇒ Il controller coordina/controlla le attività, ma non le esegue.

Controller di Caso d'Uso o di sessione:

- ⇒ Si utilizza la classe controller per tutti gli eventi nello stesso scenario;
- ⇒ Una sessione è un'istanza di una conversazione con un attore. Generalmente una sessione corrisponde a un'esecuzione di un Caso d'Uso;
- ⇒ Il controller conserva lo stato della sessione.

Concetto sbagliato 6.1: Il pattern Controller è lo stesso controller del MVC

Il Controller GRASP e il controller MVC sono due cose diverse. Il controller GRASP è un pattern di progettazione orientato agli oggetti, mentre il controller MVC è un componente del pattern architetturale MVC.

MVC:

- ⇒ Fa parte dell'UI e gestisce gli eventi dell'utente;
- ⇒ La sua implementazione dipende dalla tecnologia UI e della piattaforma.

GRASP:

- ⇒ Fa parte del dominio e gestisce le richieste del sistema;
- ⇒ Non dipende dall'UI utilizzata.

Generalmente il controller MVC delega le richieste dell'utente al controller GRASP.

7

Pattern GoF

7.1 Design Pattern GoF

Definizione 7.1.1: Pattern GoF

I *pattern GoF* (Gang of Four) sono un insieme di 23 pattern di progettazione software, descritti nel libro *Design Patterns: Elements of Reusable Object-Oriented Software* di Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (vengono mostrati in C++, analizzando dei progetti di successo e delle soluzioni a problemi ricorrenti). Si dividono in tre categorie: creazionali, strutturali e comportamentali.

Corollario 7.1.1 GRASP e GoF

I *pattern GoF* sono degli "schemi di progettazione avanzata", mentre i *pattern GRASP* sono dei principi di progettazione.

Osservazioni 7.1.1

I pattern GoF preferiscono la composizione all'ereditarietà. Il meccanismo di delega rende la composizione potente quanto l'ereditarietà.

Ereditarietà tra classi:

- ⇒ Si definisce un oggetto in termini di un altro;
- ⇒ Riuso *White-box*, ossia la visibilità della superclasse è la visibilità della sottoclasse;
- ⇒ Definità *staticamente*, non è possibile cambiarla a tempo di esecuzione;
- ⇒ Una modifica alla sopraclassa può avere effetti collaterali sulle sottoclassi (non viene rispettato l'incapsulamento).

Composizione tra oggetti:

- ⇒ Le funzionalità sono ottenute assemblando e componendo gli oggetti per avere funzionalità più complesse;
- ⇒ Riuso *Black-box*, ossia i dettagli interni non sono conosciuti;

- ⇒ Se una classe ne usa un'altra questa può essere referenziata come un'interfaccia, quindi è possibile cambiare l'implementazione a tempo di esecuzione;
- ⇒ Utilizzando un'interfaccia si rispetta l'incapsulamento per cui solo una modifica all'interfaccia può avere effetti collaterali.

Note:-

In questo corso se ne esamineranno solo 9.

Definizione 7.1.2: Polimorfismo

Le sottoclassi possono essere castate in base al tipo, nascondendo alcuni dettagli al cliente.

Definizione 7.1.3: Specializzazione

Le sottoclassi guadagnano funzionalità aggiuntive rispetto alla superclasse.

Note:-

La dicitura <<stereotype>> (o <<stereotype>>) indica che non si fa riferimento a nomi di classi. Si tratta di metaclassi.

7.2 Pattern Creazionali

Definizione 7.2.1: Pattern Creazionali

I *pattern creazionali* risolvono problematiche inerenti l'istanziamento degli oggetti.

7.2.1 Abstract Factory

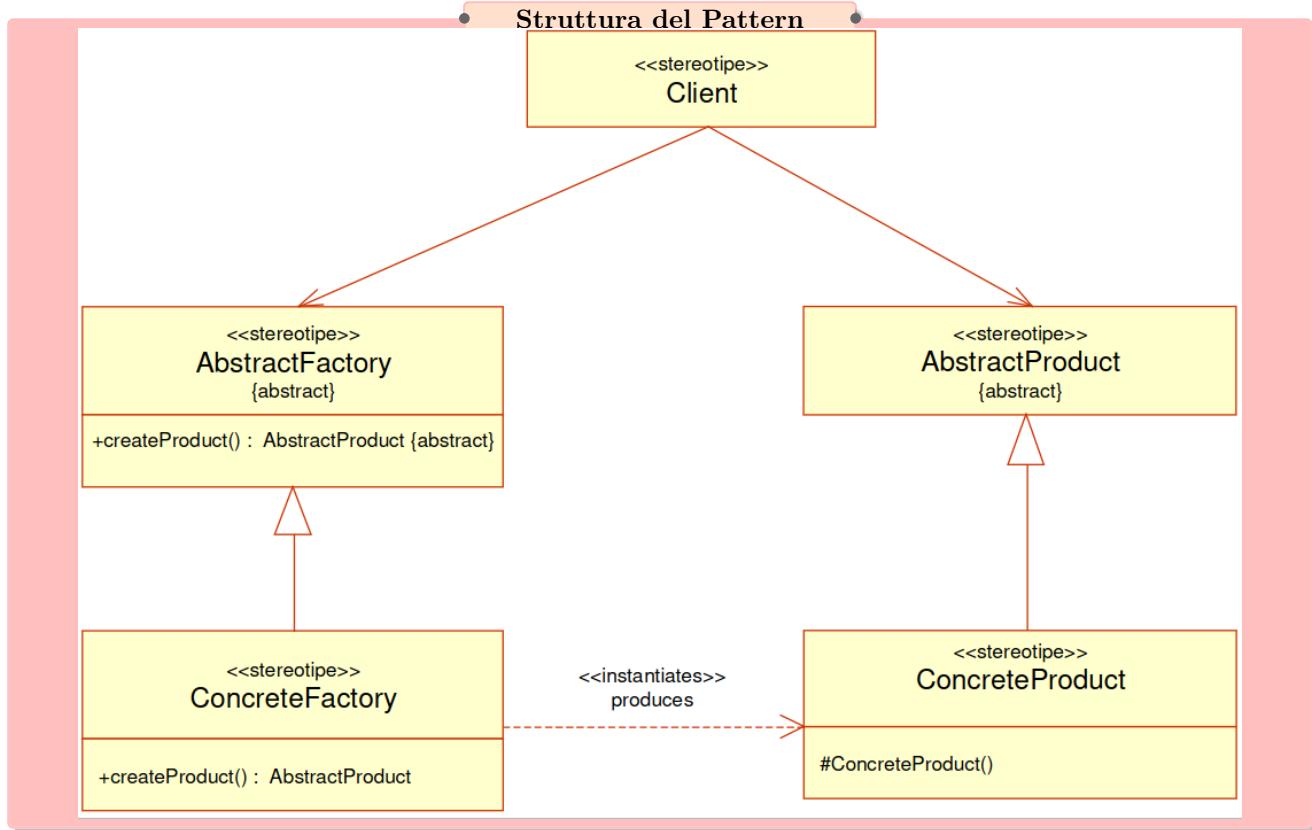
Pattern 7.2.1 (*Abstract Factory*):

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

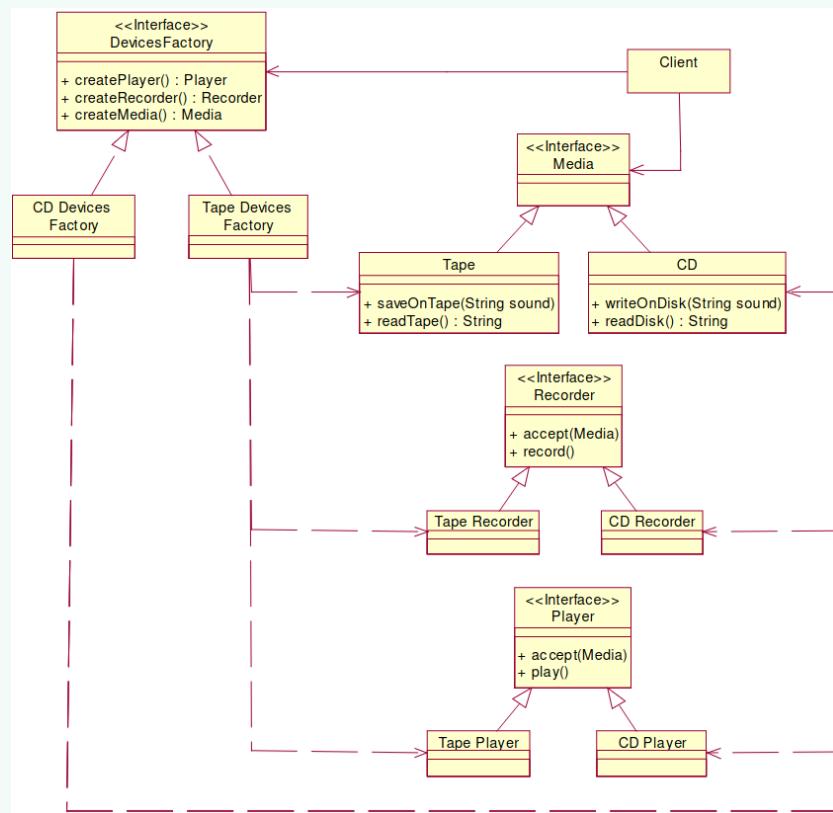
Soluzione: Definire un'interfaccia **Factory** (astratta). Definire una classe **Factory** concreta per ogni famiglia di elementi da creare. Opzionalmente si può definire una classe astratta che implementi l'interfaccia factory e fornisce servizi comuni alle factory che la estendono.

Osservazioni 7.2.1

- ⇒ Presenta un'interfaccia per la creazione di famiglie di prodotti;
- ⇒ Il cliente può creare prodotti vincolati tra loro;
- ⇒ È possibile cambiare la famiglia di prodotti senza modificare il codice del cliente;
- ⇒ È possibile creare una classe AbstractFactory a cui si accede tramite un Singleton;
- ⇒ È usato nelle librerie Java per la creazione di famiglie GUI per diversi SO o sottosistemi GUI.



Esempio 7.2.1 (Applicazione di Abstract Factory)



7.2.2 Singleton

Pattern 7.2.2 (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: Si definisce un metodo statico della classe che restituisce l’istanza della classe stessa (Singleton).

Osservazioni 7.2.2

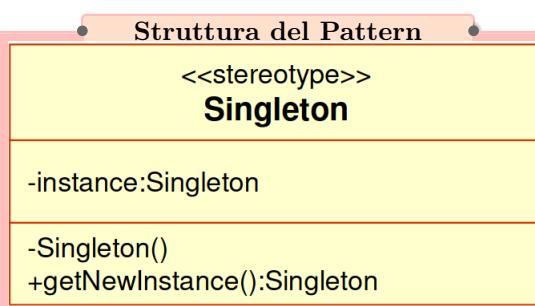
- ⇒ Il Singleton definisce una classe della quale è possibile l’istanziazione di un unico oggetto tramite un metodo statico incaricato di restituire l’istanza;
- ⇒ Le diverse richieste di istanziazione restituiscono un riferimento allo stesso oggetto;
- ⇒ In UML viene illustrato con un "1" nella sezione del nome.

In Java ci sono tre possibili implementazioni:

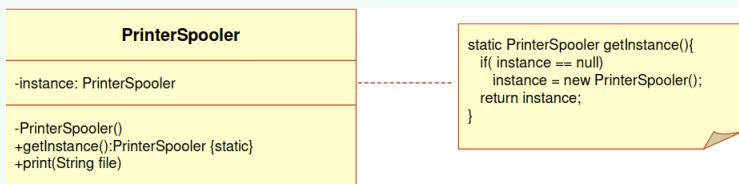
- ⇒ **Come classe statica:** non è un vero Singleton, perchè si lavora con la classe e non con l’oggetto;
- ⇒ **Creato da un metodo statico:** la prima chiamata alla classe crea l’oggetto, le successive restituiscono l’oggetto creato (inizializzazione *pigra*);
- ⇒ **Multi-thread:** versione a multi-thread della precedente.

Il Singleton creato da un metodo statico è preferibile al Singleton come classe statica perchè:

- ⇒ I metodi d’istanza consentono la ridefinizione nelle sottoclassi e il raffinamento;
- ⇒ 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.



Esempio 7.2.2 (Applicazione di Singleton)



7.3 Pattern Strutturali

Definizione 7.3.1: Pattern Strutturali

I *pattern strutturali* risolvono problematiche inerenti la struttura delle classi e degli oggetti.

7.3.1 Adapter

Pattern 7.3.1 (Adapter):

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

Soluzione: Si converte l'interfaccia originale di un componente in un'altra interfaccia attraverso un adattatore intermedio.

Note:-

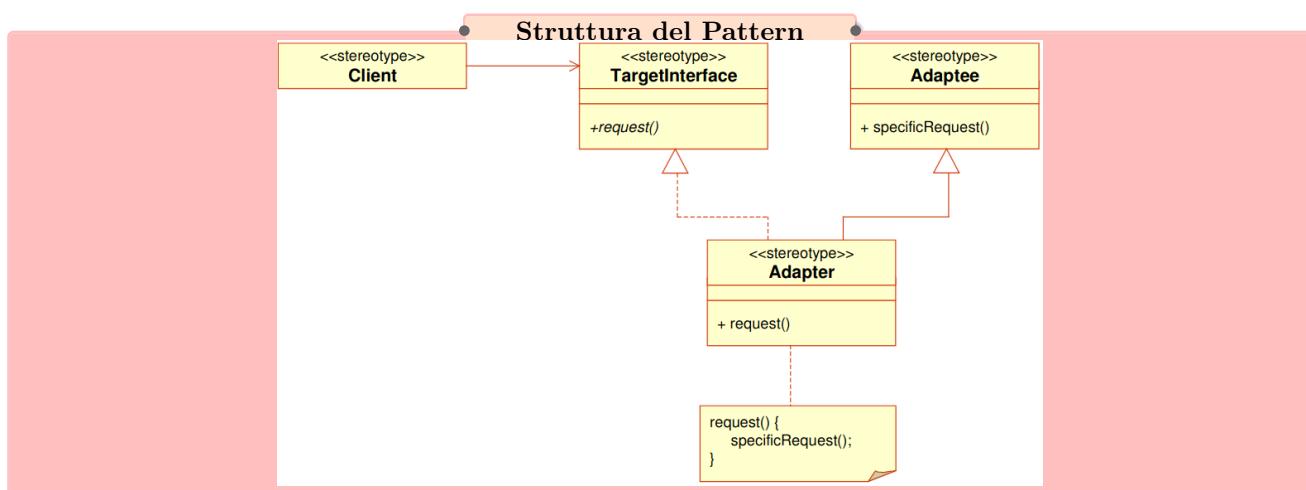
Adapter viene spesso usato per aumentare la riusabilità del software.

Osservazioni 7.3.1

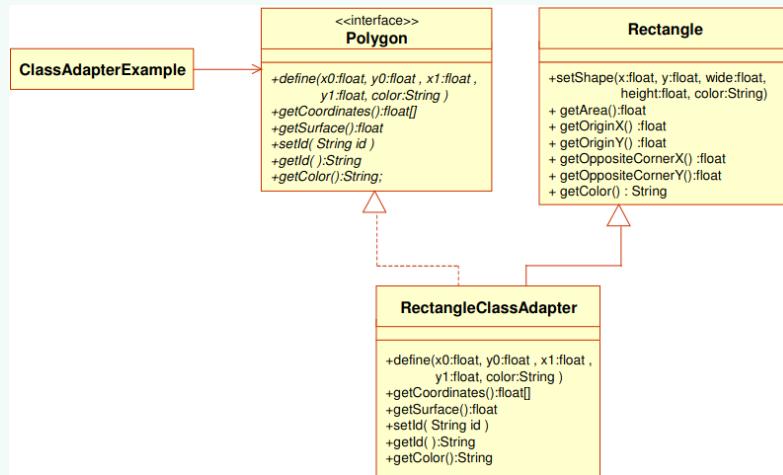
- ⇒ In una coppia client-server le interfacce sono *incompatibili* quando l'oggetto server offre servizi di interesse per l'oggetto client, ma l'oggetto client vuole fruire di questi servizi in una modalità diversa da quella offerta dall'oggetto server;
- ⇒ Ci sono più oggetti server che offrono servizi simili, che hanno interfacce simili, ma diverse. Un oggetto client vuole fruire dei servizi di uno di questi oggetti server (componenti simili, ma con interfacce diverse).

Generalmente:

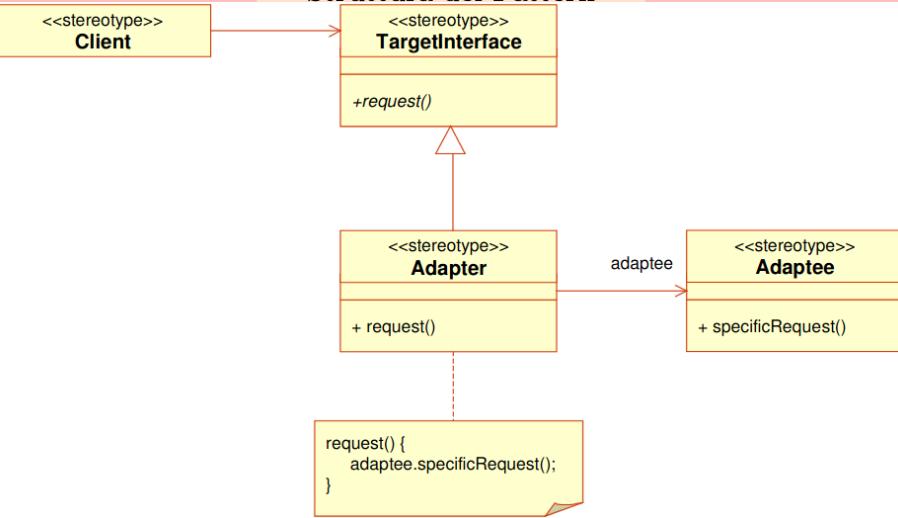
1. Un adattatore riceve una richiesta da un client (nel formato del client);
2. L'adattatore converte la richiesta nel formato del server;
3. L'adattatore inoltra la richiesta al server;
4. Se il server fornisce una risposta lo fa nel suo formato;
5. L'adattatore converte la risposta nel formato del client e la restituisce al client.



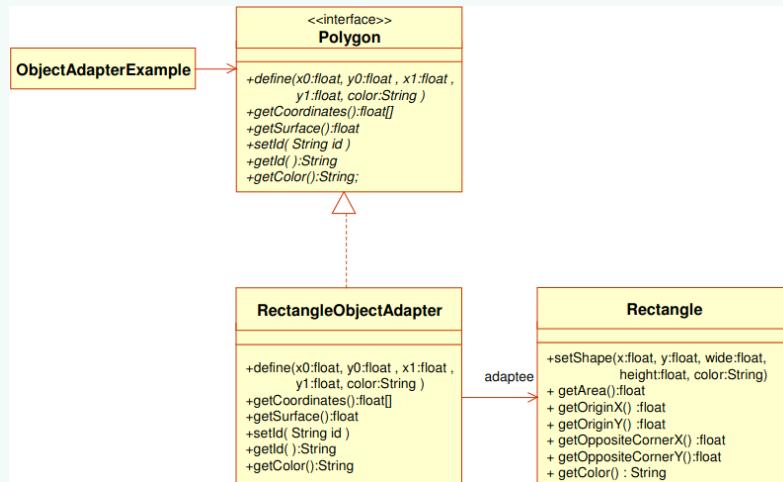
Esempio 7.3.1 (Applicazione di Adapter: classe)



Struttura del Pattern



Esempio 7.3.2 (Applicazione di Adapter: oggetto)



7.3.2 Composite

Pattern 7.3.2 (Composite):

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

Soluzione: Si definiscono le classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia.

Osservazioni 7.3.2

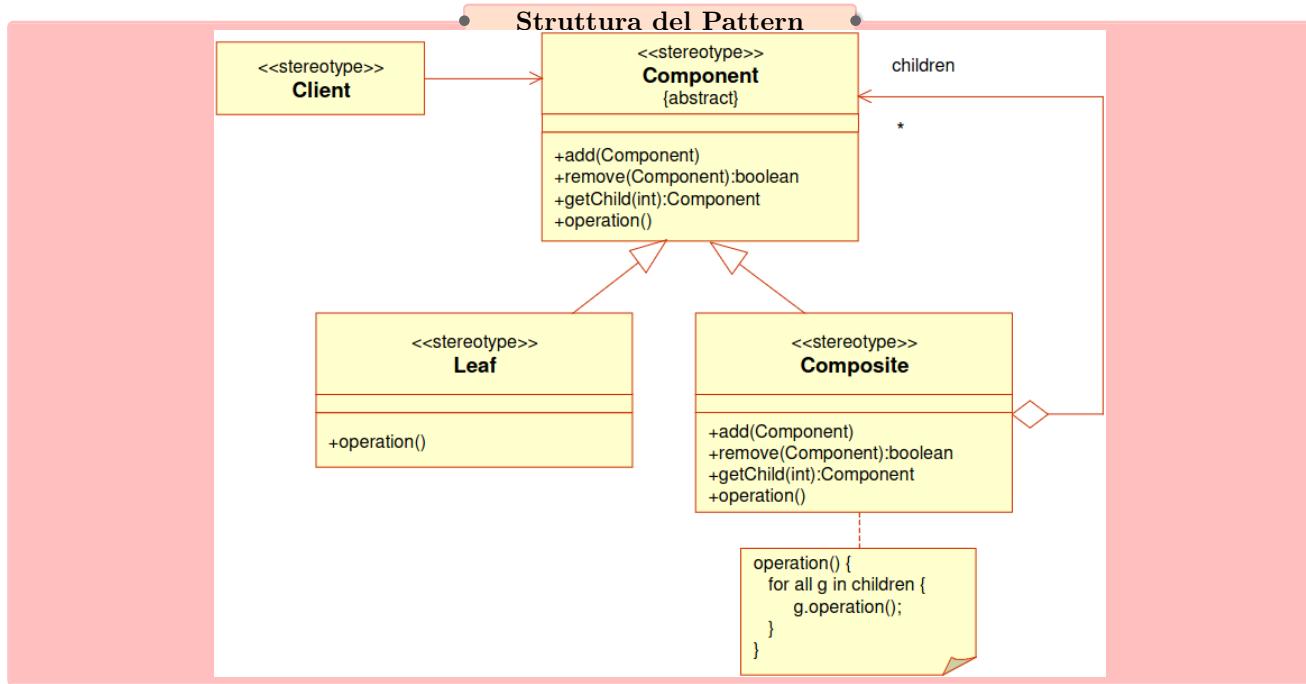
- ⇒ Consente la costruzione di gerarchie di oggetti composti;
- ⇒ È anche noto come *struttura ad albero*, *composizione ricorsiva* o *struttura induttiva*, in cui foglie e nodi hanno le stesse funzionalità.

È utile per:

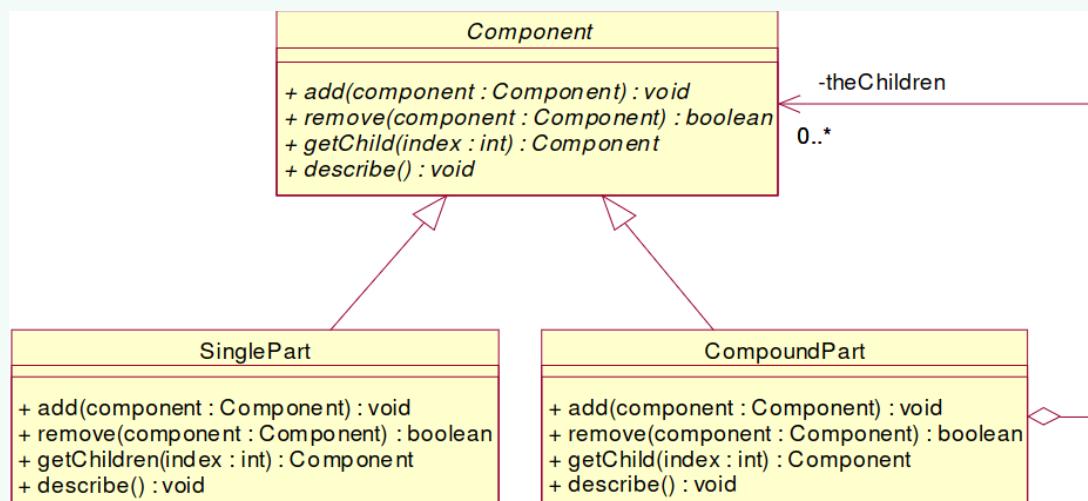
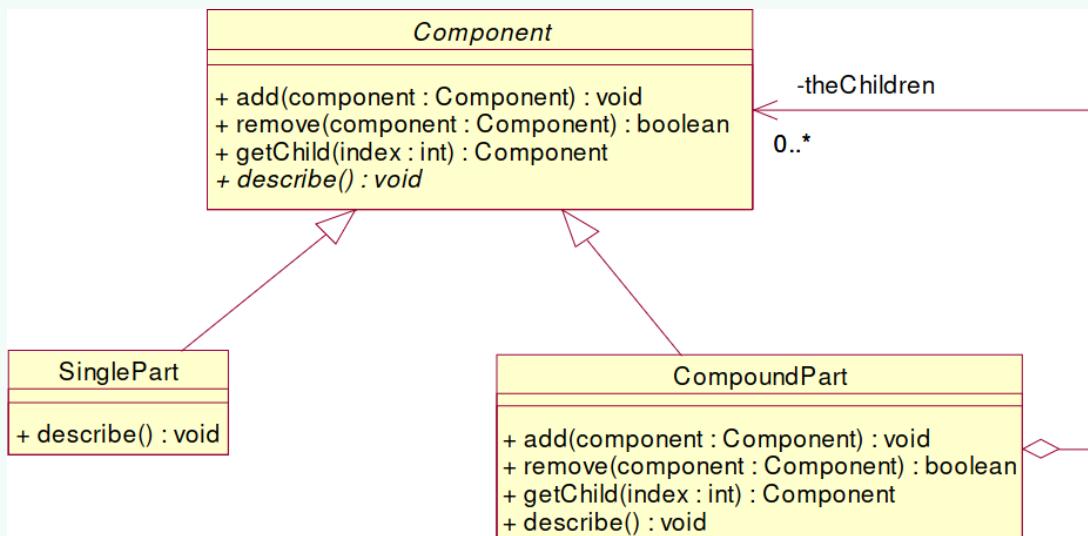
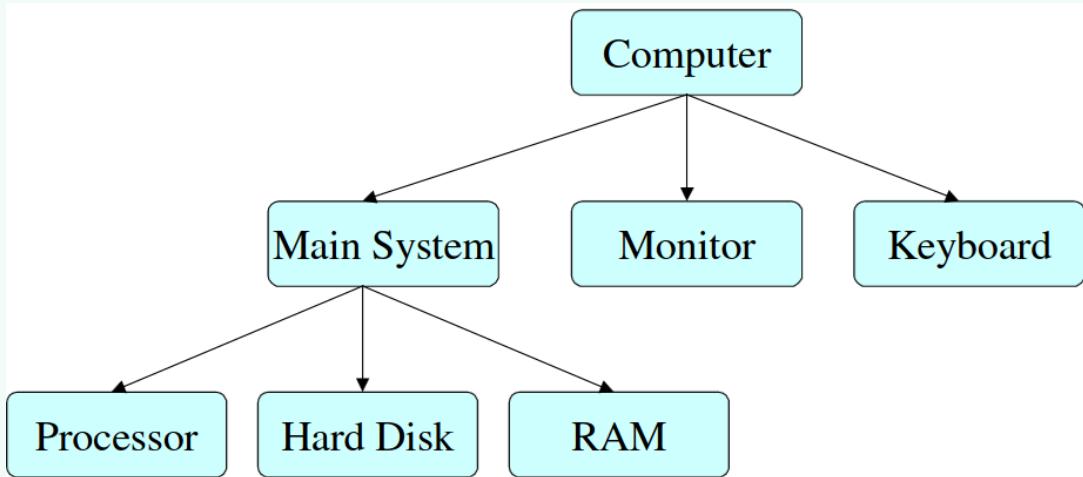
- ⇒ Creare gerarchie *tutto-parte*;
- ⇒ Ignorare le differenze tra oggetti atomici e composti;
- ⇒ Implementare la stessa interfaccia per tutti gli elementi contenuti.

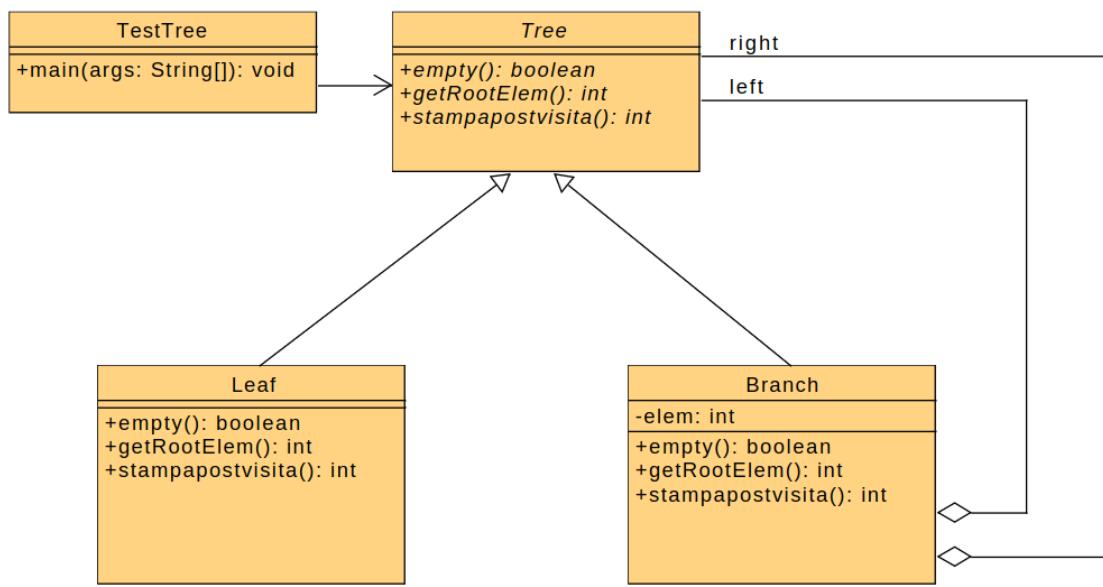
Il pattern definisce una classe astratta **componente** che deve essere estesa in due sottoclassi:

- ⇒ *Foglia*: rappresenta un oggetto atomico;
- ⇒ *Nodo*: rappresenta un oggetto composto e si implementa come contenitore di componenti.



Esempio 7.3.3 (Applicazione di Composite)



Esempio 7.3.4 (Applicazione di composite: tree)**7.3.3 Decorator****Pattern 7.3.3 (Decorator):**

Problema: Come permettere l'assegnamento di una o più responsabilità a 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?

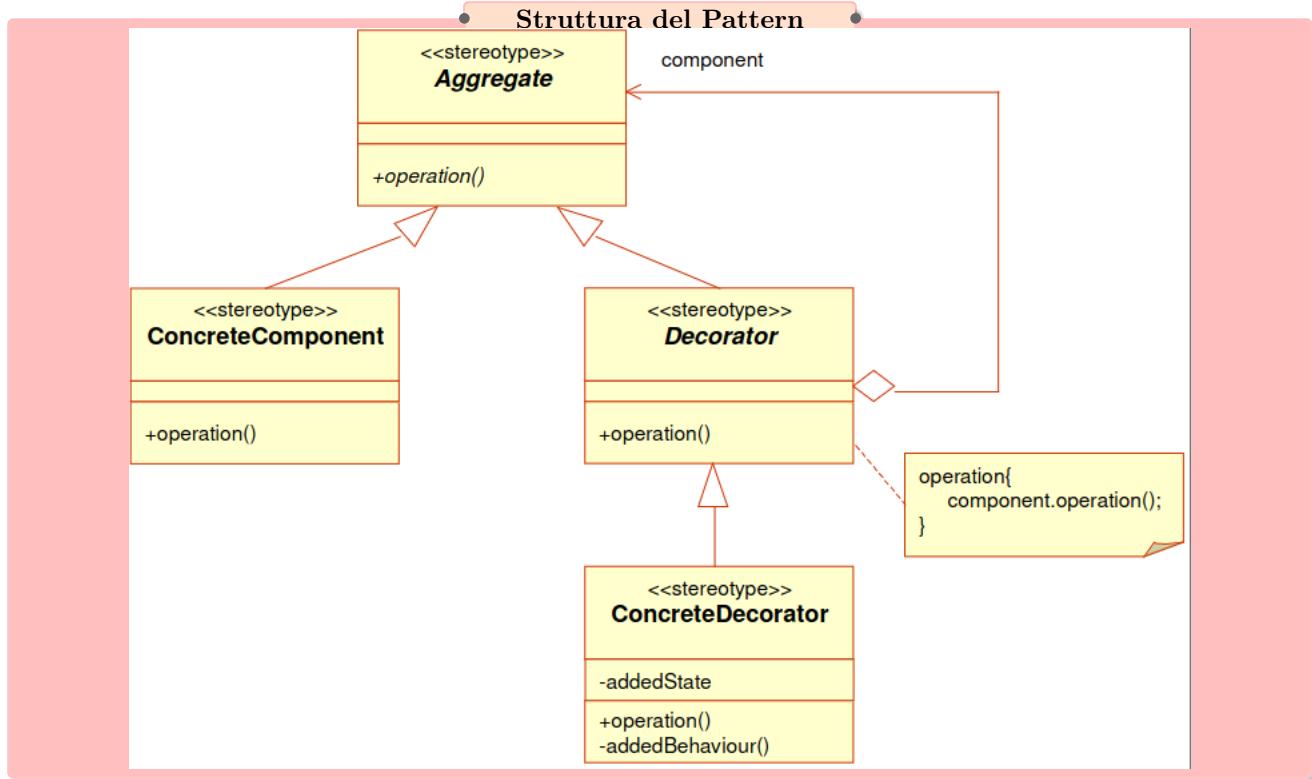
Soluzione: Si ingloba l'oggetto all'interno di un altro che aggiunge le nuove funzionalità.

Note:-

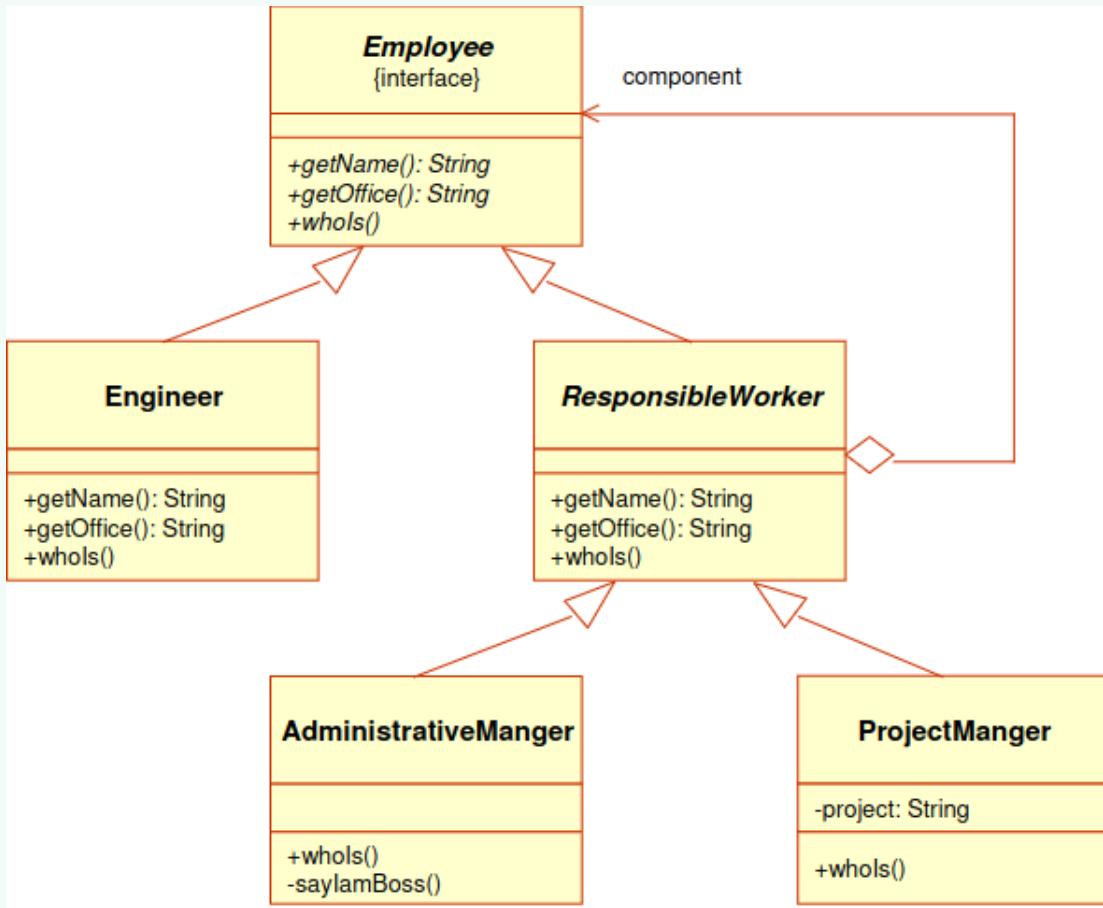
Riduce l'uso di extends, di fatto "simulandolo".

Osservazioni 7.3.3

- ⇒ Il Decorator è anche chiamato *Wrapper*;
- ⇒ È usato per aggiungere responsabilità a oggetti individualmente, dinamicamente e in maniera trasparente (senza impatti su altri oggetti);
- ⇒ Le responsabilità possono essere *ritirate*;
- ⇒ Evita l'*esplosione delle sottoclassi* (evita la creazione di una classe per ogni combinazione di responsabilità).



Esempio 7.3.5 (Applicazione di Decorator)



7.4 Pattern Comportamentali

Definizione 7.4.1: Pattern Comportamentali

I *pattern comportamentali* risolvono problematiche inerenti riguardanti l'interazione tra oggetti.

7.4.1 Observer

Pattern 7.4.1 (*Observer*):

Problema: Diversi tipi di oggetti *subscriber* (abbonato) sono interessati ai cambiamenti di stato o agli eventi di un oggetto *publisher* (editore). Ciascun subscriber vuole reagire in modo suo quando il publisher genera un evento. Inoltre il publisher vuole mantenere un accoppiamento basso (vedi 6.3.3) verso i suoi subscriber. Cosa si deve fare?

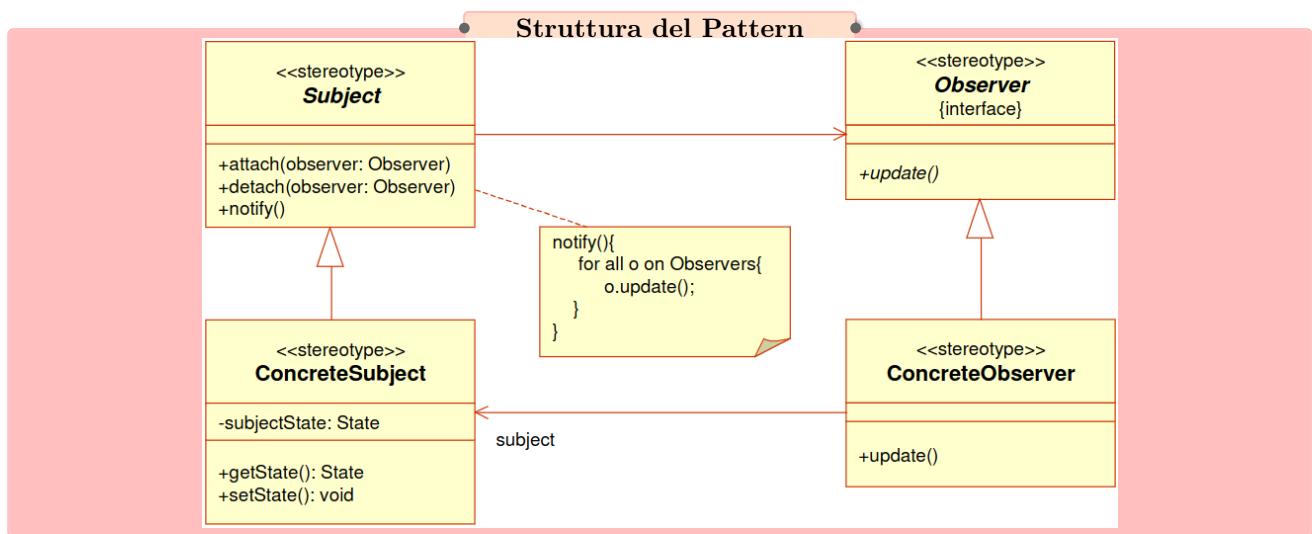
Soluzione: Si definisce un'interfaccia *subscriber* o *listener*. Gli oggetti *subscriber* implementano questa interfaccia. Il publisher registra dinamicamente i *subscriber* che sono interessati ai suoi eventi e li notifica quando avvengono.

Note:-

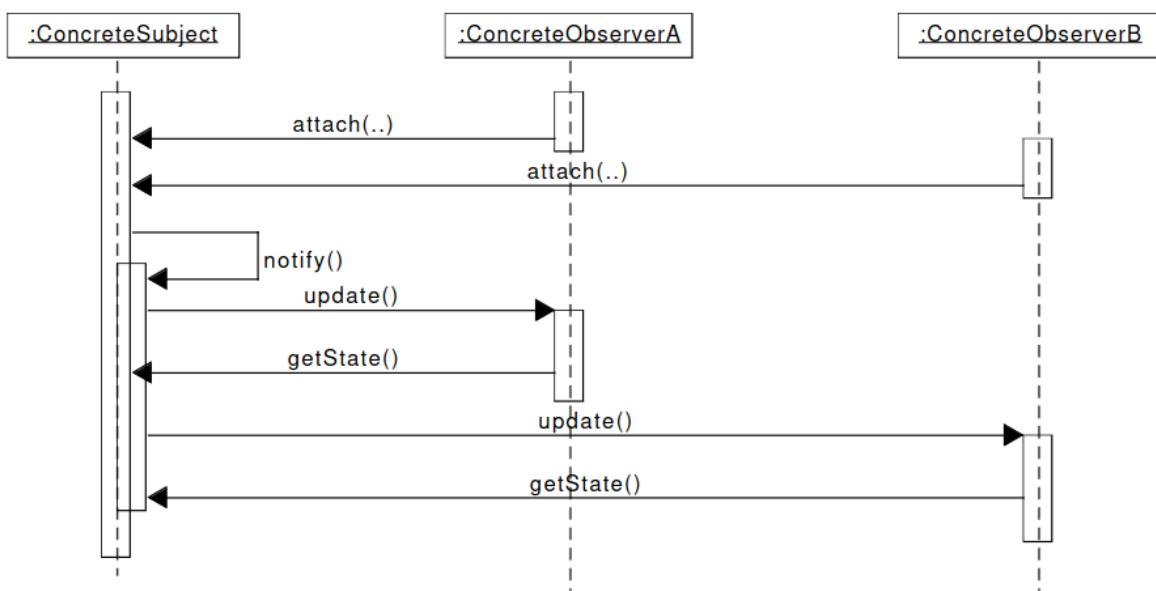
Questo pattern è anche chiamato *publish-subscribe* o *dependents*.

Osservazioni 7.4.1

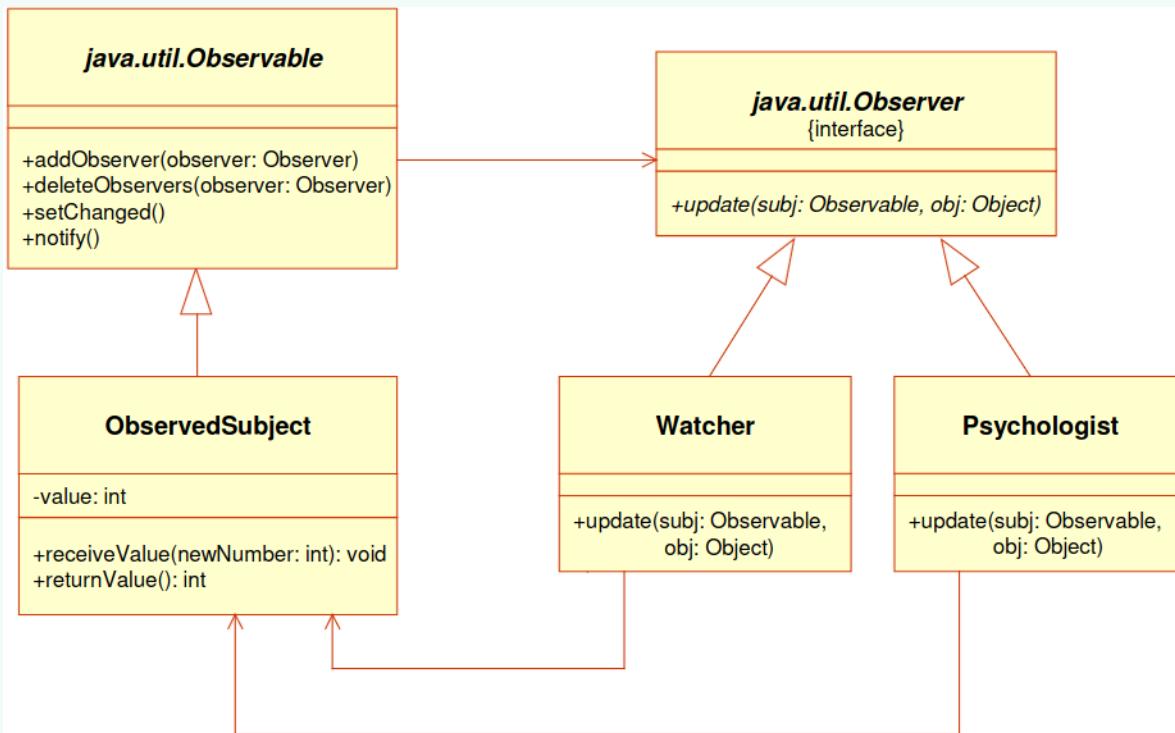
- ⇒ Definisce una dipendenza uno-a-molti tra oggetti. Quando un oggetto cambia stato, tutti i suoi dipendenti vengono notificati;
- ⇒ L'oggetto che notifica il cambiamento di stato non fa nessuna assunzione sulla natura degli oggetti notificati (sono disaccoppiati);
- ⇒ Il numero degli oggetti affetti 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 cancellarsi) dinamicamente con il publisher;
- ⇒ Spesso viene associato al pattern MVC.



Esempio 7.4.1 (Funzionamento di Observer)



Esempio 7.4.2 (Applicazione di Observer)



7.4.2 State

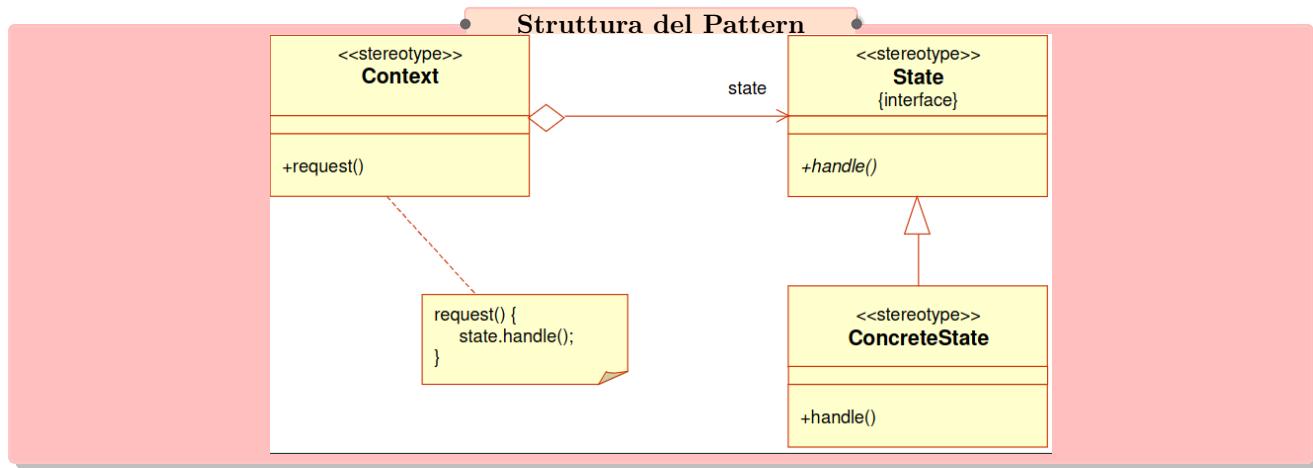
Pattern 7.4.2 (State):

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

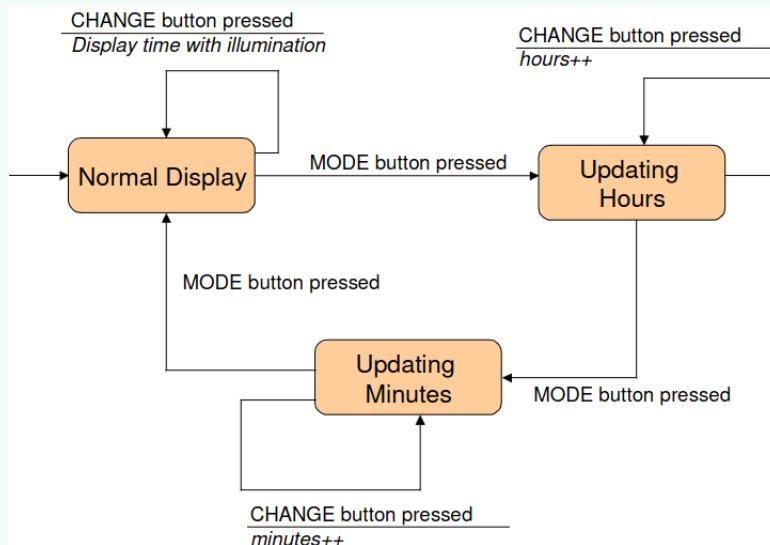
Soluzione: Si creano delle classi stato per ciascuno stato che implementano un'interfaccia comune. Si delegano le operazioni che dipendono dallo stato dell'oggetto contesto all'oggetto stato corrispondente. Si assicura che l'oggetto contesto referenzi sempre un oggetto stato che riflette il suo stato corrente.

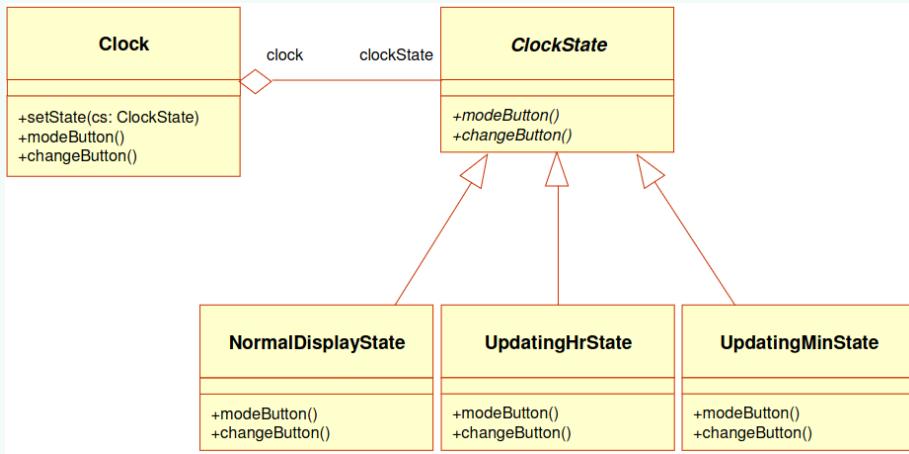
Osservazioni 7.4.2

⇒ Permette a un oggetto di modificare il suo comportamento quando il suo stato interno cambia.



Esempio 7.4.3 (Funzionamento di State)



Esempio 7.4.4 (Applicazione di State)**7.4.3 Strategy****Pattern 7.4.3 (Strategy):**

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

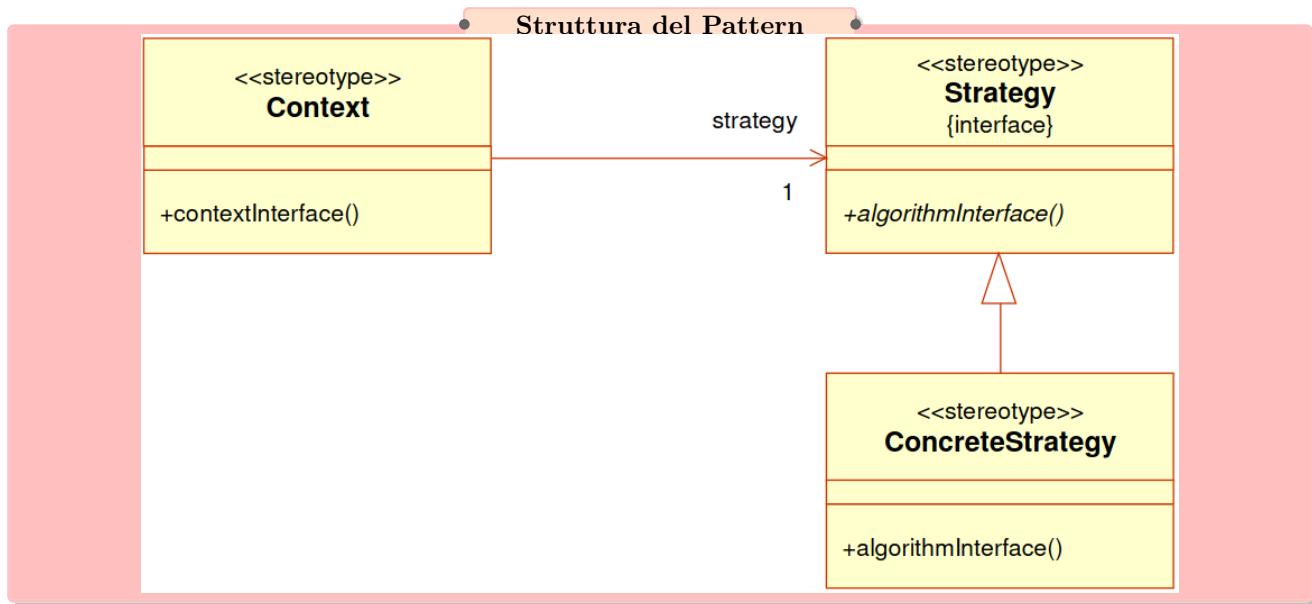
Soluzione: Si definisce ciascun algoritmo/politica/strategia in una classe separata, con un'interfaccia comune.

Note:-

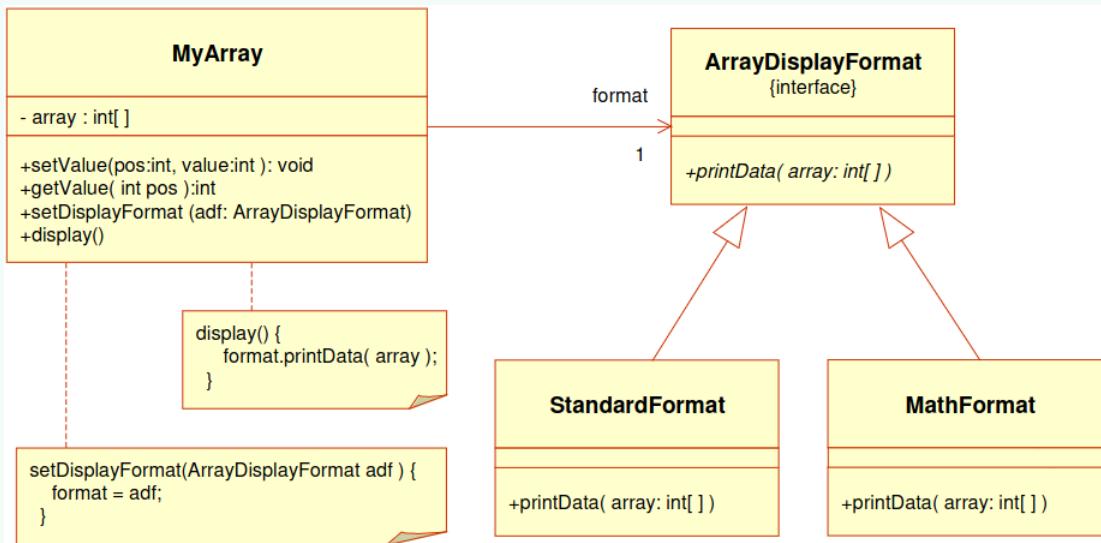
Il pattern Strategy è anche chiamato *policy*.

Osservazioni 7.4.3

- ⇒ 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 l'algoritmo;
- ⇒ Strategy consente la definizione di una famiglia di algoritmi, li incapsula e li rende intercambiabili;
- ⇒ Strategy consente di variare l'algoritmo indipendentemente dagli oggetti che lo usano;
- ⇒ Disaccoppia gli algoritmi dai clienti che vogliono usarli dinamicamente;
- ⇒ Permette al cliente di scegliere l'algoritmo più adatto;
- ⇒ È usato per modificare il comportamento di un oggetto a tempo di esecuzione;
- ⇒ Usa la composizione invece dell'ereditarietà.



Esempio 7.4.5 (Applicazione di Strategy)



Corollario 7.4.1 Strategy e State

- ⇒ State si occupa di che cosa (stato o tipo) un oggetto è (al suo interno) e incapsula un comportamento che dipende dallo stato;
- ⇒ Strategy si occupa di come un oggetto fa qualcosa e incapsula un algoritmo. Un'implementazione diversa che realizza la stessa cosa, in modo che un'implementazione possa sostituire l'altra a seconda della strategia richiesta.

7.4.4 Visitor

Pattern 7.4.4 (Visitor):

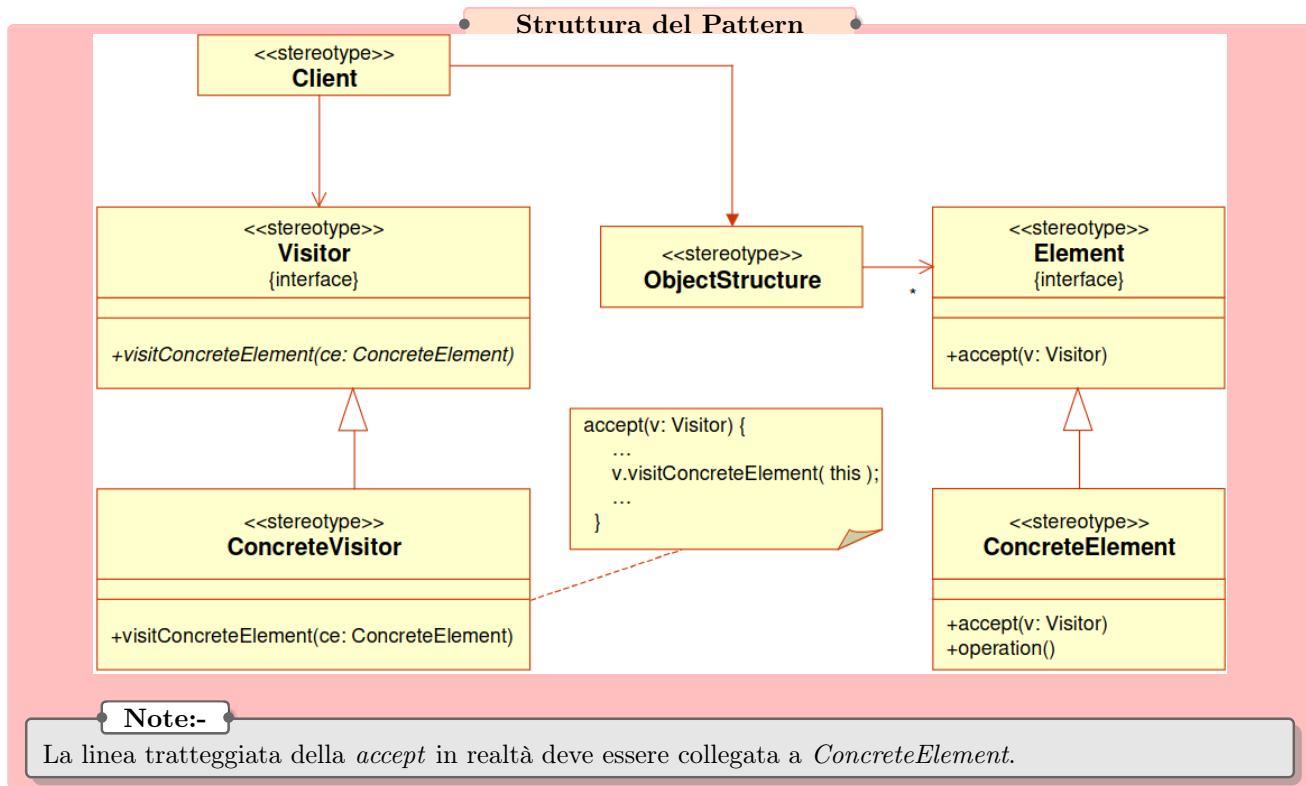
Problema: Come separare l'operazione applicata su un contenitore complesso dalla struttura dati cui è applicata? Come poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa? Come attraversare il contenitore complesso i cui elementi sono eterogenei applicando azioni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

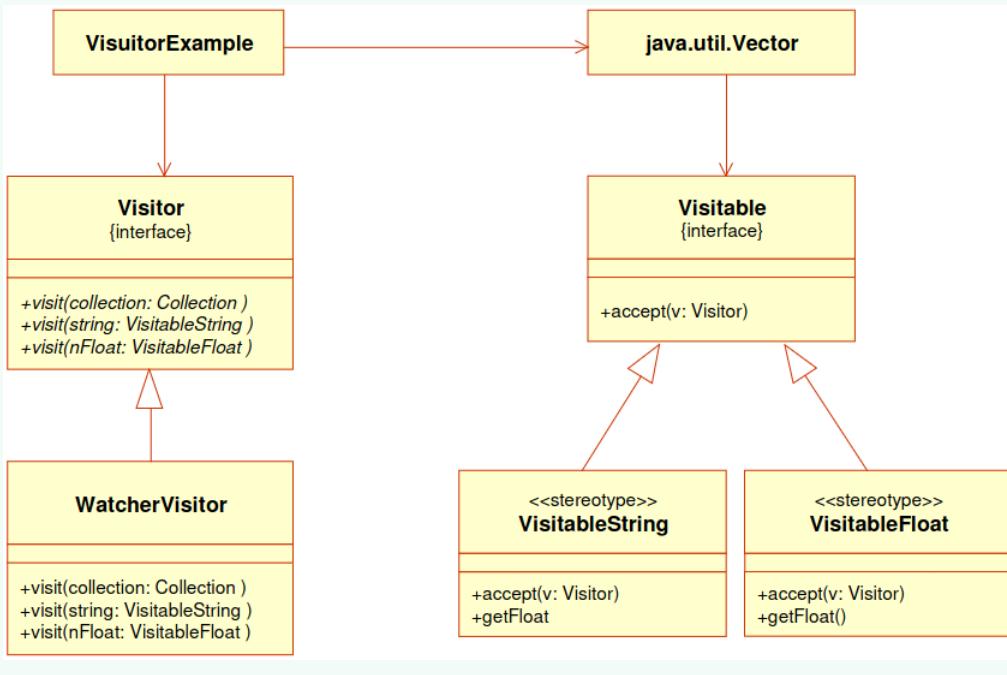
Soluzione: Si crea un oggetto (ConcreteVisitor) che è in grado di percorrere la collezione, e di applicare un metodo proprio su un oggetto (Element) visitato nella collezione (avendo un riferimento a questi ultimi come parametro). Ogni oggetto della collezione aderisce a un'interfaccia (Visitable) che 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.

Osservazioni 7.4.4

Visitor consente:

- ⇒ Flessibilità delle operazioni;
- ⇒ Organizzazione logica;
- ⇒ Vista di vari tipi di classi;
- ⇒ Mantenimento di uno stato aggiornabile;
- ⇒ Le diverse modalità di vista della struttura possono essere definite come sottoclassi di Visitor.



Esempio 7.4.6 (Applicazione di Visitor)**Note:-**

Nella classe VisitableString `getFloat` è `getString`.

8

Dal progetto al codice

8.1 Trasformare i progetti in codice

Definizione 8.1.1: Modello di implementazione

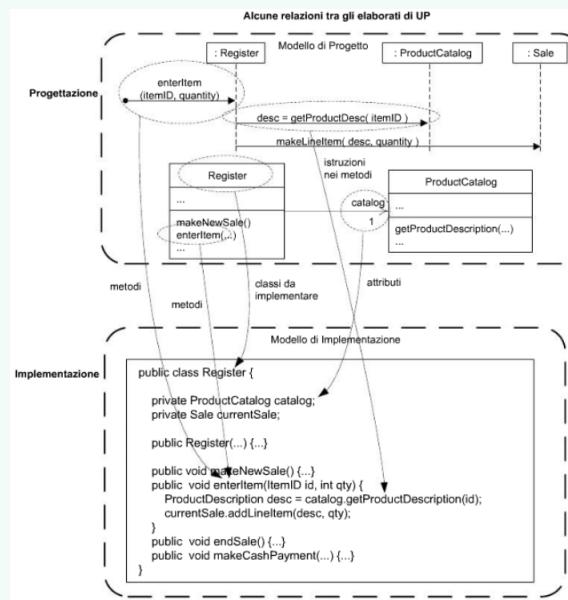
Un *modello di implementazione* è costituito da tutti gli elaborati dell'implementazione, come il *codice sorgente*, la definizione delle *basi di dati*, le pagine JSP/XML/HTML, etc.

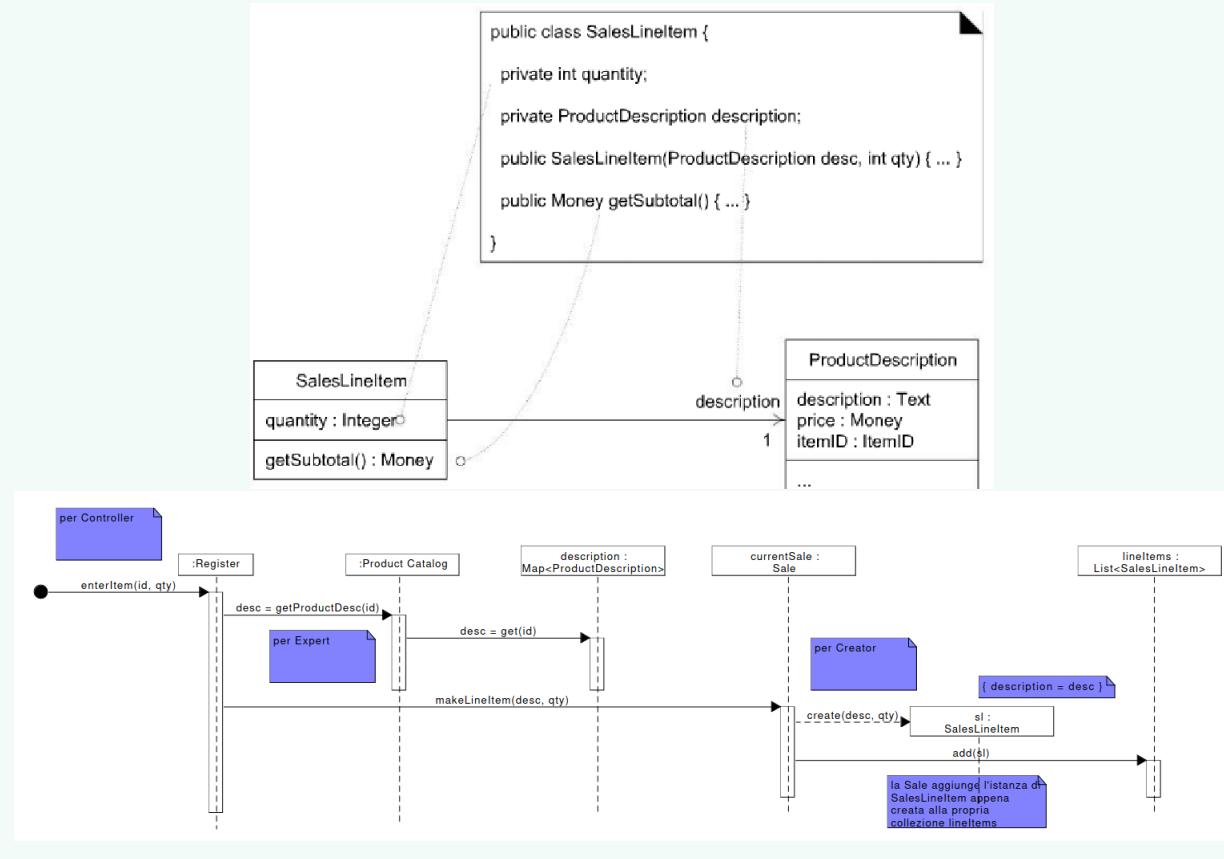
- ⇒ L'implementazione è un processo di traduzione relativamente meccanico del progetto in codice;
- ⇒ Ci possono essere dei cambiamenti e delle deviazioni rispetto al progetto originale.

Note:-

Java viene utilizzato come linguaggio di esempio, ma i concetti sono applicabili a qualsiasi linguaggio di programmazione OO.

Esempio 8.1.1 (Traduzione)



Esempio 8.1.2 (Traduzione di attributi e firme)**8.1.1 Collezioni e costruttori****Definizione 8.1.2: Collezione**

Una *collezione* è un contenitore di oggetti, che può essere usato per memorizzare, recuperare e manipolare dati.

Note:-

Le relazioni uno a molti sono generalmente implementate tramite collezioni.

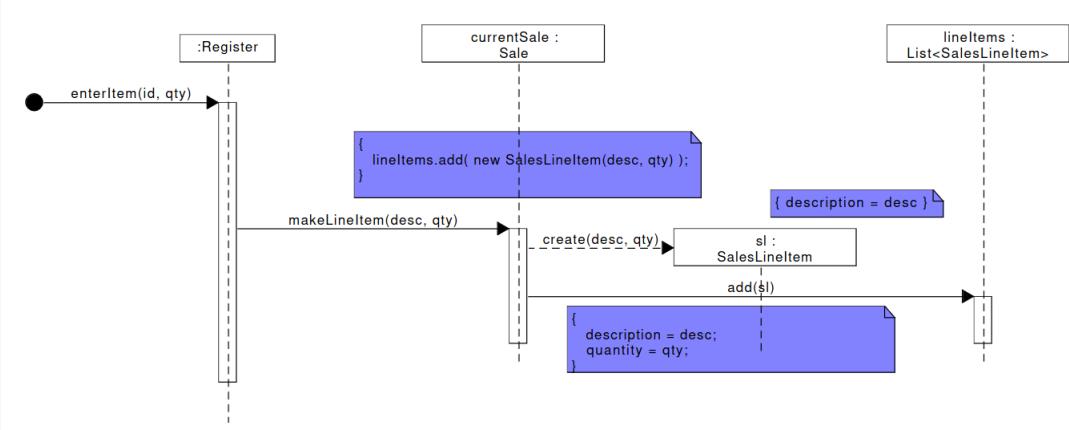
Esempio 8.1.3 (Associazioni)

Definizione 8.1.3: Costruttore

Un *costruttore* è un metodo speciale che viene invocato quando un oggetto viene creato.

⇒ I costruttori possono venir scritti per ispezione da un diagramma delle classi.

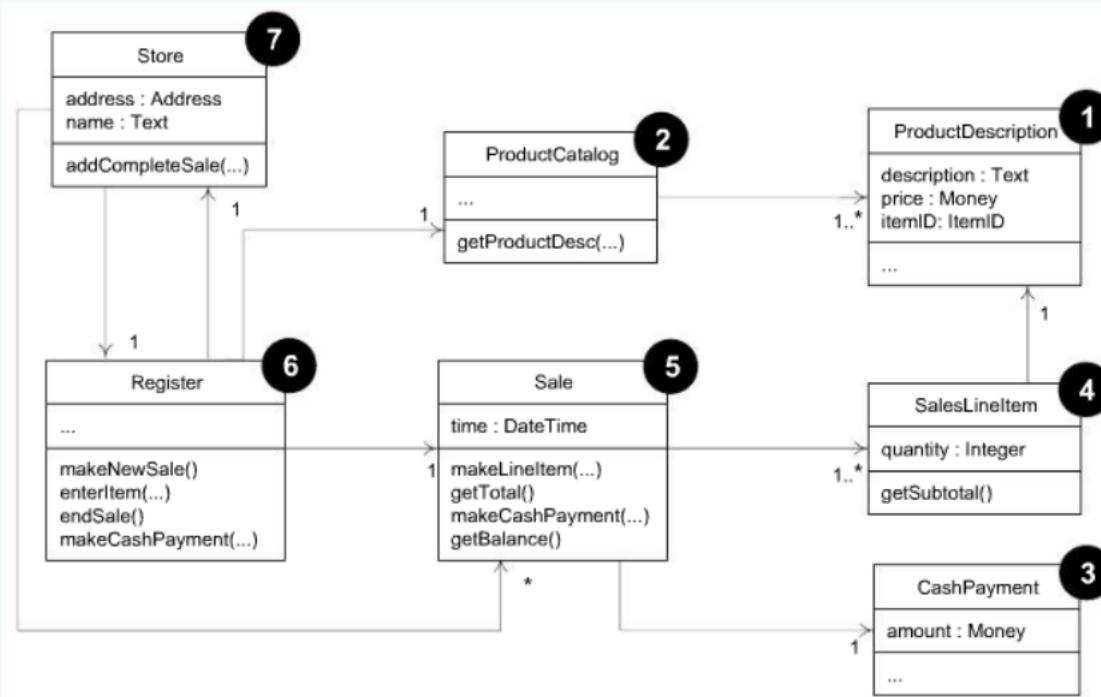
Esempio 8.1.4 (Costruttori)



8.1.2 Ordine di implementazione

⇒ Le classi possono essere implementate in modo e ordine diversi (es. dalla meno accoppiata alla più accoppiata).

Esempio 8.1.5 (Ordine di implementazione)



8.2 Sviluppo test-driven e refactoring

8.2.1 I test

⇒ XP ha promosso la *pratica dei test*: si scrivono i test prima di scrivere il codice.

Definizione 8.2.1: Test-Driven Development (TDD)

Il *Test-Driven Development* è una pratica di sviluppo software che prevede la scrittura dei test prima di scrivere il codice.

Note:-

Si immagina che il codice sorgente sia già stato scritto e si scrivono i test che si aspettano che il codice passi.

Vantaggi:

- ✓ La soddisfazione del programmatore porta a una scrittura più coerente dei test;
- ✓ L'interfaccia e i comportamenti sono più chiari;
- ✓ La verifica è dimostrabile, ripetibile e automatizzabile;
- ✓ Si ha maggiore fiducia nei cambiamenti.

Ci sono diversi tipi di test:

- ⇒ *Test unitari*¹: testano singole unità di codice;
- ⇒ *Test di integrazione*: testano la comunicazione tra specifiche parti;
- ⇒ *Test end-to-end*: testano l'intero sistema;
- ⇒ *Test di accettazione*: testano il sistema, considerandolo a scatola nera, dal punto di vista dell'utente (con riferimento ai Casi d'Uso).

I test unitari sono composti da quattro fasi:

1. *Preparazione*: si crea l'oggetto (o il gruppo di oggetti) da testare (*fixture*) e si preparano altri oggetti e/o risorse necessarie per i test;
2. *Esecuzione*: si fa fare qualcosa alla fixture, viene richiesto un comportamento specifico;
3. *Verifica*: si verifica che i risultati ottenuti siano quelli attesi;
4. *Rilascio*: si rilascia la fixture e si puliscono le risorse (per evitare la corruzione di altri test).

8.2.2 Il refactoring

Note:-

Gli sviluppi incrementali degradano la qualità del codice.

⇒ XP ha promosso il *refactoring continuo*: si modifica il codice per renderlo più chiaro e più semplice da capire, con meno duplicazioni.

Definizione 8.2.2: Refactoring

Il *refactoring* è un metodo strutturato e disciplinato per scrivere o ristrutturare del codice esistente, senza cambiarne il comportamento esterno. Si applicano piccoli passi di trasformazione in combinazione con i test a ogni passo.

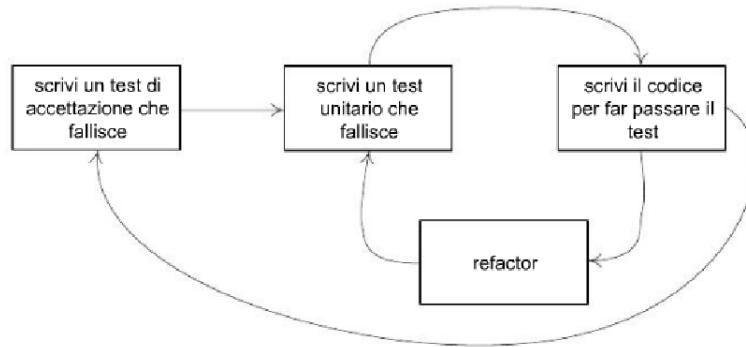
¹Visti in "Algoritmi e strutture dati"

Corollario 8.2.1 Refactoring e test

Dopo ciascuna trasformazione, si eseguono i test unitari per verificare che il refactoring non abbia provocato una **regressione** (fallimento).

Regole per TDD e refactoring:

1. Si scrive un test unitario che fallisce per dimostrare la mancanza di una funzionalità o di codice;
2. Si scrive il codice per far passare il test;
3. Si riscrive o ristruttura il codice, migliorandolo oppure si passa al prossimo test unitario.

**Gli obiettivi del refactoring sono:**

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

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