

# 01 . Processi per lo sviluppo software

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

### **Si noti che**

questi lucidi sono basati sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016” e sul materiale fornito dai docenti Viviana Bono, Claudia Picardi e Gianluca Torta dell’Università degli Studi di Torino che hanno tenuto il corso negli anni accademici precedenti.

# Table of contents

1. Object-Oriented Analysis/Design
2. Unified Modeling Language
3. Processi per lo sviluppo software
4. Sviluppo iterativo ed evolutivo
5. Unified Process

# Object-Oriented Analysis/Design

---

# OOD/A - UML e Pattern

Introduzione all'Object-Oriented Analysis/Design (**OOA/D**) con l'applicazione dell'Unified Modeling Language (**UML**) e dei **pattern**.

UML non è OOA/D: UML come strumento per *pensare e comunicare*.

OOD, progettazione guidata dalle **responsabilità**:

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

Euristiche, *best practice*, **pattern** aiutano in questo codificando principi di progettazione esemplari, coppie problema-soluzione.

# OOD/A - UML e Pattern

Introduzione all'Object-Oriented Analysis/Design (**OOA/D**) con l'applicazione dell'Unified Modeling Language (**UML**) e dei **pattern**.

UML non è OOA/D: UML come strumento per *pensare e comunicare*.

OOD, progettazione guidata dalle **responsabilità**:

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

Euristiche, *best practice*, **pattern** aiutano in questo codificando principi di progettazione esemplari, coppie problema-soluzione.

## **La progettazione orientata agli oggetti**

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

## OOD/A - Casi d'uso

L'OOD è fortemente correlata all'attività dell'**analisi dei requisiti**:

- **Casi d'uso**
- **Storie utente**

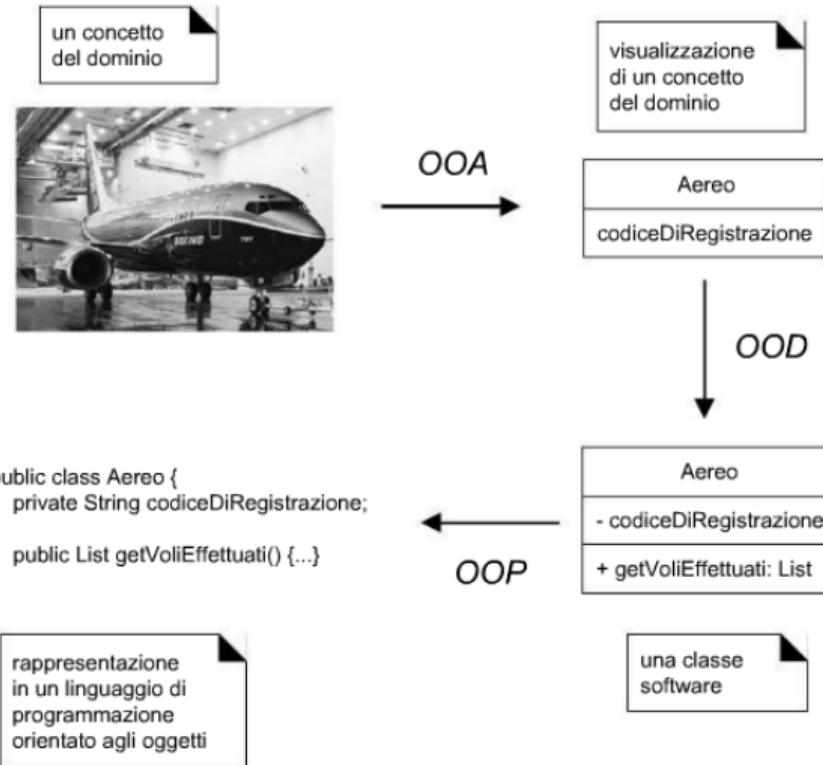
L'OOD è fortemente correlata all'attività dell'**analisi dei requisiti**:

- **Casi d'uso**
- **Storie utente**

## **L'analisi orientata agli oggetti**

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

# OOA, OOD e OOP



# Un esempio: il gioco dei dadi

## 1. Definizione dei casi d'uso: storie scritte

**Gioca una partita a dadi:** 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.

# Un esempio: il gioco dei dadi

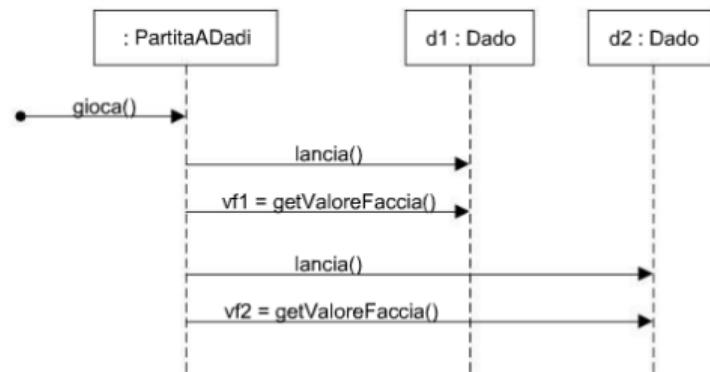
1. Definizione dei casi d'uso: storie scritte
2. Definizione di un modello di dominio: i concetti o gli oggetti significativi del dominio



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

# Un esempio: il gioco dei dadi

1. Definizione dei casi d'uso: storie scritte
2. Definizione di un modello di dominio: i concetti o gli oggetti significativi del dominio
3. Assegnare responsabilità agli oggetti e disegnare diagrammi di interazione: responsabilità e collaborazioni



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

# Un esempio: il gioco dei dadi

1. Definizione dei casi d'uso: storie scritte
2. Definizione di un modello di dominio: i concetti o gli oggetti significativi del dominio
3. Assegnare responsabilità agli oggetti e disegnare diagrammi di interazione: responsabilità e collaborazioni
4. Definizione dei diagrammi delle classi di progetto



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

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

- **Processo di sviluppo iterativo**
- **Approccio agile**
- **Unified Process (UP)**

# Unified Modeling Language

---

## Unified Modeling Language (UML)

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

UML è uno standard *de fact* per la notazione di diagrammi per disegnare o rappresentare figure relative al software, e in particolare al software OO.

L'utilizzo di UML può essere come “abbozzo” (diagrammi informali per esplorare e comunicare), come “progetto“ o come “linguaggio di programmazione” .

La modellazione agile enfatizza l'uso di UML come “abbozzo” .

- **Punto di vista concettuale** (modello di dominio): la notazione dei diagrammi di UML è utilizzata per visualizzare concetti del modo reale (classe concettuale)



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

# Uso di UML

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



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



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

Anni Sessanta e Settanta: nascita dei linguaggi OO (Simula, Smalltalk).

Nel 1988 Bertrand Meyer pubblica “Object-Oriented Software Construction”.

Nel 1991 Jim Rumbaugh pubblica “Object-Oriented Modeling and Design” (OOA/D).

Nel 1991 Grady Booch pubblica “Object-Oriented Software Engineering” (OOA/D e Casi d'uso per i requisiti)

Nel 1994 Booch e Rumbaugh fanno nascere dalle rispettive proposte UML.

Rational Corporation: Ivar Jacobson, Grady Booch e Jim Rumbaugh (i “tre amigos”) si concentrano sulla notazione piuttosto che su di un metodo comune.

L'*Object Management Group* (OMG), un consorzio per la standardizzazione industriale dei concetti relativi all'OO, avvia la standardizzazione di UML.

Nel 1997 UML 1, nel 2004 UML 2.

## **Processi per lo sviluppo software**

---

# Software di qualità

- Il software non è un semplice programma o un gruppo di programmi
- Il software include anche tutta la documentazione elettronica che serve agli utenti dei sistemi, agli sviluppatori e ai responsabili della garanzia della qualità
- Le caratteristiche essenziali di un prodotto software sono:
  - la **mantenibilità**: il software dovrebbe essere scritto in modo da evolversi in rapporto alle nuove richieste dei clienti
  - la **fidatezza**: un software fidato non dovrebbe causare danni fisici o economici nel caso di malfunzionamento del sistema
  - l'**efficienza**: il software non dovrebbe fare un uso dispendioso delle risorse del sistema, come la memoria o i cicli di processore
  - l'**accettabilità**: il software deve essere accettabile per il tipo di utenti per i quali è progettato, ossia comprensibile, usabile e compatibile con gli altri sistemi che essi usano

# Processo software

**In generale, un processo**

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

# Processo software

## In generale, un processo

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

## Un processo per lo sviluppo del software (o processo software)

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

# Processo software

## In generale, un processo

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

## Un processo per lo sviluppo del software (o processo software)

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

Quattro attività fondamentali sono comuni a tutti i processi software:

1. **Specifiche del software**: clienti e sviluppatori definiscono le funzionalità e i vincoli operativi del software da produrre
2. **Sviluppo del software**: viene progettato e sviluppato il software
3. **Convalida del software**: il software viene convalidato per garantire ciò che il cliente richiede
4. **Evoluzione del software**: il software viene modificato per soddisfare eventuali cambiamenti dei requisiti del cliente e del mercato

## Processo software

Differenti tipi di sistemi richiedono differenti approcci di sviluppo: il software real-time deve essere specificato in maniera completa prima che inizi lo sviluppo, mentre nei sistemi di commercio elettronico le specifiche e il programma sono solitamente sviluppati assieme.

# Processo software

Differenti tipi di sistemi richiedono differenti approcci di sviluppo: il software real-time deve essere specificato in maniera completa prima che inizi lo sviluppo, mentre nei sistemi di commercio elettronico le specifiche e il programma sono solitamente sviluppati assieme.

## Esempi di processi software sono

il processo 'a cascata', Unified Process (UP), Scrum, il modello di sviluppo a spirale, lo sviluppo rapido di applicazione (RAD), Extreme Programming (XP).

- Il **modello di processo software** o **paradigma di processo** è una rappresentazione semplificata di un processo software: sono strutture di processo da estendere e adattare per creare processi software più specifici
- **Modello a cascata:** le attività di processo fondamentali, specifica, sviluppo e convalida sono rappresentate come fasi distinte del processo
- **Sviluppo incrementale:** intreccia le attività di specifica, sviluppo e convalida. Il sistema viene sviluppato come una serie di versioni (incrementi), ciascuna delle quali aggiunge nuove funzionalità alla versione precedente
- **Integrazione e configurazione:** si basa sulla disponibilità di un gran numero di componenti o sistemi riutilizzabili. Il processo di sviluppo si basa sulla configurazione di questi componenti per utilizzarli in una nuova disposizione e integrarli in un sistema

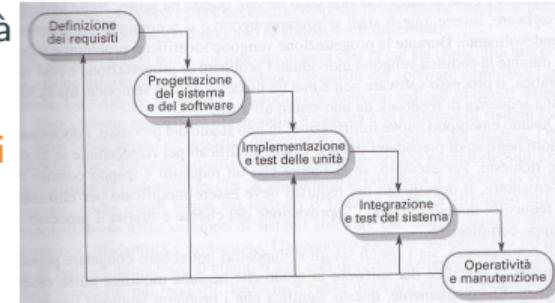
- Non esiste un modello di processo universale che si possa applicare appropriatamente a tutti i tipi di sviluppo del software
- Il processo appropriato dipende dai requisiti dei clienti e delle politiche normative, dall'ambiente in cui il software sarà utilizzato e dal tipo di software che si intende sviluppare, ad esempio:
  - Il software a sicurezza critica di solito è sviluppato utilizzando un processo a cascata in quanto sono richieste molte analisi e documentazione prima di iniziare l'implementazione
  - I prodotti software adesso sono sempre sviluppati utilizzando un modello di processo incrementale
  - I sistemi aziendali vengono sempre più sviluppati configurando e integrando i sistemi esistenti per creare un nuovo sistema con le funzionalità richieste

- Gran parte dello sviluppo pratico del software si basa su un modello generale, ma spesso incorpora caratteristiche di altri modelli
- Ingegnerizzazione di grandi sistemi: ha senso combinare alcune delle migliori caratteristiche di tutti i processi generali: è necessario avere informazioni sui requisiti essenziali dei sistemi per progettare un'architettura software a supporto di tali requisiti, non è possibile procedere solo in modo incrementale
- I sottosistemi di un sistema più grande possono essere sviluppati utilizzando approcci differenti: le parti del sistema che sono ben chiare possono essere specificate e sviluppate utilizzando il modello a cascata, altre parti del sistema, che sono difficili da specificare preventivamente, dovrebbero essere sempre sviluppate utilizzando un approccio incrementale

# Processo a cascata (o sequenziale)

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

- all'inizio del progetto vengono definiti in dettaglio tutti i requisiti

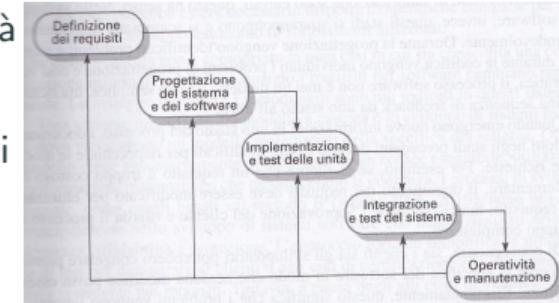


© I. Sommerville. Ingegneria del software. Pearson, 2017.

# Processo a cascata (o sequenziale)

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

- all'inizio del progetto vengono definiti in dettaglio tutti i requisiti
- all'inizio del progetto viene definito un piano temporale dettagliato delle attività da svolgere

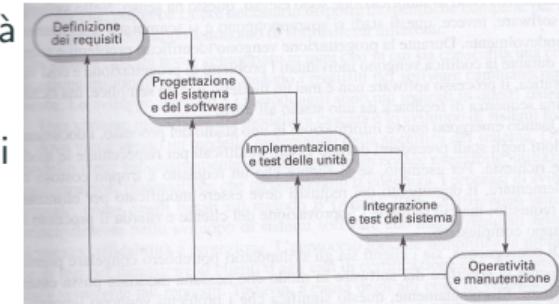


© I. Sommerville. Ingegneria del software. Pearson, 2017.

# Processo a cascata (o sequenziale)

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

- all'inizio del progetto vengono definiti in dettaglio tutti i requisiti
- all'inizio del progetto viene definito un piano temporale dettagliato delle attività da svolgere
- si prosegue con la modellazione (analisi e progettazione)

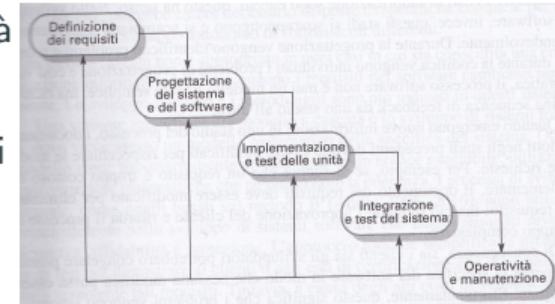


©I. Sommerville. Ingegneria del software. Pearson, 2017.

# Processo a cascata (o sequenziale)

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

- all'inizio del progetto vengono definiti in dettaglio tutti i requisiti
- all'inizio del progetto viene definito un piano temporale dettagliato delle attività da svolgere
- si prosegue con la modellazione (analisi e progettazione)
- quindi viene creato un progetto completo del software

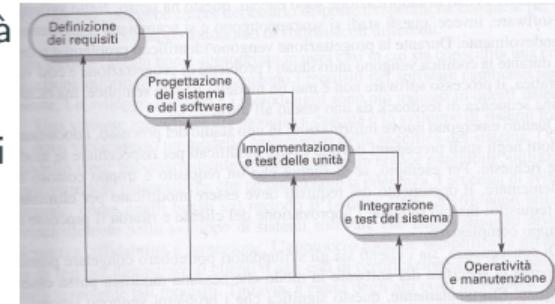


©I. Sommerville. Ingegneria del software. Pearson, 2017.

# Processo a cascata (o sequenziale)

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

- all'inizio del progetto vengono definiti in dettaglio tutti i requisiti
- all'inizio del progetto viene definito un piano temporale dettagliato delle attività da svolgere
- si prosegue con la modellazione (analisi e progettazione)
- quindi viene creato un progetto completo del software
- a questo punto inizia la programmazione del sistema software

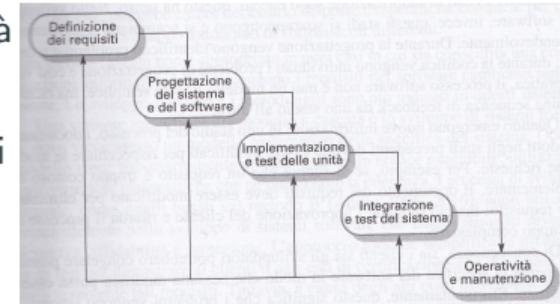


© I. Sommerville. Ingegneria del software. Pearson, 2017.

# Processo a cascata (o sequenziale)

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

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



©I. Sommerville. Ingegneria del software. Pearson, 2017.

## Processo a cascata (o sequenziale)

- Il modello di processo di sviluppo a cascata fu derivato dai modelli utilizzati nell'ingegnerizzazione di grandi sistemi militari
- Il modello a cascata era piuttosto comune fino a pochi anni fa
- Il modello a cascata è un esempio di processo guidato da un piano: almeno in principio, occorre pianificare tutte le attività di processo prima di iniziare lo sviluppo del software, è appropriato:
  - nei sistemi integrati, dove il software deve interfacciarsi con i sistemi hardware
  - i sistemi critici, dove occorre un'analisi approfondita della sicurezza e della protezione del software
  - grandi sistemi software che fanno parte di sistemi più complessi sviluppati da più società

## Processo a cascata (o sequenziale)

In Larman, C. 2003. *Agile and Iterative Development: A manager's Guide* Reading, MA., Addison-Wesley e in Larman, C. Basili, V. *Iterative and Incremental Development: A History*, IEEE Computer, June 2003 gli autori mostrano come:

- il metodo a cascata sia una pratica mediocre per la maggior parte dei progetti software, anziché un approccio valido
- l'approccio a cascata sia caratterizzato da una minore produttività e da maggiori percentuali di difetti
- le stime iniziali a cascata dei tempi e dei costi variano notevolmente dai valori finali

## Processo a cascata (o sequenziale)

In Larman, C. 2003. *Agile and Iterative Development: A manager's Guide* Reading, MA., Addison-Wesley e in Larman, C. Basili, V. *Iterative and Incremental Development: A History*, IEEE Computer, June 2003 gli autori mostrano come:

- il metodo a cascata sia una pratica mediocre per la maggior parte dei progetti software, anziché un approccio valido
- l'approccio a cascata sia caratterizzato da una minore produttività e da maggiori percentuali di difetti
- le stime iniziali a cascata dei tempi e dei costi variano notevolmente dai valori finali

### Perché? Perché partono dal presupposto che

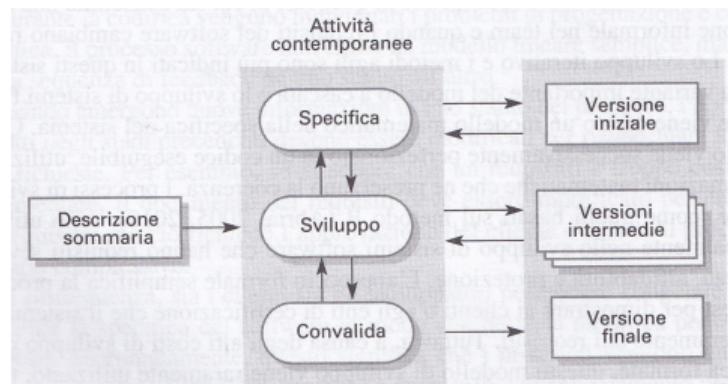
le specifiche sono prevedibili e stabili e possono essere definite correttamente sin dall'inizio, a fronte di un basso tasso di cambiamenti.

# Sviluppo incrementale

Lo sviluppo incrementale si basa sull'idea di sviluppare un'implementazione iniziale, esporla agli utenti e perfezionarla attraverso molte versioni, finché non si ottiene il sistema richiesto

## Le attività di

specificazione, sviluppo e convalida sono intrecciate anziché separate, con feedback veloci tra le varie attività.



## Sviluppo incrementale

- Lo sviluppo incrementale in qualche forma è adesso l'approccio più comune per sviluppare sistemi di applicazioni e prodotti software
- Questo approccio può essere plan-driven, agile o una combinazione di questi approcci
- In un approccio plan-driven, gli incrementi del sistema sono stabiliti in anticipo
- In un approccio agile, vengono identificati gli incrementi iniziali, ma lo sviluppo dei successivi incrementi dipende dall'avanzamento del lavoro e dalle priorità del cliente

# Sviluppo incrementale

- Lo sviluppo incrementale è migliore dell'approccio a cascata per quei sistemi i cui requisiti è probabile che cambino durante il processo di sviluppo: è questo il caso di molti sistemi aziendali e prodotti software
- Riflette il modo in cui risolviamo i problemi: raramente troviamo in anticipo la soluzione finale di un problema, ma arriviamo alla soluzione tramite una serie di passaggi, tornando indietro quando ci accorgiamo di aver commesso un errore
- È più economico e semplice apportare modifiche al software mentre viene sviluppato
- Ciascun incremento o versione del sistema incorpora qualcosa delle funzionalità richieste dal cliente
- Gli incrementi iniziali includono la funzionalità più importante e urgente richiesta dal cliente, così che il cliente possa valutare il sistema nelle prime fasi di sviluppo e, se necessario, modificato solamente l'incremento corrente

# Sviluppo incrementale

- Il costo di implementazione delle modifiche dei requisiti è ridotto
- È più facile ottenere feedback del cliente sul lavoro di sviluppo che è stato fatto
- È possibile consegnare in anticipo al cliente una versione utilizzabile del software, anche se non sono state incluse tutte le funzionalità

però:

- Il processo non è visibile, i manager devono avere delle consegne regolari per misurare i progressi
- La struttura dei sistemi tende a degradarsi quando vengono aggiunti nuovi incrementi. Per ridurre il livello di degrado strutturale e di complicazione del codice, i metodi agili suggeriscono un costante refactoring (miglioramento della struttura) del software

# Sviluppo incrementale VS ‘a cascata’

## Lo sviluppo incrementale o iterativo ed evolutivo

- comporta fin dall'inizio la programmazione e il test di un sistema software
- comporta che lo sviluppo inizi prima che tutti i requisiti siano stati definiti in modo dettagliato
- viene usato il feedback per chiarire e migliorare le specifiche in evoluzione del sistema

# Sviluppo incrementale VS ‘a cascata’

## Lo sviluppo incrementale o iterativo ed evolutivo

- comporta fin dall'inizio la programmazione e il test di un sistema software
- comporta che lo sviluppo inizi prima che tutti i requisiti siano stati definiti in modo dettagliato
- viene usato il feedback per chiarire e migliorare le specifiche in evoluzione del sistema

## Sviluppo incrementale VS ‘a cascata’

- Passi di sviluppo brevi e feedback per chiarire i requisiti VS (grande) lavoro speculativo iniziale
- In base a dati statistici, i metodi iterativi sono associati a percentuali di successo e di produttività più elevate, nonché a livelli minori di difetti

## Sviluppo incrementale o iterativo ed evolutivo: storia

- Alla fine degli anni cinquanta, al progetto spaziale Mercury fu applicato lo sviluppo evolutivo, iterativo e incrementale piuttosto che il metodo a scascata (IDD, Iterative and Incremental Development)



## Sviluppo incrementale o iterativo ed evolutivo: storia

- Alla fine degli anni cinquanta, al progetto spaziale Mercury fu applicato lo sviluppo evolutivo, iterativo e incrementale piuttosto che il metodo a scascata (IDD, Iterative and Incremental Development)
- Il programma Mercury fu il primo programma statunitense a prevedere missioni spaziali con equipaggio



## Sviluppo incrementale o iterativo ed evolutivo: storia

- Alla fine degli anni cinquanta, al progetto spaziale Mercury fu applicato lo sviluppo evolutivo, iterativo e incrementale piuttosto che il metodo a scascata (IDD, Iterative and Incremental Development)
- Il programma Mercury fu il primo programma statunitense a prevedere missioni spaziali con equipaggio
- Trident nuclear programme (missili balistici su sottomarini nucleari)



# Sviluppo incrementale o iterativo ed evolutivo: storia

- Alla fine degli anni cinquanta, al progetto spaziale Mercury fu applicato lo sviluppo evolutivo, iterativo e incrementale piuttosto che il metodo a scascata (IDD, Iterative and Incremental Development)
- Il programma Mercury fu il primo programma statunitense a prevedere missioni spaziali con equipaggio
- Trident nuclear programme (missili balistici su sottomarini nucleari)
- Anni '70: il software di controllo di volo dello Space Shuttle fu costruito con 17 iterazioni di circa quattro settimane l'una



## Sviluppo incrementale o iterativo ed evolutivo: storia

- Alla fine degli anni cinquanta, al progetto spaziale Mercury fu applicato lo sviluppo evolutivo, iterativo e incrementale piuttosto che il metodo a scascata (IDD, Iterative and Incremental Development)
- Il programma Mercury fu il primo programma statunitense a prevedere missioni spaziali con equipaggio
- Trident nuclear programme (missili balistici su sottomarini nucleari)
- Anni '70: il software di controllo di volo dello Space Shuttle fu costruito con 17 iterazioni di circa quattro settimane l'una
- IBM (1968): primo documento che promuoveva lo sviluppo iterativo



## Sviluppo incrementale e agili: storia

- Le aziende oggi lavorano in un ambiente globale dal cambiamento rapido; devono cogliere nuove opportunità, e rispondere a nuovi mercati alle condizioni economiche variabili e alla presenza di prodotti e servizi concorrenti
- Il software è parte essenziali delle operazioni aziendali, deve trarre vantaggio dalle nuove opportunità
- La rapidità dello sviluppo e della consegna è quindi il requisito più critico per la maggior parte dei sistemi software aziendali
- Spesso è praticamente impossibile ottenere un insieme completo di requisiti stabili
- I requisiti cambiano perché per i clienti è impossibile prevedere come un sistema influenzerà le pratiche operative, come interagirà con gli altri sistemi e quali operazioni degli utenti dovranno essere automatizzate
- I requisiti reali diventano chiari solo dopo che il sistema è stato consegnato e utilizzato dagli utenti, fattori esterni possono indurre a modificare ulteriormente i requisiti

## Sviluppo incrementale e agili: storia

---

- Anni '80 e '90: attenta pianificazione dei progetti, la garanzia di qualità formale, l'uso di metodi di analisi e progettazione supportati da strumenti software e processi di sviluppo software controllati e rigorosi (da progetti di grandi dimensioni, sistemi aeroplaziali e governativi)
- Nell'applicare tali principi ai sistemi di aziende di piccole e medie dimensioni, gli overhead richiesti erano troppo alti
- Anni '90: metodi agili, concentrarsi sul software stesso anziché sulla progettazione e sulla documentazione, consegna rapida e requisiti che cambiano rapidamente, consegne incrementali

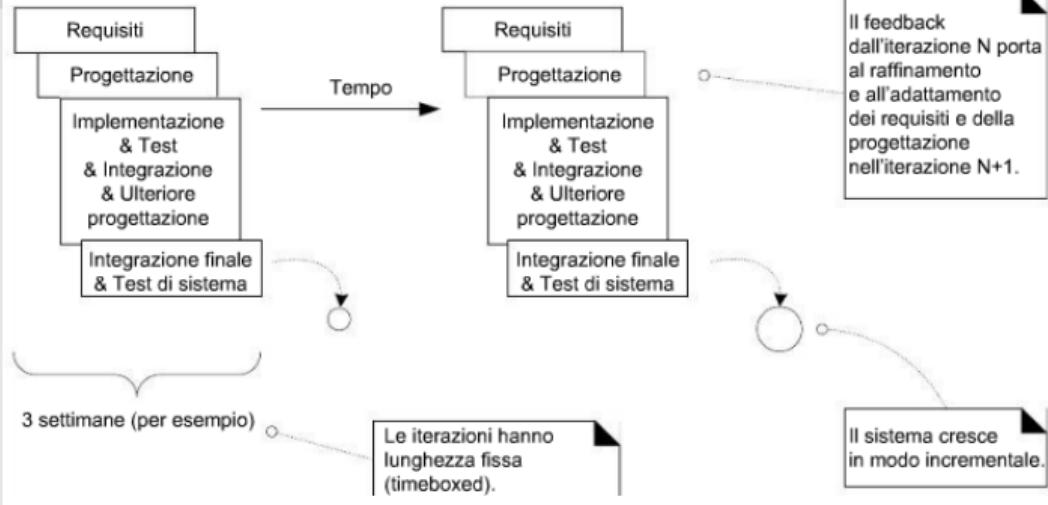
## **Sviluppo iterativo ed evolutivo**

---

# Sviluppo iterativo

## Nell'approccio iterativo

- lo sviluppo è organizzato in una serie di mini-progetti brevi, di lunghezza fissa, chiamati **iterazioni**
- il risultato di ciascuna iterazione è un **sistema eseguibile**, testato e integrato, anche se parziale.
- ciascuna iterazione comprende le proprie attività di analisi dei requisiti, progettazione, imprementazione e test

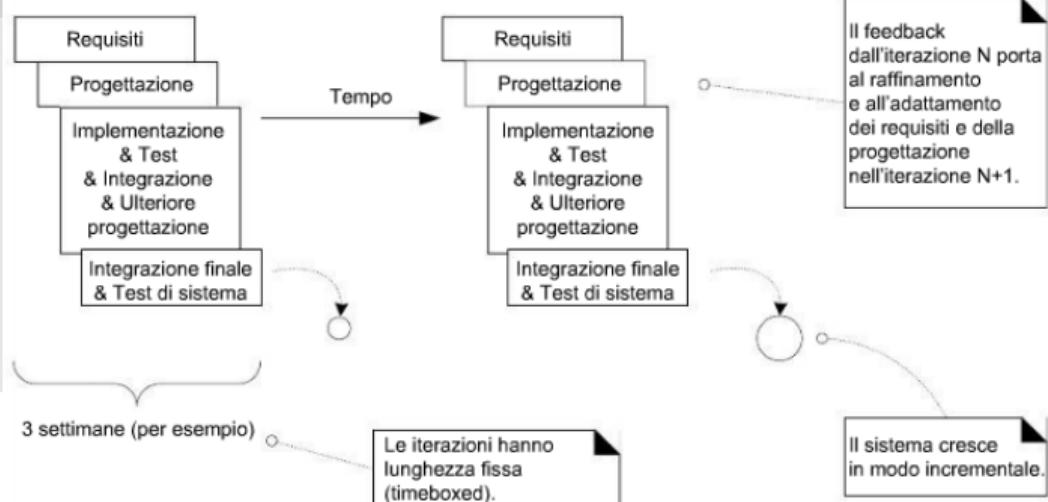


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Sviluppo iterativo

## Sviluppo iterativo e

- **incrementale:** il sistema cresce in modo incrementale nel tempo, iterazione dopo iterazione
- **evolutivo:** il feedback e l'adattamento fanno evolvere le specifiche e il progetto



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

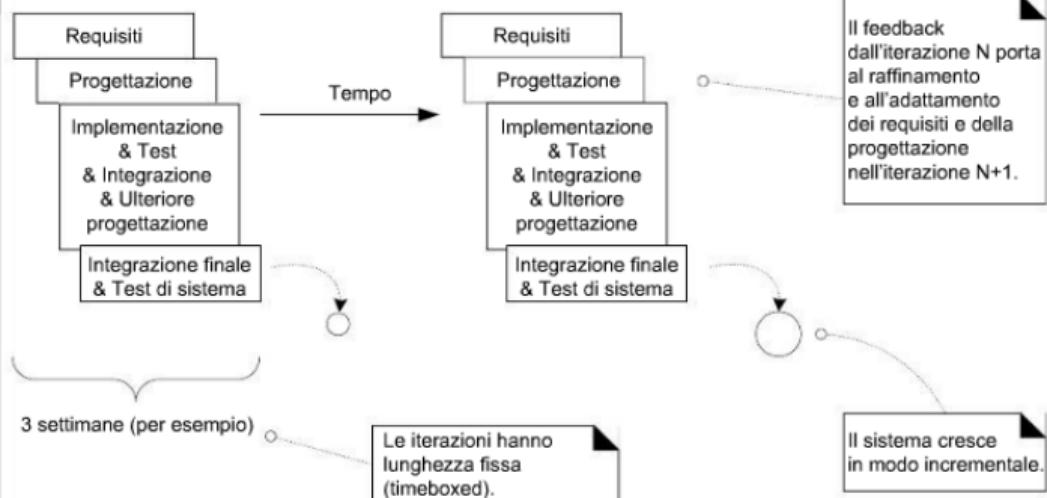
# Sviluppo iterativo

## Nel processo iterativo

Non vi è fretta di iniziare la codifica, né una fase di progettazione lunga che provi a perfezionare tutti i dettagli della progettazione prima della programmazione.

Il risultato di ciascuna iterazione è un **sistema eseguibile** ma **incompleto** (lo sarà dopo 10/15 iterazioni).

Il risultato di un'iterazione **non è un prototipo** ma un **sottoinsieme del sistema finale**.



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

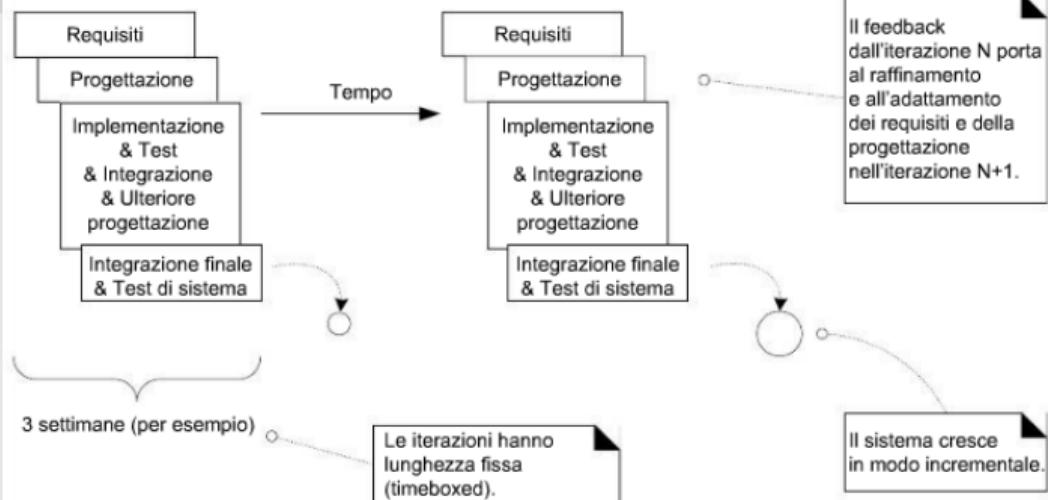
# Sviluppo iterativo

## Nel processo iterativo

Ciascuna iterazione comporta la scelta di un piccolo sottoinsieme di requisiti, una rapida progettazione implementazione e test.

Questo permette **feedback rapidi** da parte degli utenti, degli sviluppatori e dei test.

Un'opportunità per **modificare o adattare** la comprensione dei requisiti e il progetto.

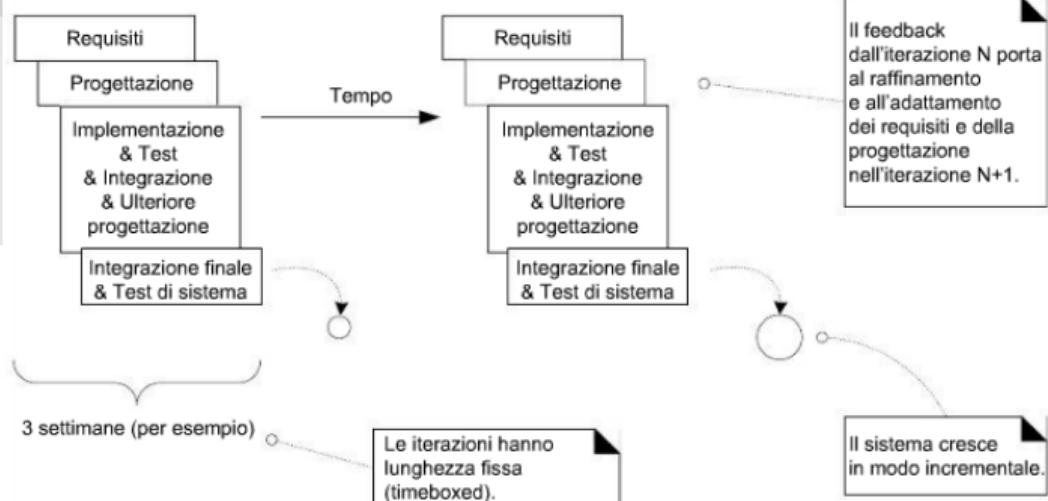


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Sviluppo iterativo

## Nel processo iterativo

Attraverso il feedback iterativo e l'adattamento, il sistema **evolve** e **converge** verso i requisiti corretti e il progetto più appropriato.



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

## Sviluppo iterativo ed evolutivo

Esempi di processi di sviluppo iterativo ed evolutivo sono: **Unified Process (UP)** o *Processo Unificato*, **Extreme Programming (XP)**, **Scrum**.

Vantaggi:

- Riduzione precoce dei rischi maggiori (tecnici, requisiti, obiettivi, usabilità, ...)
- Progresso visibile fin dall'inizio
- Feedback (dai modellatori, sviluppatori, programmatore, cliente, mercato) precoce, coinvolgimento dell'utente e adattamento
- Gestione della complessità (evita la “*paralisi da analisi*” )

Fondamentale è un'iterazione da due a sei settimane, passi piccoli, feedback rapido e adattamento.

Un'iterazione di lunghezza fissata è detta **timeboxed**.

# Sviluppo iterativo ed evolutivo: un esempio

Un esempio di progetto iterativo (UP) composto da 20 iterazioni prima del rilascio al cliente (dal libro di testo, sezione 2.5).

## Passo 1

1. Prima dell'iterazione 1, si tiene il primo workshop dei requisiti timeboxed, per esempio di due giorni esatti. Sono presenti i responsabili dell'organizzazione e dello sviluppo (compreso il chief architect, il capo architetto).
  - La mattina del primo giorno si esegue un'analisi dei requisiti di alto livello, per esempio identificando solo i nomi dei casi d'uso e le caratteristiche, oltre ai requisiti non funzionali più importanti. L'analisi non sarà perfetta.
  - Si chiede al chief architect e ai responsabili dell'organizzazione di scegliere da questo elenco di alto livello il 10% (per esempio il 10% dei 30 nomi dei casi d'uso identificati) che possegga una miscela delle seguenti tre qualità: 1) significatività dal punto di vista dell'architettura (per implementarlo, richiede di progettare, costruire e testare il nucleo dell'architettura); 2) elevato valore di business (si tratta di caratteristiche importanti per l'organizzazione); 3) rischio elevato (per esempio che "sia in grado di gestire 500 transazioni concorrenti"). Per esempio, procedendo in questo modo vengono identificati tre casi d'uso: UC2, UC11 e UC14.
  - Per il rimanente giorno e mezzo, si esegue un'analisi intensa e dettagliata dei requisiti funzionali e non funzionali per questi tre casi d'uso. Al termine, il 10% dei requisiti è stato analizzato in profondità, e il rimanente 90% solo ad alto livello.

# Sviluppo iterativo ed evolutivo: un esempio

Un esempio di progetto iterativo (UP) composto da 20 iterazioni prima del rilascio al cliente (dal libro di testo, sezione 2.5).

## Passi 2 e 3

2. Prima dell'iterazione 1, si tiene una riunione di pianificazione dell'iterazione in cui si sceglie un sottoinsieme di requisiti da UC2, UC11 e UC14 su cui fare **progettazione, implementazione e test** entro un tempo specificato (per esempio, un'iterazione timeboxed di tre settimane). Si noti che potrebbe non essere possibile sviluppare interamente i tre casi d'uso scelti, perché questo potrebbe richiedere troppo lavoro. Dopo aver scelto un sottoinsieme specifico di obiettivi, questi vanno suddivisi in un insieme di compiti più dettagliati da svolgere nell'iterazione, anche sulla base delle indicazioni del team di sviluppo.
3. Si esegue l'iterazione 1 in tre settimane (occorre attenersi al timebox scelto).
  - I primi due giorni, gli sviluppatori e gli altri fanno **modellazione e progettazione a coppie**, abbozzando diagrammi UML su più lavagne (abbozzando eventualmente anche altri tipi di modelli) nella stanza comune del progetto, guidati e istruiti dal chief architect.

# Sviluppo iterativo ed evolutivo: un esempio

Un esempio di progetto iterativo (UP) composto da 20 iterazioni prima del rilascio al cliente (dal libro di testo, sezione 2.5).

## Passo 3

- Gli sviluppatori si levano il “cappello da modellatore” e si mettono il “cappello da programmatore”, passando dunque dalla modellazione alla programmazione. Iniziano a programmare, fare test e integrazione in modo continuo nelle settimane rimanenti dell’iterazione, utilizzando i modelli abbozzati come punto di partenza e di ispirazione, sapendo però che i modelli sono parziali e spesso solo approssimativi.
- Viene eseguito un grosso lavoro di test: unitari, di accettazione, di carico, di usabilità e così via.
- Una settimana prima della fine, si chiede al team se gli obiettivi originari dell’iterazione possono essere raggiunti; in caso contrario, si riduce la portata dell’iterazione, rimettendo gli obiettivi secondari nell’elenco delle cose da fare nelle iterazioni successive.
- Il martedì dell’ultima settimana il codice viene congelato; tutto il codice deve essere caricato, integrato e testato per creare la baseline dell’iterazione.
- Il mercoledì mattina viene fatta una dimostrazione del sistema parziale alle parti interessate esterne, per rendere visibili i progressi iniziali. Alle parti interessate viene richiesto un feedback.

# Sviluppo iterativo ed evolutivo: un esempio

Un esempio di progetto iterativo (UP) composto da 20 iterazioni prima del rilascio al cliente (dal libro di testo, sezione 2.5).

## Passi 4, 5, 6 e 7

4. Si esegue il secondo workshop sui requisiti verso la fine dell'iterazione 1, per esempio l'ultimo mercoledì e giovedì. Tutto il materiale dell'ultimo workshop viene rivisto e raffinato. Viene quindi scelto un altro 10% o 15% dei casi d'uso significativi dal punto di vista dell'architettura e di elevato valore di business, lo si analizza in dettaglio per uno o due giorni. Al termine, probabilmente il 25% dei casi d'uso e dei requisiti non funzionali sarà stato scritto in modo dettagliato. Ma non sarà perfetto.
5. Il venerdì mattina si tiene un'altra riunione di pianificazione dell'iterazione, per l'iterazione successiva.
6. Si esegue l'iterazione 2, con gli stessi passi.
7. Si ripete il tutto, per quattro iterazioni e cinque workshop dei requisiti, di modo che alla fine dell'iterazione 4 probabilmente l'80% o 90% dei requisiti sia stato scritto in modo dettagliato; solo il 10% del sistema sarà stato implementato.
  - Si noti che questo grande e dettagliato insieme di requisiti è basato su feedback ed evoluzione, ed è pertanto di qualità molto superiore rispetto a delle specifiche a cascata puramente speculative.

# Sviluppo iterativo ed evolutivo: un esempio

Un esempio di progetto iterativo (UP) composto da 20 iterazioni prima del rilascio al cliente (dal libro di testo, sezione 2.5).

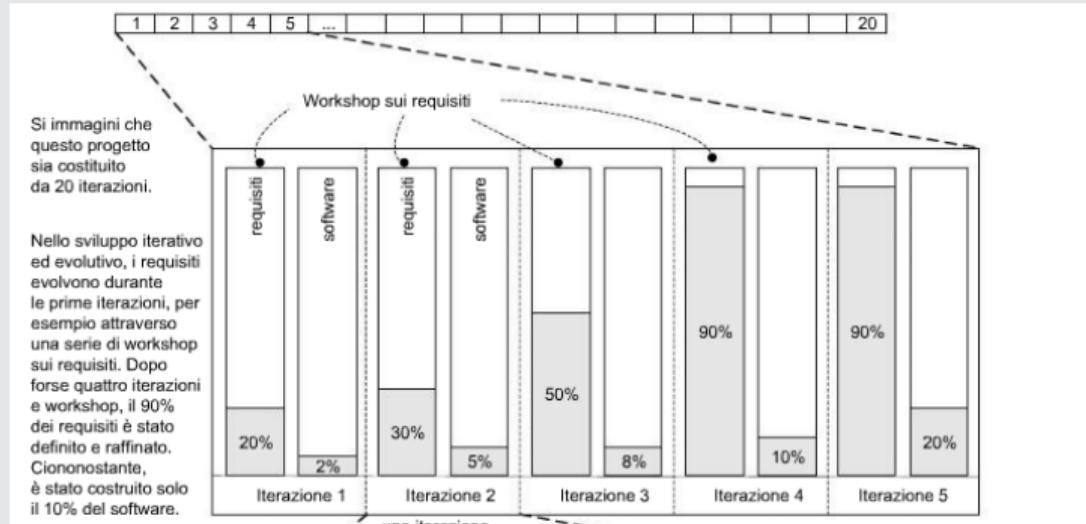
## Passi 8 e 9

8. Si è probabilmente solo al 20% della durata dell'intero progetto. In termini di UP, questa è la fine della **fase di elaborazione**. A questo punto, vengono fatte delle stime più dettagliate dei tempi e dei costi, con riferimento ai requisiti di qualità elevata in possesso. Grazie all'indagine significativa e realistica fatta, basata su programmazione, test e feedback iniziali, le stime di ciò che si può fare e di quanto tempo occorrerà sono molto più affidabili.
9. Da questo punto in poi, è poco probabile che si tengano altri workshop sui requisiti; i requisiti sono stabili, anche se non vanno mai considerati completamente congelati. Si continua con una serie di iterazioni di tre settimane, scegliendo gli obiettivi di lavoro successivi in modo adattivo in ciascuna riunione di pianifica-

# Sviluppo iterativo ed evolutivo: un esempio

Un esempio di progetto iterativo (UP) composto da 20 iterazioni prima del rilascio al cliente (dal libro di testo, sezione 2.5).

## Overview



## Pianificazione guidata dal rischio e dal cliente

Nei processi iterativi, in ogni iterazione viene stabilito il piano di lavoro per una sola iterazione, **pianificazione iterativa o adattativa**:

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

Gli obiettivi dell'iterazione non vengono cambiati: accettabile perché le iterazioni sono brevi e il feedback frequente. L'interesse è che il team di sviluppatori lavori al suo meglio, concentrandosi sul lavoro stabilito per l'iterazione.

## Pianificazione guidata dal rischio e dal cliente

La pianificazione è **guidata dal rischio e guidata dal cliente**:

- le iterazioni iniziali vengono scelte per identificare e attenuare i rischi maggiori
- per costruire e rendere visibili le caratteristiche a cui il cliente tiene di più
- stabilizzare il nucleo dell'architettura del software (è un rischio molto alto!)

# **Unified Process**

---

# Unified Process (UP)

## Unified Process (UP) - Booch, Rumbaugh, Jacobson

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

UP è flessibile e può essere applicato usando un approccio agile come **Extreme Programming (XP)** e **Scrum**.

UP non è uno standard, è un processo di sviluppo che utilizza UML. La versione commerciale (IBM-Rational) è *RUP* (Rational Unified Process).

# Unified Process (UP)

---

UP incoraggia l'uso di pratiche agili introdotte da altre metodologie:

- Iterazioni corte e timeboxed
- Raffinamento di piani, requisiti, progettazione
- Gruppi di lavoro auto-organizzati che si coordinano in riunioni regolari (da Scrum)
- Programmazione a coppie e sviluppo guidato dai test (da eXtremeProgramming)
- Modellazione *agile*: l'obiettivo è la comprensione del software piuttosto che la documentazione dello stesso

I metodi per lo **sviluppo agile** di solito applicano lo sviluppo iterativo ed evolutivo.

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

## Agile Modeling

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

Ossia, esplorare rapidamente (più rapidamente che con il codice) le alternative e il percorso verso un buon progetto OO.

# Unified Process (UP)

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

Ruoli ⇒ Attività ⇒ Artefatti

# Fasi di UP

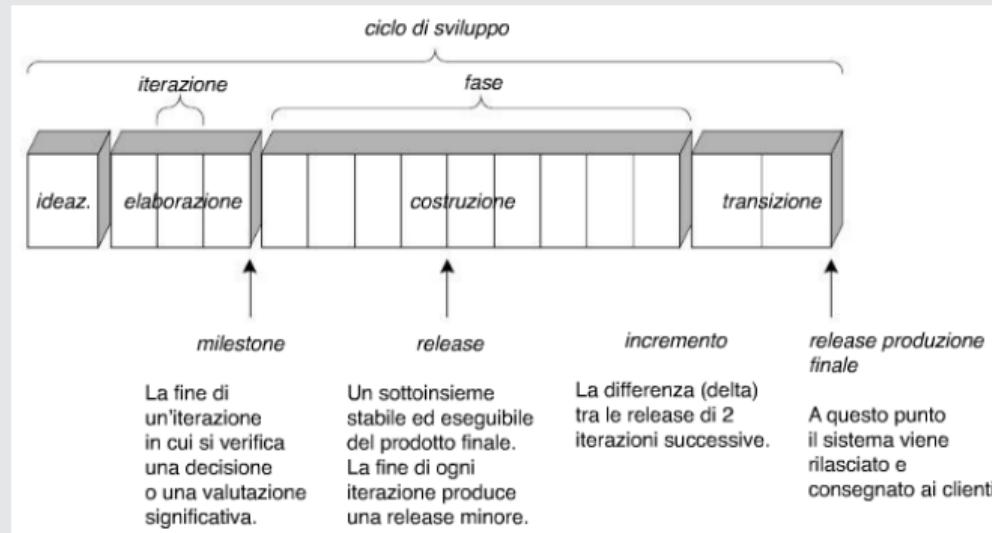
Un progetto UP organizza il lavoro e le iterazioni in **quattro fasi**:

- **Ideazione** (*inception*): visione approssimativa, studio economico, portata, stime approssimate dei costi e dei tempi  
Milestone: **obiettivi**
- **Elaborazione** (*elaboration*): Visione raffinata, implementazione iterativa del nucleo dell'architettura, risoluzione dei rischi maggiori, identificazione della maggior parte dei requisiti e della portata, stime più realistiche sulle loro inter-relazioni  
Milestone: **architetturale**
- **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**

# Fasi di UP

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

## Organizzazione temporale dei progetti in UP



## Attenzione!

- **Ideazione:** non è una fase di requisiti bensì di fattibilità, in cui viene eseguita un'indagine sufficiente per decidere di proseguire con il progetto o di interromperlo
- **Elaborazione:** non è una fase di requisiti o di progettazione bensì una fase in cui viene implementata in modo iterativo l'architettura del sistema e vengono mitigati i rischi maggiori

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

## Disciplina

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

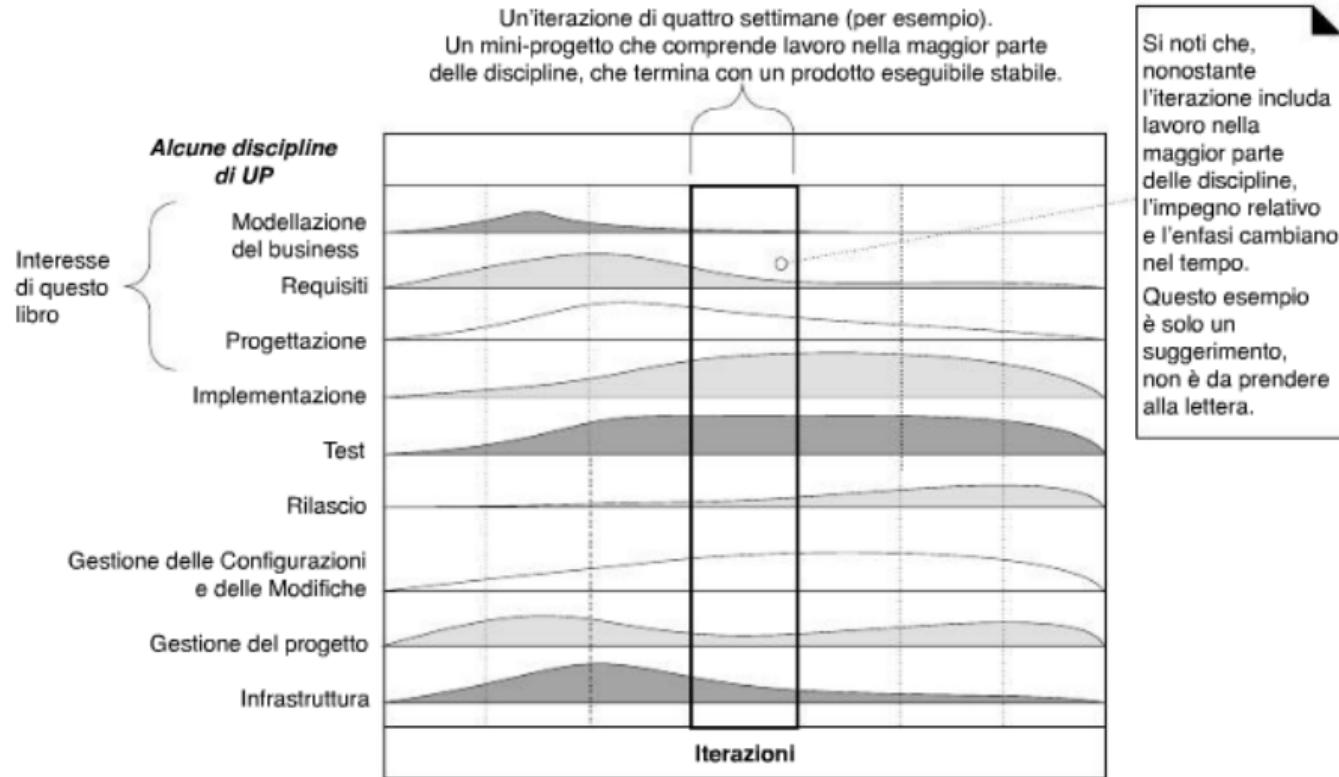
## Elaborato

Elaborato (artefatti o *work product*, *product* in RUP) è il termine generico che indica un qualsiasi prodotto di lavoro: codice, schemi di base di dati, documenti di testo, diagrammi, modelli, ecc.

- **Modellazione del business.** Attività che modellano il dominio del problema ed il suo ambito.
- **Requisiti.** Attività di raccolta dei requisiti del sistema.
- **Progettazione (*analysis and design*).** Attività di analisi dei requisiti e progetto architettonico del sistema.
- **Implementazione.** Attività di progetto dettagliato e codifica del sistema, test su componenti.
- **Test.** Attività di controllo di qualità, test di integrazione e di sistema.
- **Rilascio.** Attività di consegna e messa in opera.

- **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.

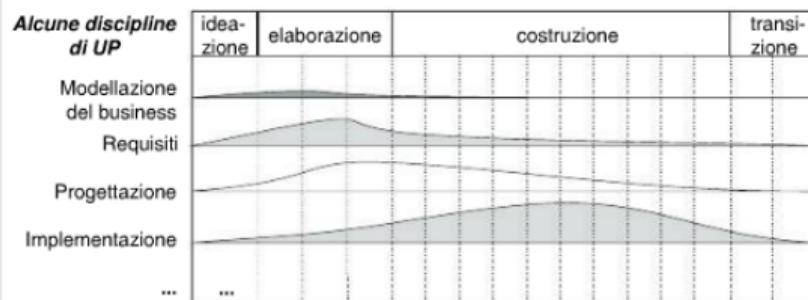
# Le discipline di UP



# Relazione tra discipline e fasi

## Attenzione!

- Le fasi sono sequenziali e la fine di una fase corrisponde ad una *milestone*
- Le discipline (tipologie di attività) non sono sequenziali e si eseguono nel progetto in ogni iterazione
- Il numero di iterazioni dipende dalla decisione del manager di progetto e dai rischi del progetto



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

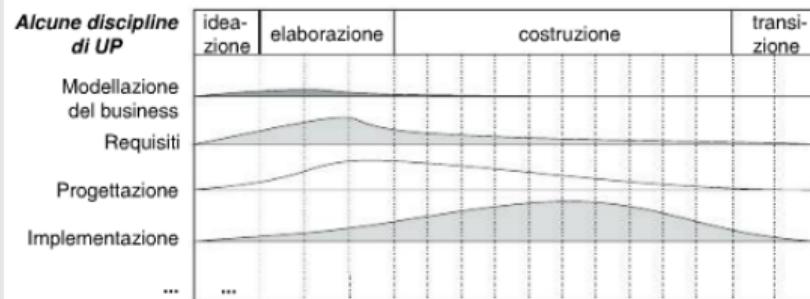
L'impegno relativo nelle discipline cambia a seconda delle fasi.

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

# Relazione tra discipline e fasi

## Attenzione!

- Le iterazioni iniziali tendono in modo naturale a dare una maggiore enfasi relativa sui requisiti e sulla progettazione, mentre quelle successive lo faranno in misura minore
- Questo perché i requisiti e il progetto si stabilizzano attraverso un processo di feedback e adattamento



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

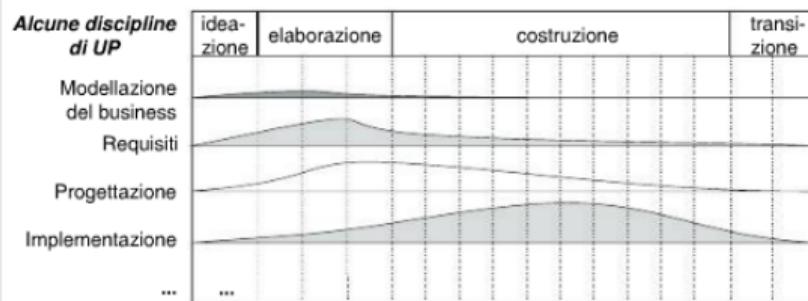
L'impegno relativo nelle discipline cambia a seconda delle fasi.

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

# Relazione tra discipline e fasi

## Attenzione!

- Durante l'elaborazione le iterazioni tendono ad avere un livello relativamente alto di lavoro sui requisiti e la progettazione, sebbene prevedano anche un certo livello di implementazione
- Durante la costruzione, l'enfasi è maggiore sull'implementazione e minore sull'analisi dei requisiti



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

L'impegno relativo nelle discipline cambia a seconda delle fasi.  
Questo esempio è solo un suggerimento, non è da prendere alla lettera.

- UP usa solo UML come linguaggio di modellazione (ad esempio, non si usano i Data Flow Diagram)
- I diagrammi UML si usano con variabilità: se un diagramma non è necessario non si usa, però tale scelta si indica esplicitamente. Bisogna **personalizzare UP**
- I diagrammi si usano in UP seguendo le caratteristiche di **iterazione** ed **incremento** (incrementi definiti su uno stesso diagramma)
- UP dice **quando** usare un diagramma

## Adattamento del processo

- In UP quasi tutto (tra artefatti e pratiche) è **opzionale**, eccetto che lo sviluppo iterativo e guidato dal rischio, la verifica continua della qualità e naturalmente il codice
- La scelta delle pratiche e artefatti UP per un progetto si riassume in un documento chiamato *scenario di sviluppo* (artefatto della disciplina Infrastruttura)

# Esempio di scenario di sviluppo

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

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

## Non si è capito lo sviluppo iterativo UP se

- Si cerca di definire tutti i requisiti del software prima di iniziare la progettazione o l'implementazione
- Si dedicano giorni o settimane a modellare con UML prima di iniziare a programmare
- Si pensa: ideazione = requisiti, elaborazione = progettazione, costruzione = implementazione (cioè, si adotta l'approccio 'a cascata')
- Si pensa che l'obiettivo dell'elaborazione sia quello di definire in maniera completa e dettagliata i modelli, che verranno tradotti in codice durante la costruzione
- Si pensa che la durata adeguata per una iterazione siano 3 mesi al posto di 3 settimane
- Si cerca di pianificare il progetto nei dettagli dall'inizio fino alla fine, e di prevedere in maniera speculativa tutte le iterazioni e cosa deve accadere in ognuna di esse

## 02 . Ideazione

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

### **Si noti che**

questi lucidi sono basati sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016” e sul materiale fornito dai docenti Viviana Bono, Claudia Picardi e Gianluca Torta dell’Università degli Studi di Torino che hanno tenuto il corso negli anni accademici precedenti.

# Table of contents

---

1. Requisiti evolutivi
2. Prima fase: ideazione

## **Requisiti evolutivi**

---

# Che cosa sono i requisiti

## Requisito

Un requisito è una capacità o una condizione a cui il sistema, e più in generale il progetto, deve essere **conforme**.

## Sorgenti dei requisiti

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

## Tipi di requisiti

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

## Requisiti in UP

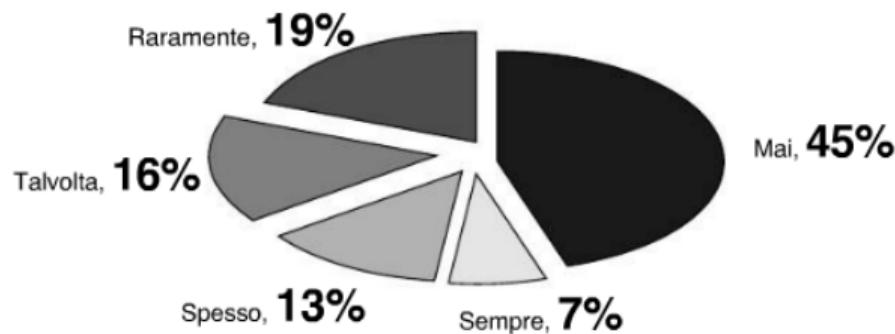
UP promuove un insieme di *best practice*, una delle quali è *gestire i requisiti*.

Nel contesto dei desiderata delle parti interessate, che sono poco chiari e che inevitabilmente cambiano, *un approccio sistematico per trovare, documentare, organizzare e tracciare i requisiti che cambiano di un sistema.*

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

## Requisiti in UP vs Requisiti 'a cascata'

Secondo uno studio, il 45% di queste caratteristiche specificate con un approccio a cascata non è mai stato utilizzato, e un altro 19% è stato raramente utilizzato.



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

## Requisiti in UP vs Requisiti 'a cascata'

UP incoraggia un'acquisizione (un approccio sistematico per trovare) dei requisiti abile, attraverso tecniche quali:

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

# Tipologie di requisiti

Modello FURPS+, acronimo per:

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

# Requisiti ed elaborati di UP

UP ha diversi elaborati (molti opzionali):

- **Modello dei Casi d'Uso:** scenari tipici dell'utilizzo di un sistema (requisiti funzionali, comportamento)
- **Specifiche Supplementari:** ciò che non rientra nei casi d'uso, requisiti non funzionali o funzionali non esprimibili attraverso casi d'uso (es. generazione di un report)
- **Glossario:** termini significativi, dizionario dei dati (requisiti relativi ai dati, regole di validazione, valori accettabili)
- **Visione:** riassume i requisiti ad alto livello, un documento sintetico per apprendere rapidamente le idee principali del progetto
- **Regole di Business:** regole di dominio, i requisiti o le politiche che trascendono un unico progetto software e a cui il sistema deve conformarsi (es. leggi fiscali dello stato)

## **Prima fase: ideazione**

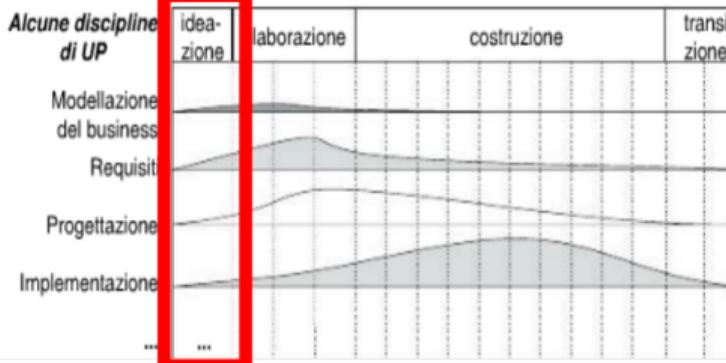
---

# UP maps

**Tabella 2.1 Scenario di Sviluppo di esempio (i – inizio; r – affinamento)**

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

*Alcune discipline di UP*



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

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

## Che cos'è l'ideazione

Permette di stabilire una visione comune e la portata del progetto (*studio di fattibilità*).

Durante l'ideazione:

- Si analizzano circa il 10% dei casi d'uso in dettaglio
- Si analizzano i requisiti non funzionali più critici
- Si realizza uno studio economico per stabilire l'ordine di grandezza del progetto e la stima dei costi
- Si prepara l'ambiente di sviluppo
- Durata: normalmente breve (primo workshop dei requisiti e pianificazione della prima iterazione dell'elaborazione)

# Che cos'è l'ideazione

## Lo scopo della fase di ideazione

non è quello di definire tutti i requisiti, né di generare una stima o un piano di progetto affidabili.

## L'ideazione

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

## Durante l'ideazione

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

# Artefatti nell'ideazione

Elaborato <sup>1</sup>	Commento
Visione e Studio economico	Describe gli obiettivi e i vincoli di alto livello, lo studio economico, e fornisce un sommario del progetto.
Modello dei Casi d'Uso	Describe i requisiti funzionali. Durante l'ideazione vengono identificati i nomi della maggior parte dei casi d'uso, e circa il 10% dei casi d'uso viene analizzato in modo dettagliato.
Specifiche supplementari	Descrivono altri requisiti, per lo più non funzionali. Durante l'ideazione è utile avere un'idea dei requisiti non funzionali fondamentali che avranno un impatto significativo sull'architettura.
Glossario	Terminologia chiave del dominio e dizionario dei dati.
Lista dei Rischi e Piano di Gestione dei Rischi	Describe i rischi (aziendali, tecnici, di risorse, di calendario) e le idee per attenuarli o rispondervi.
Prototipi e proof of concept	Per chiarire la visione e validare delle idee tecniche.
Piano dell'Iterazione	Describe che cosa fare nella prima iterazione dell'elaborazione.
Piano delle Fasi e Piano di Sviluppo del Software	Ipotesi (poco precise) riguardo alla durata e allo sforzo della fase di elaborazione. Strumenti, persone, formazione e altre risorse.
Scenario di Sviluppo	Una descrizione della personalizzazione dei passi e degli elaborati di UP per questo progetto. In UP, UP viene sempre personalizzato per il progetto.

## La documentazione non è troppa?

Lo scopo della creazione degli elaborati e dei modelli non è nel documento o nel diagramma in sé, ma nel pensare:

- Si scelgono quelli che aggiungono valore al progetto
- Si completano parzialmente
- Sono preliminari ed approssimativi

## Agile Modeling

il valore della modellazione è quello di migliorare la comprensione, anziché quello di documentare delle specifiche affidabili.

# Artefatti nell'ideazione

## La documentazione non è troppa?

Lo scopo della creazione degli elaborati e dei modelli non è nel documento o nel diagramma in sé, ma nel pensare:

- Si scelgono quelli che aggiungono valore al progetto
- Si completano parzialmente
- Sono preliminari ed approssimativi

## Agile Modeling

il valore della modellazione è quello di migliorare la comprensione, anziché quello di documentare delle specifiche affidabili.

## Generale Eisenhower

“nel prepararmi per la battaglia ho sempre trovato che i piani siano inutili, ma la pianificazione è indispensabile”.

## Le specifiche supplementari

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

---

## Specifiche Supplementari

### Cronologia revisioni

---

Versione	Data	Descrizione	Autore
Bozza ideazione	10 gen. 2031	Prima bozza. Da raffinare soprattutto durante l'elaborazione.	Craig Larman

---

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

# Specifiche supplementari: esempio dal libro di testo, Cap. 8

## Introduzione

Questo documento raccoglie tutti i requisiti POS NextGen non descritti nei casi d'uso.

## Funzionalità

(Funzionalità comuni a molti casi d'uso)

### Logging e gestione degli errori

Log (registrazione) di tutti gli errori in memoria persistente.

### Regole inseribili

In vari punti degli scenari di diversi casi d'uso (da definire) consentire la personalizzazione delle funzionalità del sistema con un insieme di regole arbitrarie eseguite in quel punto o evento.

### Sicurezza

Qualsiasi utilizzo richiede l'autenticazione dell'utente.

## Usabilità

### Fattori umani

Il cliente vede i dati presentati dal POS su un monitor grande. Pertanto:

- Il testo dovrebbe essere facilmente visibile da una distanza di 1 metro.
- Evitare i colori associati con le comuni forme di daltonismo.

Velocità, facilità ed elaborazione priva di errori sono basiliari nel processo di vendita, poiché l'acquirente vuole andarsene rapidamente, altrimenti vivrà l'esperienza dell'acquisto (e dunque il rapporto con il venditore) come meno positivi.

Il cassiere spesso guarda il cliente o gli articoli, non lo schermo del computer. Pertanto, i segnali e gli avvertimenti devono essere forniti anche tramite un suono, anziché solamente in modo grafico.

## Affidabilità

### Ripristinabilità

Se si verificano problemi nell'utilizzo di servizi esterni (autorizzazione ai pagamenti, sistema di contabilità...), cercare di risolvere il problema con una soluzione locale (per esempio, store and forward per essere in grado di completare comunque la vendita. Qui è necessaria un'analisi molto più approfondita...).

## Prestazioni

Come accennato riguardo ai fattori umani, gli acquirenti vogliono completare molto rapidamente il processo di vendita. Un collo di bottiglia è rappresentato dalle autorizzazioni ai pagamenti esterna. Il nostro obiettivo: ottenere l'autorizzazione in meno di 30 secondi, il 90% delle volte.

## Sostenibilità

### Adattabilità

I diversi acquirenti di POS NextGen hanno delle regole di business e delle esigenze di elaborazione uniche, da applicare durante l'elaborazione di una vendita. Pertanto, In diversi punti definiti nello scenario (per esempio, quando viene iniziata una nuova vendita o quando viene aggiunto un nuovo articolo alla vendita) saranno abilitate delle regole di business inseribili.

### Configurabilità

I diversi acquirenti desiderano differenti configurazioni di rete per i loro sistemi POS, come thick client o thin client, a 2-level oppure a N-level fisici, e così via. Inoltre vogliono avere la possibilità di modificare queste configurazioni, in base a cambiamenti del loro business e ad esigenze per le prestazioni. Pertanto il sistema dovrà essere in un certo modo configurabile per soddisfare queste necessità. In questo settore è necessaria un'analisi molto più approfondita, per scoprire le aree e il grado di flessibilità e gli sforzi per ottenerla.

## Vincoli di implementazione

I leader di NextGen insistono su una soluzione basata su tecnologie Java, perché sostengono che ciò migliorerà la portabilità e la sostenibilità a lungo termine, offre alla facilità di sviluppo.

## Componenti acquistati

- Calcolatore delle imposte. È necessario supportare l'uso di calcolatori differenti e per diversi paesi.

# Specifiche supplementari: esempio dal libro di testo, Cap. 8

## Componenti open source

In generale per questo progetto si consiglia di massimizzare l'utilizzo di componenti open source e gratuiti con tecnologia Java.

Anche se è prematuro progettare e scegliere definitivamente i componenti, si consigliano i seguenti come probabili candidati:

- framework di logging JLog
- ...

## Interfacce

### Hardware e interfacce significative

- Monitor touch screen (venerà considerato dai sistemi operativi come un monitor normale, e i gesti di sfioramento come eventi del mouse)
- Lettore laser di codice a barre (normalmente sono collegati a una tastiera speciale, e l'input della scansione è considerata nel software come dei tasti premuti)
- Stampante per le ricevute
- Lettore di carte di credito/bancomat
- Lettore di firme (ma non nella versione 1)

### Interfacce software

Per la maggior parte dei sistemi di collaborazione esterni (calcolatore delle imposte, contabilità, inventario...) occorre essere in grado di connettersi a vari sistemi, quindi a interfacce diversificate.

## Aspetti legali

Si consiglia l'uso di alcuni componenti open source, se le loro restrizioni di licenza possono essere risolte in modo da consentire la rivendita di prodotti che comprendono software open source.

Per legge, tutte le regole fiscali vanno applicate durante le vendite. Si tenga presente che queste regole possono variare di frequente.

## Regole di dominio (di business) specifiche dell'applicazione

(Vedere il documento Regole di Business separato per le regole generali.)

ID	Regola	Modificabilità	Sorgente
R1	Regole di sconto per l'acquirente. Esempi: Dipendente – sconto del 20%. Clienti abituali – sconto del 10%. Anziano – sconto del 15%.	Elevata. Ciascun negoziante utilizza regole diverse.	Politica del negoziante.
R2	Regole di sconto vendita (a livello di transazione). Si applica al totale, al lordo delle imposte. Esempi: 10% di sconto se il totale è superiore a \$100. 5% di sconto il lunedì. 10% di sconto su tutte le vendite dalle 10:00 alle 15:00 di oggi. 50% di sconto sul tutto dalle 9:00 alle 10:00 di oggi.	Elevata. Ciascun negoziante utilizza regole diverse, che possono cambiare ogni giorno o ogni ora.	Politica del negoziante.
R3	Regole di sconto prodotto (a livello di riga di vendita per articolo). Esempi: 10% di sconto sui trattori questa settimana. Ogni 2 hamburger vegetariani, se ne ottiene 1 gratis.	Elevata. Ciascun negoziante utilizza regole diverse, che possono cambiare ogni giorno o ogni ora.	Politica del negoziante.

# Specifiche supplementari: esempio dal libro di testo, Cap. 8

## Informazioni nei domini di interesse

### Determinazione del prezzo

Oltre alle regole di determinazione del prezzo descritte nella sezione delle regole di dominio, si noti che i prodotti hanno un *prezzo originale* e, optionalmente, un *prezzo ribassato permanente*. Il prezzo di un prodotto (prima di ulteriori sconti) è il prezzo ribassato permanente, se presente. Le aziende memorizzano il prezzo originale anche se esiste un prezzo ribassato permanente, per motivi di contabilità e fiscali.

### Gestione dei pagamenti con carta di credito e bancomat

Quando un pagamento elettronico con carta di credito o bancomat è approvato da un servizio di autorizzazione ai pagamenti, è quest'ultimo il responsabile del pagamento al venditore, e non l'acquirente. Di conseguenza, per ciascun pagamento il venditore deve registrare il denaro a lui dovuto tra i crediti da riscuotere dal servizio di autorizzazione. Questo servizio, di solito ogni notte, esegue un trasferimento elettronico di fondi sul conto del venditore pari all'importo giornaliero totale dovuto, meno una (piccola) commissione per ogni operazione, addebitata per il servizio.

### Imposte sulle vendite

Il calcolo delle imposte sulle vendite può essere molto complesso, anche perché cambia regolarmente in conformità alle leggi, a tutti i livelli (federale, statale, locale e così via). È pertanto consigliabile delegare il calcolo delle imposte a un calcolatore software prodotto da terzi (ce ne sono diversi a disposizione). Le imposte possono essere dovute a enti cittadini, provinciali, regionali e nazionali. Alcuni articoli possono essere esenti da imposte in modo incondizionato, oppure essere esenti da imposte a seconda dell'acquirente o del beneficiario (per esempio, un bambino).

### Codici identificativi degli articoli: codici UPC, EAN, SKU e codici a barre e lettori di codici a barre

Il sistema POS NextGen deve supportare diversi schemi di identificazione degli articoli. UPC (*Universal Product Codes*), EAN (*European Article Numbering*) e SKU (*Stock Keeping Units*) sono tre sistemi di identificazione diffusi per i prodotti venduti. JAN (*Japanese Article Numbers*) è un tipo di versione EAN. Gli SKU sono codici identificativi definiti dal negoziante in modo completamente arbitrario. Tuttavia, UPC ed EAN sono degli standard e hanno degli enti di controllo. Si consulti [www.adams1.com/upccode.html](http://www.adams1.com/upccode.html) per una panoramica generale. Si consultino inoltre [www.gs1.org](http://www.gs1.org) e [www.gs1us.org](http://www.gs1us.org).

Il documento **Visione** riassume alcune delle informazioni contenute del *modello dei casi d'uso* e nelle *specifiche supplementari*.

Inoltre, descrive brevemente il progetto, come contesto per i partecipanti, al fine di stabilire una visione comune del progetto.

- Obiettivi e problemi fondamentali ad alto livello delle parti interessate (utile soprattutto per i requisiti non funzionali)
- Riepilogo delle caratteristiche di sistema (usualmente non più di 10 per essere compreso facilmente, si considerino raggruppamenti e astrazioni)

# Visione: esempio dal libro di testo, Cap. 8

Visione			
Cronologia revisioni			
Versione	Data	Descrizione	Autore
Bozza ideazione	10 gen. 2031	Prima bozza. Da raffinare soprattutto durante l'elaborazione.	Craig Larman

## Introduzione

Prevediamo la realizzazione di un'applicazione point-of-sale (POS) di prossima generazione, tollerante ai guasti, chiamata NextGen POS, dotata della flessibilità per supportare le regole di business diversificate e variabili del cliente, terminali e meccanismi di interfaccia utente multipli e l'integrazione con più sistemi di supporto prodotti da terze parti.

## Posizionamento

### Opportunità di business

I prodotti POS esistenti non sono adattabili all'attività del cliente, in termini di regole di business variabili e progetto di rete (per esempio, thin client o meno; architetture a 2, 3 o 4 livelli). Inoltre non scalano bene rispetto alla crescita del numero di terminali o del volume di affari. Nessuno di essi può lavorare sia in modalità online che offline, con un adattamento dinamico in base ai fallimenti. Nessuno dei POS esistenti si integra facilmente con molti dei sistemi prodotti da terze parti. Nessuno consente l'integrazione con nuove tecnologie per i terminali, quali i PDA mobili. Il mercato è insoddisfatto rispetto a questa situazione poco flessibile, ed emerge la richiesta di un POS che risolva il problema.

### Formulazione del problema

I sistemi POS tradizionali sono poco flessibili, poco tolleranti ai guasti e difficili da integrare con sistemi prodotti da terze parti. Ciò provoca problemi (tra l'altro) nell'elaborazione tempestiva delle vendite, nell'istituzione di processi migliorati che non corrispondono al software, e nella precisione e nell'aggiornamento dei dati di contabilità e inventario a supporto della misurazione e della pianificazione. Questo influisce sui cassieri, i gestori del negozio, gli amministratori di sistema e il management aziendale.

### Formulazione della posizione del prodotto

– Riepilogo conciso del destinatario del sistema, delle sue caratteristiche principali e di ciò che lo differenzia dalla concorrenza.

### Alternative e concorrenza...

### Descrizioni delle parti interessati

### Dati demografici di mercato...

# Visione: esempio dal libro di testo, Cap. 8

## Riepilogo delle parti interessate (non utenti)...

### Riepilogo dell'utente...

#### Obiettivi e problemi fondamentali ad alto livello delle parti interessate

Un workshop sui requisiti di un giorno con esperti del settore e altre parti interessate, nonché sondaggi presso diversi punti vendita, hanno portato all'identificazione dei seguenti obiettivi e problemi fondamentali.

Obiettivo ad alto livello	Priorità	Problemi	Soluzioni attuali
Elaborazione delle vendite rapida, robusta, integrata	Alta	Velocità ridotta all'aumentare del carico. Perdite nella capacità di elaborare le vendite se i componenti falliscono. Mancanza di informazioni aggiornate e precise dalla contabilità e da altri sistemi causa della mancata integrazione con i sistemi esistenti di contabilità, inventario e personale. Provoca difficoltà nella misurazione e nella pianificazione. Incapacità di personalizzare le regole di business in base a requisiti di business unici. Difficoltà nell'aggiungere nuovi tipi di terminali o interfacce utente (per esempio PDA mobili).	I prodotti POS esistenti forniscono un'elaborazione di base delle vendite, ma non risolvono questi problemi.
...	...	...	...

### Obiettivi a livello dell'utente

Gli utenti (e i sistemi esterni) necessitano di un sistema che soddisfi i seguenti obiettivi.

- *Cassiere*: elaborare le vendite, gestire i resi, cash in, cash out.
- *Amministratore del Sistema*: gestire gli utenti, gestire la sicurezza, gestire le tabelle di sistema.
- *Direttore*: avviare il sistema, arrestare il sistema.
- *Sistema delle attività di vendita*: analizzare i dati delle vendite.
- ...

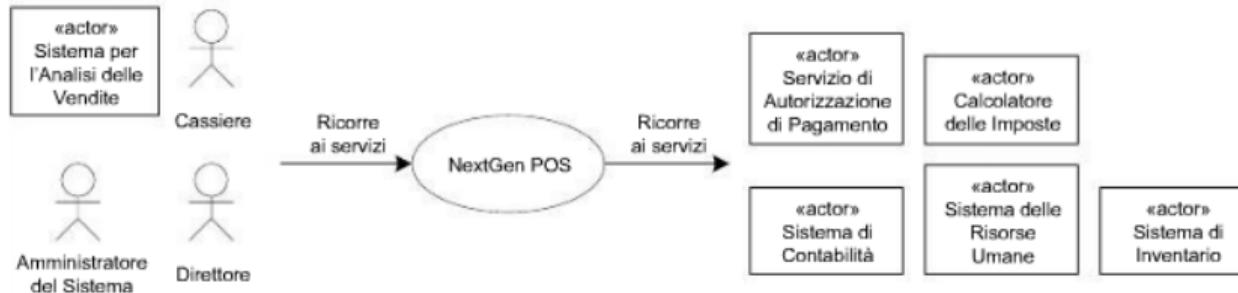
### Ambiente dell'utente...

#### Descrizione generale del prodotto

##### Punto di vista del prodotto

Il POS NextGen solitamente sarà installato presso i negozi; se sono utilizzati dei terminali mobili, saranno in stretta prossimità della rete aziendale, all'interno o nelle immediate vicinanze all'esterno. Fornirà servizi agli utenti e collaborerà con altri sistemi, come indicato nella Figura Visione -1.

# Visione: esempio dal libro di testo, Cap. 8



**Figura Visione -1** Diagrammi di contesto del sistema POS NextGen.

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

# Visione: esempio dal libro di testo, Cap. 8

## Riepilogo dei vantaggi

Caratteristica di supporto	Vantaggi per le parti interessate
A livello di funzionalità, il sistema fornirà tutti i normali servizi richiesti da un'organizzazione di vendita, tra cui l'acquisizione delle vendite, le autorizzazioni ai pagamenti, la gestione dei resi e così via.	Servizi POS rapidi e automatizzati.
Rilevamento automatico dei fallimenti, passaggio all'elaborazione locale offline per i servizi non disponibili.	L'elaborazione delle vendite non viene interrotta quando si verifica un fallimento dei componenti esterni.
Regole di business inseribili in vari punti degli scenari durante l'elaborazione delle vendite.	Configurazione flessibile della logica applicativa.
Operazioni in tempo reale con sistemi prodotti da terze parti, utilizzando dei protocolli standard.	Informazioni aggiornate e precise su vendite, contabilità e inventario, a supporto della misurazione e della pianificazione.
...	...

Ipotesi e dipendenze...

Costo determinazione del prezzo...

Licenze e installazione...

## Riepilogo delle caratteristiche del sistema

- Acquisizione delle vendite.
- Autorizzazioni ai pagamenti (carte di credito, bancomat, assegni).
- Amministrazione di sistema per utenti, sicurezza, tabelle di codici e costanti, e così via.
- Elaborazione automatica offline delle vendite in caso di fallimenti dei componenti esterni.
- Operazioni in tempo reale, basate sui protocolli standard, con sistemi di terze parti, compreso inventario, contabilità, personale, calcolatori delle imposte e servizi di autorizzazione ai pagamenti.
- Definizione ed esecuzione di regole di business personalizzate "inseribili" in punti fissi comuni degli scenari di elaborazione.
- ...

## Altri requisiti e vincoli

Per i vincoli di progettazione, usabilità, affidabilità, prestazioni, sostenibilità, vincoli di progettazione, documentazione, packaging e così via: vedere le Specifiche Supplementari e i Casi d'uso.

# Glossario e dizionario dei dati

Il documento **Glossario** è un elenco dei termini significativi e delle relative definizioni. Include pseudonimi e termini composti.

## Perché è importante?

È comune che ci siano molti termini, spesso tecnici o specifici di dominio, che vengono usati in modi diversi. L'obiettivo è eliminare le discrepanze per ridurre i problemi di comunicazione e di ambiguità dei requisiti.

È spesso utile iniziare da questo documento.

# Glossario e dizionario dei dati

Il documento **Glossario** è un elenco dei termini significativi e delle relative definizioni. Include pseudonimi e termini composti.

## Perché è importante?

È comune che ci siano molti termini, spesso tecnici o specifici di dominio, che vengono usati in modi diversi. L'obiettivo è eliminare le discrepanze per ridurre i problemi di comunicazione e di ambiguità dei requisiti.

È spesso utile iniziare da questo documento.

## Dizionario dei dati

In UP, il Glossario svolge anche il ruolo di un dizionario di dati, un documento dati relativi ad altri dati, ovvero metadati.

⇒ Regole di validazione.

# Glossario: esempio dal libro di testo, Cap. 8

## Glossario

### Cronologia revisioni

Versione	Data	Descrizione	Autore
Bozza ideazione	10 gen. 2031	Prima bozza. Da raffinare soprattutto durante l'elaborazione.	Craig Larman

### Definizioni

Termine	Definizione e informazioni	Formato	Regole di validazione	Anche detto
articolo	Un prodotto o un servizio in vendita			
autorizzazione di pagamento	Validazione da parte di un servizio esterno di autorizzazione ai pagamenti che effettuerà o garantirà il pagamento al venditore.			

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

## Glossario: esempio dal libro di testo, Cap. 8

richiesta di autorizzazione al pagamento	Un insieme di elementi inviati elettronicamente a un servizio di autorizzazione, solitamente come array di caratteri. Gli elementi comprendono: ID del negozio, numero di conto del cliente, importo, data e ora.			
UPC	Codice numerico che identifica un prodotto. Solitamente rappresentato da un codice a barre applicato ai prodotti. Si consulti <a href="http://www.gs1.org">www.gs1.org</a> per i dettagli sul formato e la validazione.	Codice di 12 cifre, con diverse parti secondarie.	La cifra 12 è una cifra di controllo.	Universal Product Code (codice universale prodotto)
...	...			

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

## |Glossario

Termine	Descrizione	Sinonimi
<b>Sezione</b>		

# Glossario: esempio dal laboratorio a.a. 2017/2018

## Glossario

Termine	Descrizione	Sinonimi
<b>Parte 1: Configurazione del ristorante</b>		
Abilitazione degli utenti	Rendere accessibile l'applicazione ad una persona	
Apertura	Orario in cui il ristorante è aperto al pubblico	Servizio
Assegnazione dei ruoli	Dire a quali ruoli appartiene un certo utente	
Chef	Lo chef che gestisce il ristorante giorno per giorno	Capocuoco
Cuoco	Una delle persone che cucina nel ristorante	
Executive chef	Lo chef responsabile della proposta gastronomica del ristorante	
Manager	la persona che gestisce l'applicazione	
Personale di Sala	Le persone (maitre e camerieri) che si occupano di interagire con i clienti	
Preparazione della linea	Fase in cui il ristorante è chiuso e i cuochi si dedicano a preparare il necessario in vista dell'apertura	
Ruolo	caratterizzazione di un utente che definisce le funzionalità a lui accessibili. I ruoli disponibili in CookALot sono: Manager, Executive Chef, Chef, Cuoco, Personale di Sala	
Turno	Periodo di attività di una persona con un orario di inizio e un orario di fine, può essere "di apertura" o "di preparazione della linea".	
<b>Parte 2: Il ricettario e i menu</b>		
Autore (di una ricetta)	Chi ha ideato la ricetta	

## Regole di dominio

---

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

# Regole di domino: esempio dal libro di testo, Cap. 8

## Regole di dominio

### Cronologia revisioni

Versione	Data	Descrizione	Autore
Bozza ideazione	10 gen. 2031	Prima bozza. Da raffinare soprattutto durante l'elaborazione.	Craig Larman

### Elenco di regole

(Si veda anche il documento separato delle Regole specifiche per l'applicazione nelle Specifiche Supplementari.)

ID	Regole	Modificabilità	Sorgente
R1	Per pagamenti con carta di credito è richiesta la firma.	La "firma" dell'acquirente continua a essere richiesta, ma entro 2 anni la maggior parte dei nostri clienti richiederà l'acquisizione della firma su dispositivo digitale, ed entro 5 anni si prevede che ci sarà una richiesta di supporto della nuova "firma" con codice digitale univoco attualmente supportata dalle leggi statunitensi.	La politica di quasi tutte le società di autorizzazione crediti.
R2	Regole fiscali sulle imposte. Le vendite richiedono l'aggiunta di imposte. Vedere leggi e regolamenti governativi per i dettagli attuali.	Elevata. Le leggi fiscali cambiano di anno in anno, a tutti i livelli di governo.	Leggi
R3	Lo storno dei pagamenti con carta di credito può essere pagato solo come credito sulla carta di credito dell'acquirente, non in contanti.	Bassa	Politica della società di autorizzazioni di credito

## Elaborati: tempi di realizzazione

Durante l'ideazione, le parti interessate devono decidere se il progetto merita un'indagine seria: la Visione, le Specifiche Supplementari, il Glossario e le Regole di Business hanno questa funzione ma il loro sviluppo completo può avvenire solo durante l'elaborazione.

**Tabella 8.1** Esempio di tempificazione per alcuni elaborati di UP. i - inizio; r - raffinamento

Disciplina	Elaborato Iterazione →	Ideaz. I1	Elab. E1..En	Costr. C1..Cn	Trans. T1..T2
Modellazione del business	Modello di dominio		i		
Requisiti	Modello dei Casi d'Uso (SSD)	i	r		
	<b>Visione</b>	i	r		
	<b>Specifiche Supplementari</b>	i	r		
	<b>Glossario</b>	i	r		
	<b>Regole di business</b>	i	r		
Progettazione	Modello di progetto		i	r	
	Documento dell'Architettura				
	Software		i		
	Modello dei Dati		i	r	

## Elaborati: tempi di realizzazione

Durante l'**ideazione** i documenti vengono abbozzati e definiti in maniera “leggera”.

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

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

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

# Scenario di sviluppo

La scelta delle pratiche e degli elaborati UP per un progetto può essere scritta nel documento chiamato **Scenario di Sviluppo** (disciplina *Infrastruttura*).

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

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

# Scenario di sviluppo

La scelta delle pratiche e degli elaborati UP per un progetto può essere scritta nel documento chiamato **Scenario di Sviluppo** (disciplina *Infrastruttura*).

Scenario di sviluppo								
periodo	Ideazione	Elaborazione				Costruzione		
	iterazione1	iterazione1	iterazione2	iterazione 3	iterazione 4	iterazione 1	iterazione 2	
<b>data inizio</b>	06-mar	14-mar	22-mar	10-apr	02-mag	16-mag	30-mag	
<b>data fine</b>	14-mar	22-mar	10-apr	02-mag	16-mag	...esame	...esame	
<b>Disciplina</b>	Elaborati							
<b>Requisiti</b>	Glossario	Glossario Attori/obiettivi/UC brevi						
	Visione							
	Specifiche supplementari	UC-Diagram						
	Modello dei casi d'uso	UC dettagliati S#1	UC dettagliati S#2	UC dettagliati S#3	END	END	END	END
<b>Modellazione del business</b>	<b>Modello di dominio</b>	NO	SSD+ bozza MDom S#1	SSD+ bozza MDom S#2	SSD + bozza MDom S#3	END		
			Contratti + MDom S#1	Contratti + MDom completo S#2	Contratti + MDom completo S#3	END	END	
<b>Progettazione</b>	<b>Modello di progetto</b>	NO	NO	NO	DDC + DSD + test S#1	DDC + DSD + test S#2	DDC + DSD + test S#3	END
<b>Implementazione</b>	<b>Codifica</b>				CODE logic + GUI S#1	CODE logic + GUI S#2	CODE logic S#3	
	<b>Testing</b>	NO	NO	NO	NO			

Dal progetto di laboratorio 2017/2018.

## 03 . Casi d'uso

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

### **Si noti che**

questi lucidi sono basati sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016” e sul materiale fornito dai docenti Viviana Bono, Claudia Picardi e Gianluca Torta dell’Università degli Studi di Torino che hanno tenuto il corso negli anni accademici precedenti.

# Table of contents

---

1. Disciplina dei requisiti
2. Casi d'uso
3. POS NextGen: esempio di caso d'uso breve, informale e dettagliato
4. Come scrivere un caso d'uso

## **Disciplina dei requisiti**

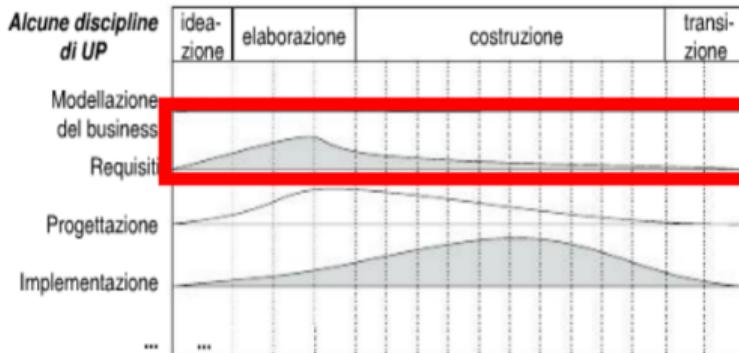
---

# UP maps

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

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

## Alcune discipline di UP



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

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

# Disciplina dei requisiti

## Disiplina dei requisiti

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

## Requisiti di sistema

Capacità e condizioni alle quali il sistema deve essere conforme, scritti nel "linguaggio" del committente.

# Disciplina dei requisiti

## Disiplina dei requisiti

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

## Requisiti di sistema

Capacità e condizioni alle quali il sistema deve essere conforme, scritti nel “linguaggio” del committente.

## Flusso delle attività in UP

Passi principali (non necessariamente eseguiti separatamente):

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

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

Ogni requisito è caratterizzato:

- **breve descrizione**
- **stato** (es. proposto, approvato, incorporato, validato)
- **costi di implementazione stimato**
- **priorità**
- **rischio associato per la sua implementazione**

## **La lista dei requisiti**

è usata anche per stimare la taglia del progetto e per decidere come suddividere il progetto in sequenze di iterazioni

# Capire il contesto del sistema

Due approcci:

- **Modellazione del dominio**

## Modello di dominio

Describe i concetti importanti del sistema come oggetti di dominio e relaziona i concetti con associazioni

# Capire il contesto del sistema

Due approcci:

- **Modellazione del dominio**
- **Modellazione del business**

## Modello di dominio

Describe i concetti importanti del sistema come oggetti di dominio e relaziona i concetti con associazioni

## Modello di business

- è un super-insieme del modello di dominio, descrive i processi di business (collection of related, structured activities of task that produce a specific service or product, serve a particular goal, for a particular customer)
- è un prodotto dell'ingegneria del business
- ha lo scopo di migliorare i processi di business

## Catturare i requisiti funzionali

---

- In UP vengono usati i **casi d'uso**
- Un caso d'uso rappresenta **una maniera di utilizzare il sistema** da parte di un utente
- Sono **descrizioni testuali**

## Catturare i requisiti non funzionali

- Possono essere inclusi nei casi d'uso se relazionati con il requisito funzionale descritto dal caso d'uso
- Altrimenti, vengono descritti nelle **Specifiche Supplementari**

## Approccio UP-Agile [Larman]

- I requisiti funzionali sono catturati con i **casi d'uso** (UC)
- Se ci sono requisiti non funzionali relazionati, questi vengono inclusi nel caso d'uso
- I requisiti non funzionali “generali” sono inclusi del documento di **Specifiche Supplementari** (SS)
- Il contesto del sistema è catturato dal **diagramma UML dei casi d'uso**
- Tali artefatti (UC & SS) costituiscono l'**input** per definire il **modello di dominio**
- Non viene considerato il modello di business

## Casi d'uso

---

UP è una metodologia “**use-case driven**”

- I requisiti funzionali si descrivono con casi'uso
- I casi d'uso si usano per pianificare le iterazioni
- L'analisi e la progettazione si basano sulla realizzazione di casi d'uso
- I test si basano sui casi d'uso realizzati
- I casi d'uso influiscono nella redazione dei manuali utente e nella definizione della visione del progetto

- Sono descrizioni (testuali) di scenari di uso interessanti del sistema software che si deve realizzare
- **Attori:** qualcosa o qualcuno dotato di un comportamento
- **Scenario** (o istanza di caso d'uso): sequenza specifica di azioni ed interazioni tra il sistema e alcuni attori. Descrive una particolare storia nell'uso del sistema, ovvero un percorso attraverso il caso d'uso
- **Caso d'uso** (o *casi di utilizzo*): una collezione di scenari correlati (di successo e di fallimento) che descrivono un attore che usa il sistema per raggiungere un obiettivo specifico

## 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, ...

## Attenzione!!

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

### 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, ...

- **Modello dei casi d'uso:** modello delle funzionalità del sistema
- Include un diagramma UML dei casi d'uso che serve come modello di contesto del sistema e come indice dei nomi di caso d'uso
- I casi d'uso non sono una pratica di OOA/D classica, però sono utili a rappresentare i requisiti come input all'OOA/D. In altre parole, i casi d'uso non sono orientati agli oggetti
- I casi d'uso definiscono i contratti in relazione al comportamento del sistema

**Nota:** *il mancato coinvolgimento dell'utente nei progetti software è quasi al primo posto tra i motivi di fallimento dei progetti, e quindi è decisamente opportuno utilizzare tutto ciò che può contribuire al loro coinvolgimento [Larman, 1993].*

## Enfasi sull'utente e non solo sul sistema:

- I casi d'uso mettono in risalto gli obiettivi degli utenti e il loro punto di vista
- Chi utilizza il sistema?
- Quali sono i loro scenari d'uso tipici?
- Quali sono i loro obiettivi?

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

Non sono caratteristiche del sistema! **NO** “Il sistema dovrà fare...”

# Attori e tipi di attori

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

Nota: il sistema stesso è considerato un attore.

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

- **Attore primario:**

- raggiunge gli obiettivi utente utilizzando i servizi del sistema
- utile per trovare gli obiettivi utente

- **Di supporto:**

- offre un servizio al sistema
- utile per chiarire le interfacce esterne e i protocolli

- **Fuori scena:**

- ha un interesse nel comportamento del caso d'uso
- utile per garantire che tutti gli interessi necessari vengano soddisfatti

# Formati del caso d'uso (UC)

Possono essere di tre formati diversi.

- **Formato breve:** riepilogo conciso di un solo paragrafo, relativo (normalmente) al solo scenario principale di successo  
Serve a capire rapidamente l'argomento e la portata
- **Formato informale:** più paragrafi, scritti in modo informale, relativi a vari scenari  
Stessa funzione del formato breve ma con maggiore dettaglio
- **Formato dettagliato:** tutti i passi e le variazioni sono scritti in dettaglio, include *pre-condizioni* e *garanzie di successo*  
Si scrivono a partire dal formato breve o informale

Durante l'ideazione si scrivono circa il 10% dei casi d'uso tra i più critici in formato dettagliato utilizzando *template* appositi (*caso d'uso dettagliato e strutturato*).

## **POS NextGen: esempio di caso d'uso breve, informale e dettagliato**

---

# Esempio caso d'uso breve

## Elabora vendita

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

# Esempio caso d'uso informale

## Gestire restituzioni

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

# Sezioni del template del caso d'uso dettagliato

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

# Esempio caso d'uso dettagliato

## (UC1) Elabora Vendita (Process Sale)

Si veda libro di testo di C. Larman (**quarta edizione**), Cap. 7, Sezione 7.8 “*Esempio: elabora Vendita, stile dettagliato*”, pag. 77–81.

Si veda anche “Il sistema POS NextGen”, libro di testo di C. Larman, Cap. 4, Sezione 4.1, pag. 50.

## (UC1) Elabora Vendita (Process Sale)

Si veda libro di testo di C. Larman (**quinta edizione**), Cap. 7, Sezione 7.8 “*Esempio: elabora Vendita, stile dettagliato*”, pag. 72–76.

Si veda anche “Il sistema POS NextGen”, libro di testo di C. Larman, Cap. 4, Sezione 4.1, pag. 46.

# Esempio caso d'uso dettagliato

## 4.1 Caso uno: il sistema POS NextGen

Il primo studio di caso è il sistema POS (*point-of-sale*, punto di vendita) NextGen. Un sistema POS è un'applicazione software utilizzata, fra l'altro, per registrare le vendite e gestire i pagamenti; è normalmente presente nei negozi e nei supermercati. In questo dominio apparentemente semplice si vedrà che ci sono problemi di requisiti, analisi e progettazione interessanti da risolvere. Inoltre si tratta di un problema reale; i sistemi POS vengono veramente sviluppati con tecnologie a oggetti.



Un sistema POS comprende componenti hardware, come un computer e un lettore di codici a barre, nonché del software. Si interfaccia a varie applicazioni di servizio, come un sistema per l'inventario o per la contabilità, che possono essere realizzati da terzi. Un sistema POS deve essere relativamente tollerante ai guasti; ciò significa che, anche se i servizi remoti sono temporaneamente non disponibili (come il sistema di inventario), il sistema deve

essere comunque in grado di gestire le vendite, permettendo almeno il pagamento in contanti, in modo che l'attività di vendita non venga interrotta e danneggiata.

Un sistema POS deve essere in grado sempre più di supportare terminali e interfacce lato client multiple e diversificate. Fra queste, un browser web usato come terminale thin-client, un normale personal computer con un'interfaccia grafica utente tipo Java Swing oppure FX, dispositivi touch screen, PDA wireless e così via.

Inoltre il POS da realizzare è un sistema commerciale, destinato a essere venduto a diversi clienti, con esigenze differenti rispetto alla gestione delle regole di business. Ciascun cliente vorrà un proprio insieme unico di logica da eseguire in alcuni punti prevedibili negli scenari di utilizzo del sistema, come quando viene iniziata una nuova vendita o viene aggiunto un nuovo articolo a una vendita. Pertanto sarà necessario un meccanismo per fornire questa flessibilità e personalizzazione.

Utilizzando una strategia di sviluppo iterativo, si procederà attraverso i requisiti, l'analisi e la progettazione orientata agli oggetti e l'implementazione.

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

---

**Portata:** Applicazione POS NextGen

**Livello:** Obiettivo utente

**Attore primario:** Cassiere

**Attore finale:** Cliente

**Parti interessate e Interessi**

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

**Pre-condizioni:** Il Cassiere è identificato e autenticato.

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

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

1. Il Cliente arriva alla cassa POS con gli articoli e/o i servizi da acquistare.
2. Il Cassiere inizia una nuova vendita.
3. Il Cassiere inserisce il codice identificativo di un articolo.
4. Il Sistema registra la riga di vendita per l'articolo e mostra la descrizione dell'articolo, il suo prezzo, il totale parziale. Il prezzo è calcolato in base a un insieme di regole di prezzo.
- Il Cassiere ripete i passi 3-4 fino a che non indica che ha terminato.*
5. Il Sistema mostra il totale con le imposte calcolate.
6. Il Cassiere riferisce il totale al Cliente, e richiede il pagamento.
7. Il Cliente paga e il Sistema gestisce il pagamento.
8. Il Sistema registra la vendita completata e invia informazioni sulla vendita e sul pagamento ai sistemi esterni di Contabilità (per la contabilità e le commissioni) e di Inventario (per l'aggiornamento dell'inventario).
9. Il Sistema genera la ricevuta.
10. Il Cliente va via con la ricevuta e gli articoli acquistati.

**Estensioni (o Flussi alternativi):**

- \*a. In qualsiasi momento, il Direttore chiede di eseguire un'operazione di sovrascrittura:
  - 1. Il Sistema passa alla modalità autorizzata "Direttore".
  - 2. Il Direttore o il Cassiere esegue un'operazione nella modalità "Direttore"; per esempio, riprendere una vendita sospesa su un altro registratore di cassa, annullare una vendita, cambiare la chiusura di cassa, e così via.
  - 3. Il Sistema torna alla modalità autorizzata "Cassiere".
- \*b. In qualsiasi momento, il Sistema fallisce: Per consentire il ripristino e una gestione corretta della contabilità, bisogna garantire che tutto lo stato transazionale significativo possa essere ripristinato, a partire da qualsiasi passo dello scenario.
  - 1. Il Cassiere riavvia il Sistema, si autentica, e richiede il ripristino dello stato precedente.
  - 2. Il Sistema ricostruisce lo stato precedente.
  - 2a. Il Sistema rileva delle anomalie che impediscono il ripristino:
    - 1. Il Sistema segnala un errore al Cassiere, registra l'errore e passa a uno stato pulito.
    - 2. Il Cassiere inizia una nuova vendita.
  - 1a. Il Cliente o il Direttore chiedono di riprendere una vendita sospesa.
    - 1. Il Cassiere esegue l'operazione di ripresa e inserisce il codice identificativo della vendita da riprendere.
    - 2. Il Sistema visualizza lo stato della vendita ripresa, con il totale parziale.
  - 2a. Vendita non trovata:
    - 1. Il Sistema segnala l'errore al Cassiere.
    - 2. Il Cassiere probabilmente inizia una nuova vendita e reinserisce tutti gli articoli.
    - 3. Il Cassiere continua con la vendita (probabilmente inserendo altri articoli o gestendo il pagamento).
  - 2-4a. Il Cliente dice al Cassiere di godere di un'esenzione dalle imposte (per esempio, perché è anziano)
    - 1. Il Cassiere verifica, quindi inserisce il codice per lo stato di esenzione dalle imposte.
    - 2. Il Sistema registra lo stato (che utilizzerà durante il calcolo delle imposte).
  - 3a. Codice identificativo dell'articolo non valido (non trovato nel sistema):
    - 1. Il Sistema segnala l'errore e rifiuta l'inserimento.
    - 2. Il Cassiere risponde all'errore:
      - 2a. C'è un codice identificativo dell'articolo leggibile (per esempio, un codice UPC numerico):
        - 1. Il Cassiere inserisce il codice dell'articolo manualmente.
        - 2. Il Sistema visualizza descrizione e prezzo.
        - 2a. Codice identificativo dell'articolo non valido: Il Sistema segnala l'errore. Il Cassiere prova in un altro modo.
      - 2b. Non c'è un codice identificativo dell'articolo, ma sul cartellino è presente un prezzo:
        - 1. Il Cassiere chiede al Direttore di eseguire un'operazione di sovrascrittura.
        - 2. Il Direttore esegue la sovrascrittura.
        - 3. Il Cassiere richiede l'inserimento manuale del prezzo, inserisce il prezzo e seleziona il tipo di tassazione per questo prodotto (poiché non vi sono informazioni sul prodotto, non si saprebbe altrimenti come calcolare le imposte)
        - 2c. Il Cassiere esegue Aiuto.Ricerca.Prodotto per ottenere il vero codice identificativo dell'articolo e il suo prezzo.
        - 2d. Altrimenti, Il Cassiere chiede a un dipendente il codice identificativo effettivo dell'articolo o il suo prezzo, e inserisce il codice o il prezzo manualmente (vedi sopra).
      - 3b. Ci sono più articoli della stessa categoria e non è importante tenere traccia dell'identità univoca dell'articolo (per esempio, 5 confezioni di hamburger vegetariani):
        - 1. Il Cassiere può inserire il codice identificativo della categoria dell'articolo e la quantità.

- 3c. L'articolo richiede l'inserimento manuale della categoria e del prezzo (come fiori o cartoline con un prezzo applicato):  
1. Il Cassiere inserisce manualmente il particolare codice della categoria, più il prezzo.
- 3-6a. Il Cliente chiede al Cassiere di eliminare (ovvero, di annullare) un articolo dall'acquisto (questo è consentito solo se il valore dell'articolo è inferiore al limite di annullamento per il Cassiere, altrimenti è necessaria un'operazione di sovrascrittura da parte del Direttore):  
1. Il Cassiere inserisce il codice identificativo dell'articolo da rimuovere dalla vendita.  
2. Il Sistema elimina l'articolo e visualizza il totale parziale aggiornato:  
2a. Il prezzo dell'articolo supera il limite di annullamento per il Cassiere:  
1. Il Sistema segnala l'errore, e suggerisce una sovrascrittura da parte del Direttore.  
2. Il Cassiere richiede la sovrascrittura da parte del Direttore, la ottiene e ripete l'operazione.
- 3-6b. Il Cliente dice al Cassiere di annullare la vendita:  
1. Il Cassiere annulla la vendita sul Sistema.
- 3-6c. Il Cassiere sospende la vendita:  
1. Il Sistema registra la vendita in modo tale che possa essere ripresa su qualsiasi registrator POS.  
2. Il Sistema presenta una "ricevuta di sospensione", che comprende un elenco degli articoli inseriti e un codice identificativo della vendita, da usare per recuperare e riprendere la vendita.
- 4a. Il prezzo dell'articolo fornito dal sistema non è accettato (per esempio, il Cliente si è lamentato di qualcosa e gli viene offerto un prezzo inferiore):  
1. Il Cassiere richiede l'approvazione dal Direttore.  
2. Il Direttore esegue un'operazione di sovrascrittura.  
3. Il Cassiere inserisce manualmente il prezzo corretto per l'articolo.  
4. Il Sistema mostra il nuovo prezzo.
- 5a. Il Sistema rileva un fallimento nella comunicazione con il servizio esterno di calcolo delle imposte:  
1. Il Sistema riavvia il servizio sul nodo POS, e prosegue:  
1a. Il Sistema rileva che il servizio non riprende.  
1. Il Sistema segnala l'errore.  
2. Il Cassiere può calcolare le imposte e inserirle manualmente, oppure annullare la vendita.
- 5b. Il Cliente dice di aver diritto a uno sconto (per esempio in quanto dipendente oppure cliente abituale):  
1. Il Cassiere segnala la richiesta di sconto.  
2. Il Cassiere inserisce il codice identificativo del Cliente.  
3. Il Sistema mostra il totale con lo sconto applicato, calcolato in base alle regole sugli sconti.
- 5c. Il Cliente dice di avere un credito sul proprio conto, da applicare alla vendita:  
1. Il Cassiere segnala la richiesta di credito.  
2. Il Cassiere inserisce il codice identificativo del Cliente.  
3. Il Sistema applica il credito al prezzo, e riduce il credito residuo nella stessa misura.
- 6a. Il Cliente dice che intende pagare in contanti, ma non ha abbastanza contanti:  
1. Il Cassiere chiede un metodo di pagamento alternativo.  
1a. Il Cliente dice al Cassiere di annullare la vendita. Il Cassiere annulla la vendita sul Sistema.

## 7a. Pagamento in contanti:

1. Il Cassiere inserisce l'importo in contanti presentato dal Cliente.
2. Il Sistema mostra il resto dovuto e apre il cassetto della cassa.
3. Il Cassiere deposita il contante presentato e restituisce il resto in contanti al Cliente.
4. Il Sistema registra il pagamento in contanti.

## 7b. Pagamento con carta di credito:

1. Il Cliente inserisce le informazioni sulla propria carta di credito.
2. Il Sistema visualizza il pagamento per la verifica.
3. Il Cassiere conferma.

## 3a. Il Cassiere annulla l'operazione di pagamento:

1. Il Sistema ritorna alla modalità "inserimento articolo".

4. Il Sistema invia una richiesta di autorizzazione al pagamento a un sistema esterno di Servizio di Autorizzazione di Pagamento, per richiedere l'approvazione del pagamento.

## 4a. Il Sistema rileva un problema nella comunicazione con il sistema esterno:

1. Il Sistema segnala l'errore al Cassiere.
2. Il Cassiere chiede al Cliente un metodo di pagamento alternativo.

5. Il Sistema riceve l'approvazione di pagamento, segnala l'approvazione al Cassiere e apre il cassetto (per inserire la ricevuta di pagamento firmata).

## 5a. Il Sistema riceve un rifiuto per il pagamento:

1. Il Sistema segnala il rifiuto al Cassiere.
2. Il Cassiere chiede al Cliente un metodo di pagamento alternativo.

## 5b. Il Sistema rileva un timeout nell'attesa della risposta:

1. Il Sistema segnala il timeout al Cassiere.
2. Il Cassiere può riprovare, oppure chiedere al Cliente un metodo di pagamento alternativo.

6. Il Sistema registra il pagamento con carta di credito, che comprende l'approvazione del pagamento.

7. Il Sistema presenta il meccanismo di inserimento della firma per il pagamento.
8. Il Cassiere chiede al Cliente la firma per la ricevuta del pagamento con carta di credito. Il Cliente inserisce la firma.

9. Se la firma sulla ricevuta è cartacea, il Cassiere mette la ricevuta nel cassetto e lo chiude.

## 7c. Pagamento con assegno ...

## 7d. Pagamento con carta bancomat ...

## 7e. Il Cassiere annulla l'operazione di pagamento:

1. Il Sistema ritorna alla modalità "inserimento articolo".

## 7f. Il Cliente presenta dei buoni (buoni sconto o buoni regalo):

1. Prima di gestire il pagamento, il Cassiere registra ciascun buono. Il Sistema riduce il prezzo di conseguenza. Il Sistema registra i buoni utilizzati, per motivi di contabilità.

## 1a. Il buono inserito non è valido per nessuno degli articoli acquistati:

1. Il Sistema segnala l'errore al Cassiere.

## 8a. Il Sistema rileva un fallimento nella comunicazione con il sistema esterno di Contabilità: ...

## 8b. Il Sistema rileva un fallimento nella comunicazione con il sistema esterno di Inventario: ...

## 9a. Il Cliente richiede una ricevuta regalo (senza indicazione del prezzo):

1. Il Cassiere richiede la ricevuta e il Sistema la presenta.

## 9b. La stampante ha esaurito la carta.

1. Se il Sistema riesce a rilevare l'errore, segnala il problema.
2. Il Cassiere sostituisce la carta.
3. Il Cassiere richiede un'altra ricevuta.

**Requisiti speciali:**

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

**Elenco delle varianti tecnologiche e dei dati:**

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

**Frequenza di ripetizione:** Potrebbe essere quasi ininterrotta.

**Problemi aperti:**

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

Questo caso d'uso è basato sui requisiti di un vero sistema POS, sviluppato con una progettazione OO e implementato in Java. Il caso d'uso è stato presentato in modo più illustrativo che esaustivo. Ciò nonostante presenta dettaglio e complessità sufficienti per dimostrare, realisticamente, che un caso d'uso dettagliato può comprendere molti dettagli dei requisiti. Questo esempio sarà usato come modello per discutere diversi aspetti relativi ai casi d'uso.

## 7.9 Sezioni di un caso d'uso dettagliato

### Elementi del preambolo

Il preambolo è composto da tutto ciò che precede lo scenario principale e le estensioni. Contiene informazioni che è importante leggere prima degli scenari del caso d'uso.

### Portata

La portata descrive i confini del sistema (o dei sistemi) in via di progettazione. Normalmente un caso d'uso descrive l'utilizzo di un sistema software (o di un sistema

## **Come scrivere un caso d'uso**

---

## Sezioni del caso d'uso: preambolo

È tutto ciò che precede lo scenario principale e le estensioni.

- **Portata:** descrive i confini del sistema in progettazione
- **Livello:** tipicamente livello di obiettivo utente o livello di 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:** elenco delle parti interessate, ossia chi ha interessi nel raggiungimento dell'obiettivo espresso dal caso d'uso in oggetto
- **Pre-condizioni:** che deve essere sempre vero prima di iniziare uno scenario del caso d'uso; **non** vengono verificate all'interno del caso d'uso
- **Garanzie di successo (*post-condizioni*):** che cosa deve essere vero quando è stato completato con successo il caso d'uso

## Sezioni del caso d'uso: scenario principale di successo

- Lo scenario principale di successo viene chiamato anche “**percorso felice**”, “*flusso di base*” o “*flusso tipico*”
- Descrive un percorso di successo comune che soddisfa gli interessi delle parti interessate
- Lo scenario principale è costituito da una sequenza di passi, che può contenere passi da ripetere più volte, ma che di solito non comprende alcuna condizione o diramazione (**perché?**).
- La gestione del comportamento condizionale e delle alternative viene solitamente descritta nella successiva sezione delle *Estensioni* (**perchè?**).

## Sezioni del caso d'uso: scenario principale di successo

I passi possono essere di tre tipi:

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

Nota: Il primo passo indica l'evento trigger che scatena l'esecuzione dello scenario (potrebbe non essere classificabile nelle categorie sopra esposte).

## Sezioni del caso d'uso: estensioni

---

- Le estensioni hanno lo scopo di descrivere tutti gli altri scenari oltre a quello principale, sia di successo che di fallimento
- Solitamente le estensioni sono descritte per differenza dallo scenario principale.
- Gli scenari relativi alle estensioni sono diramazioni dello scenario principale di successo, e vengono indicati con riferimento ai suoi passi

## Sezioni del caso d'uso: estensioni

Un'estensione è costituita da due parti:

- la **condizione**
- la **gestione**

Quando è possibile, la condizione va scritta come qualcosa che possa essere rilevato dal sistema o da un attore.

### Ad esempio

Il Sistema rileva un fallimento nella comunicazione con il servizio esterno di calcolo delle imposte.

Piuttosto che: Il servizio esterno di calcolo delle imposte non funziona (**NO!**).

La gestione può essere riassunta in un unico passo, oppure può comprendere una sequenza di passi al termine dei quali solitamente lo scenario si fonde di nuovo con lo scenario principale di successo.

## Sezioni del caso d'uso: tipi di estensioni

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

- L'attore vuole che l'esecuzione del caso d'uso proceda in modo diverso da quanto previsto nello scenario principale
- Il caso d'uso deve procedere diversamente da quanto previsto nello scenario principale, ed è il sistema che se ne accorge, mentre esegue un'azione o effettua una validazione
- Un passo dello scenario principale descrive un'azione “generica” o “astratta”, mentre le estensioni relative a questo passo descrivono le possibili azioni “specifiche” o “concrete” per eseguire il passo

# Caso d'uso: notazione

- Oltre al formato mostrato precedentemente (lucido: *Esempio caso d'uso dettagliato*) esistono altri template per i casi d'uso
- In particolare il formato “*a due colonne*” che enfatizza la conversazione tra gli attori e il sistema (*responsabilità del sistema*)
- **Questo formato è quello scelto per il laboratorio del corso**

## 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.
4. Il Sistema registra la riga di vendita per l'articolo e mostra la descrizione dell'articolo, il suo prezzo, il totale parziale. Il prezzo è calcolato in base a un insieme di regole di prezzo.
5. Il Sistema mostra il totale con le imposte calcolate.
6. Il Cassiere riferisce il totale al Cliente, e richiede il pagamento.
7. Il Cliente paga.
8. Il Sistema gestisce il pagamento.
9. Il Sistema registra la vendita completa e invia informazioni sulla vendita e sul pagamento ai sistemi esterni di Contabilità (per la contabilità e le commissioni) e di Inventario (per l'aggiornamento dell'inventario).
10. Il Sistema genera la ricevuta.
11. Il Cliente va via con la ricevuta e gli articoli acquistati.

Responsabilità del Sistema

# Caso d'uso: notazione

- Oltre al formato mostrato precedentemente (lucido:  
*Esempio caso d'uso dettagliato*) esistono altri template per i casi d'uso
- In particolare il formato “*a due colonne*” che enfatizza la conversazione tra gli attori e il sistema (*responsabilità del sistema*)
- **Questo formato è quello scelto per il laboratorio del corso**

## Scenario principale di successo

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

# Sezioni del caso d'uso: esempio numerazione delle estensioni

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

## Scenario principale di successo

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

# Sezioni del caso d'uso: esempio numerazione delle estensioni

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

## Estensione 1a: lavora su un menu esistente

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

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

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

## Estensione 3a: lavora su una sezione esistente

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

## Estensione 3b: elimina una sezione esistente

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

# Sezioni del caso d'uso: esempio numerazione delle estensioni

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

## Estensione 4a: elimina una voce dal menù

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

## Estensione 4b: modifica una voce del menù

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

## Estensione 4c: cambia il nome di una sezione

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

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

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

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

#	Attore	Sistema
	Va al passo 6 dello scenario principale	

Dal progetto di laboratorio 2017/2018.

# Sezioni del caso d'uso: esempio numerazione delle estensioni

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

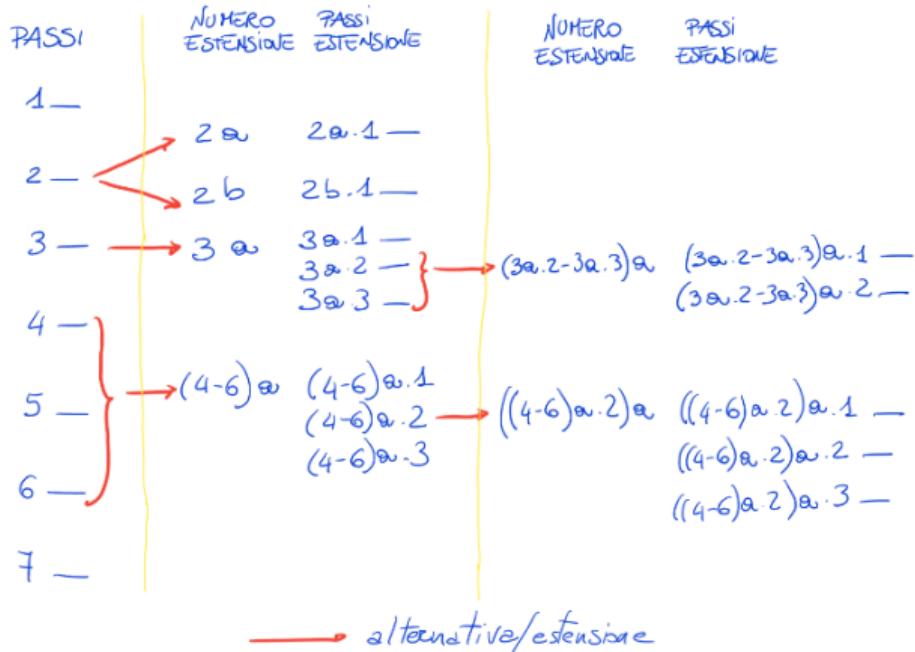
## Estensione (3-5)b: elimina questo menù

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

Dal progetto di laboratorio 2017/2018.

## Sezioni del caso d'uso: esempio numerazione delle estensioni

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



# Come scrivere i casi d'uso: stile essenziale

## Stile essenziale e conciso!

Scrivere i casi d'uso in uno stile essenziale; ignorare l'interfaccia utente, concentrarsi sull'obiettivo utente!

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

# Come scrivere i casi d'uso: stile essenziale

## Stile essenziale e conciso!

Scrivere i casi d'uso in uno stile essenziale; ignorare l'interfaccia utente, concentrarsi sull'obiettivo utente!

## Sì! Stile essenziale

- *L'Amministratore si identifica*
- *Il Sistema autentica l'identità*

## No! Stile concreto

- *L'Amministratore inserisce ID e password nella finestra di dialogo (si veda Figura 3)*
- *Il Sistema autentica l'Amministratore*
- *Il Sistema visualizza la finestra “edit users” (si veda Figura 4)*

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

# Come scrivere i casi d'uso: stile essenziale

## Stile essenziale e conciso!

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

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

# Come scrivere i casi d'uso: stile essenziale

## Stile essenziale e conciso!

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

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

## Practical reasoning (ragionamento su azioni)



- Il **practical reasoning** è il ragionamento sulle azioni, sul processo di capire cosa fare
- "*Practical reasoning is a matter of weighing conflicting considerations for and against competing options, where the relevant considerations are provided by what the agent desires/values/cares about and what the agent believes.*" (Bratman)
- Il practical reasoning si distingue da theoretical reasoning - il *theoretical reasoning* riguarda le credenze

10

Dal Corso Agenti Intelligenti 2020/2021.

# Come scrivere i casi d'uso: stile essenziale

## Stile essenziale e conciso!

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

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

## Practical reasoning (ragionamento su azioni)



- Il practical reasoning è il ragionamento sulle azioni, sul processo di capire cosa fare.
- Consiste (nelle attività umane) di due attività:
  - deliberazione (*deliberation*): decidere **quale** stato di cose vogliamo raggiungere
  - pianificazione (*means-ends reasoning – planning*): decidere **come** raggiungere questi stati di cose
- L'output della deliberazione sono le **intenzioni**.

11

Dal Corso Agenti Intelligenti 2020/2021.

# Come scrivere i casi d'uso: stile essenziale

## Stile essenziale e conciso!

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

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

## Practical reasoning (ragionamento su azioni)



### Esempio

Quando una persona si laurea all'università, si trova di fronte con la scelta di decidere che tipo di carriera seguire. Ad esempio, si potrebbe considerare una carriera come accademico o un carriera nell'industria.

- Il processo per decidere a quale carriera puntare è **deliberazione** (*deliberation*)
- Il passo successivo è decidere come raggiungere questo stato di cose (*state of affairs*). Questo processo è chiamato **means-ends reasoning**. Il fine del **means-ends reasoning** è un piano per raggiungere il prescelto stato di cose. (Ad esempio, un piano potrebbe coinvolgere prima fare domanda per un posto di dottorato)
- Dopo aver ottenuto un piano, in genere un agente tenterà di **eseguirlo**

12

Dal Corso Agenti Intelligenti 2020/2021.

# Come scrivere i casi d'uso: stile essenziale

## Stile essenziale e conciso!

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

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

### Intenzioni nel practical reasoning



Deliberation e means-ends sono processi computazionali. Come tali, in tutti gli agenti reali questi processi avranno luogo in presenza di risorse limitate.

Ciò ha due importanti implicazioni:

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

#### Intenzioni

Ci riferiamo allo stato di cose che un agente ha scelto e al quale si impegna come sue **intenzioni**.

14

Dal Corso Agenti Intelligenti 2020/2021.

# Come scrivere i casi d'uso: a scatola nera

## A scatola nera!

Il sistema è descritto come dotato di responsabilità.

Si descrivere **che cosa** deve fare (comportamento o requisiti funzionali) senza decidere **come** lo farà (progettazione), es. *Il Sistema registra la vendita*.

Durante l'analisi dei requisiti bisogna specificare il comportamento esterno del sistema, considerato a scatola nera, evitando di prendere decisioni sul “*come*”.

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

# Come scrivere i casi d'uso: a scatola nera

## A scatola nera!

Il sistema è descritto come dotato di responsabilità.

Si descrivere **che cosa** deve fare (comportamento o requisiti funzionali) senza decidere **come** lo farà (progettazione), es. *Il Sistema registra la vendita*.

Durante l'analisi dei requisiti bisogna specificare il comportamento esterno del sistema, considerato a scatola nera, evitando di prendere decisioni sul “*come*”.

## No!

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

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

# Come scrivere i casi d'uso: adottare il punto di vista dell'attore

## Ivar Jacobson:

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

- Scrivere i requisiti concentrandosi sugli utenti o attore di un sistema, chiedendo quali sono i loro obiettivi e le situazioni tipiche
- Concentrarsi sulla comprensione di ciò che l'attore considera un risultato di valore

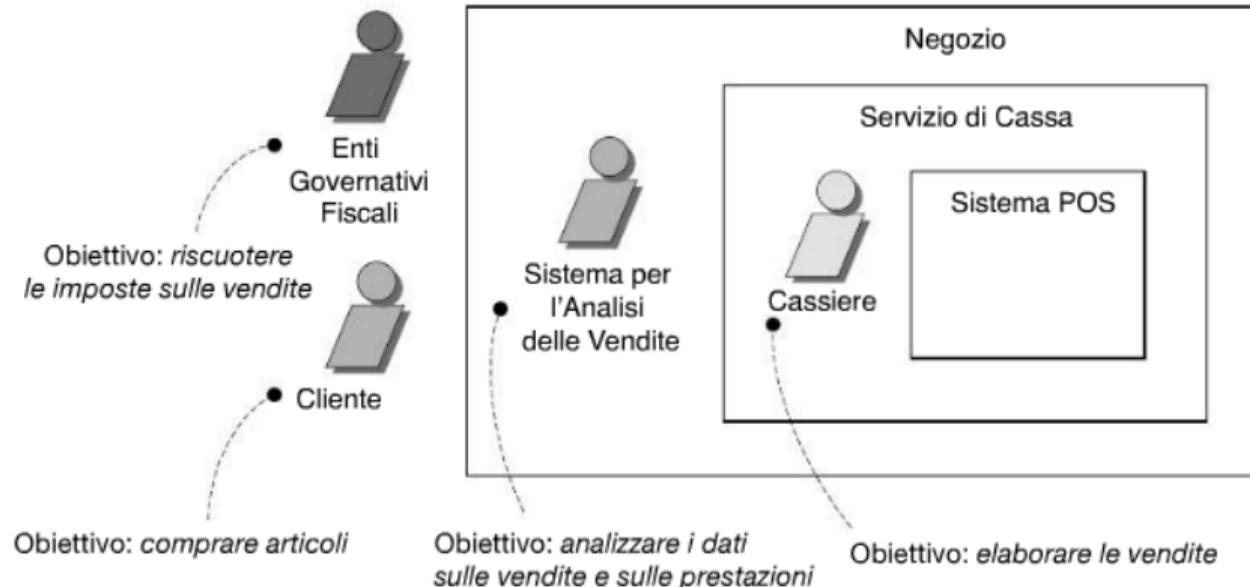
Nota: l'elenco delle caratteristiche di un sistema (come nel metodo “a cascata”) non incoraggia a porre domande su chi utilizza il prodotto e su cosa aggiunge valore.

## Trovare i casi d'uso

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

## Come scrivere i casi d'uso: come trovarli – attori primari (passi 2 e 3)

Identificare gli attori primari e i loro obiettivi: sono sempre esterni al sistema e aiutano a definire i confini dello stesso.



## Come scrivere i casi d'uso: come trovarli – attori primari (passi 2 e 3)

Identificare gli attori primari e i loro obiettivi: sono sempre esterni al sistema e aiutano a definire i confini dello stesso.

Attore	Obiettivo	Attore	Obiettivo
Cassiere	elaborare le vendite elaborare i noleggi gestire le restituzioni cash in cash out ...	Amministratore del Sistema	aggiungere utenti modificare utenti eliminare utenti gestire sicurezza gestire tabelle di sistema ...
Direttore	avviare il sistema arrestare il sistema ...	Sistema per l'Analisi delle Vendite...	analizzare dati sulle vendite ...

# Verificare l'utilità dei casi d'uso

In realtà la domanda più opportuna è: “Qual è un livello utile per esprimere i casi d'uso nell'analisi dei requisiti di una applicazione software?”

## Possibili test per rispondere

1. Il test del capo
2. Il test EBP (*Elementary Business Process* – processo di business elementare)
3. Il test della dimensione

## Verificare l'utilità dei casi d'uso: il test del capo

### Il test del capo

“Cosa avete fatto tutto i giorno?” “Il login!” – Chiedersi: “Il capo sarà felice?”

Se non lo è, il caso d'uso non supera il test del capo. Il caso d'uso non è utile.

# Verificare l'utilità dei casi d'uso: il test EBP

## EBP

Un processo di business elementare è un'attività svolta da una persona in un determinato tempo e luogo, in risposta a un evento di business, che **aggiunge un valore** di business misurabile e lascia i dati in uno stato consistente; per esempio, “Approva un credito” o “Stabilisci il prezzo per un codice”.

## Verificare l'utilità dei casi d'uso: il test della dimensione

### Valutare la dimensione

Un caso d'uso è raramente costituito da una singola azione o passo; normalmente comprende diversi passi, e nel suo formato dettagliato richiede da 3 a 10 pagine di testo.

# Verificare l'utilità dei casi d'uso: esempio

## Esempi

1. Negoziare un contratto con un fornitore
2. Gestire una restituzione
3. Effettuare il login
4. Spostare una pedina sul tabellone da gioco

(1) è molto più ampio e lungo di un EBP, potrebbe essere modellato come un caso d'uso di business invece che un caso d'uso di sistema.

(2) d'accordo con il capo, è simile a un EBP, le dimensioni vanno bene.

(3) il capo non è contento se ci si limita a fare questo tutto il giorno!

(4) passo singolo, non supera il test della dimensione.

# Livello dei casi d'uso

## Livello di obiettivo utente

Nell'analisi dei requisiti per un sistema software è utile concentrarsi soprattutto sui casi d'uso EBP: un caso d'uso di questo tipo è a livello di obiettivo utente, poiché consente all'utente di raggiungere un proprio obiettivo di valore mediante un singolo utilizzo del sistema.

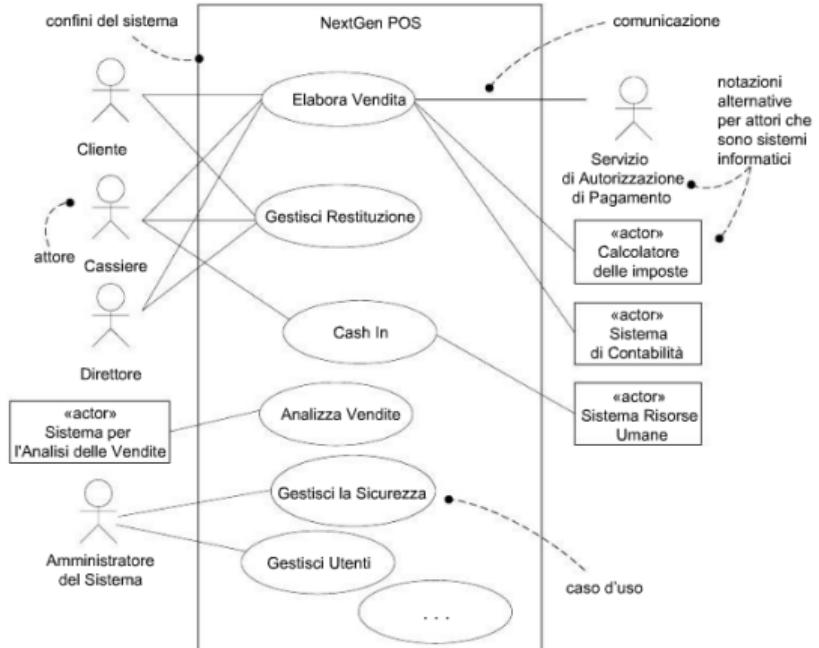
## Livello di sotto-funzione

Rappresenta una funzionalità nell'uso del sistema.

Utili per mettere a fattor comune delle sequenze di passi condivise da più casi d'uso, per evitare la duplicazione del testo in comune.

# Diagrammi dei casi d'uso (UCD)

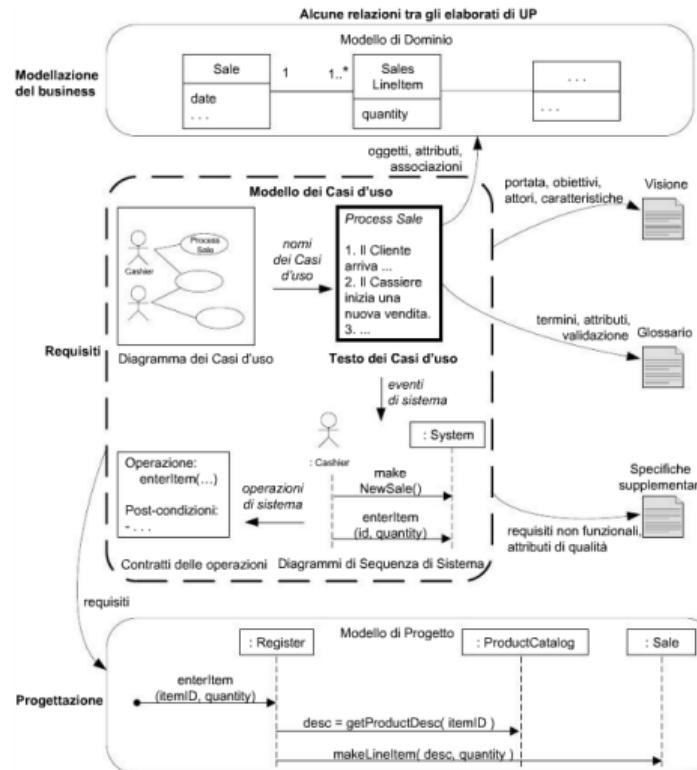
Disegnare un semplice diagramma dei casi d'uso insieme a un elenco attori-obiettivi.



# Lavorare con gli UC in UP

	<i>Identificazione UC</i>	<i>Descrizione dettagliata UC</i>	<i>Realizzazione UC</i>
<i>Ideazione</i>	50%–70%	10%	5%
<i>Elaborazione</i>	Quasi 100%	40%–80%	Meno del 10%
<i>Costruzione</i>	100%	100%	100%
<i>Transizione</i>			

# Relazione tra artefatti



# 04 . Dall'Ideazione all'Elaborazione

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

### **Si noti che**

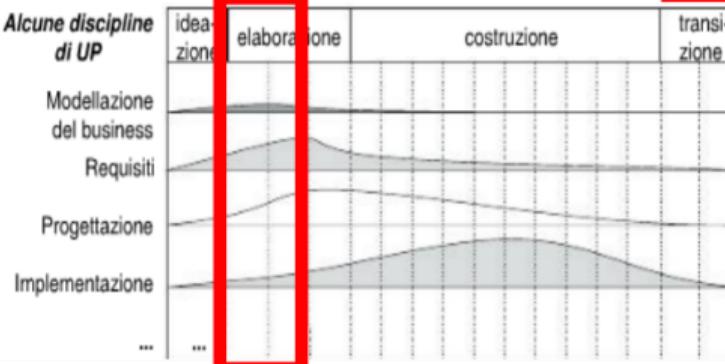
questi lucidi sono basati sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016” e sul materiale fornito dai docenti Viviana Bono, Claudia Picardi e Gianluca Torta dell’Università degli Studi di Torino che hanno tenuto il corso negli anni accademici precedenti.

# UP maps

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

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

## Alcune discipline di UP



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

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

# Verso l'elaborazione

Durante l'**ideazione**:

- Un breve workshop dei requisiti
- Assegnazione dei nomi per la maggior parte degli attori, degli obiettivi e dei casi d'uso
- Formato breve della maggior parte dei casi d'uso, il 10% anche in formato dettagliato
- Bozza di Visione e Specifiche Supplementari
- Lista dei rishi
- Proof-of-concept e prototipi (per esaminare la fattibilità tecnica dei requisiti speciali o orientati all'interfaccia utente per chiarire la visione dei requisiti funzionali)
- Raccomandazioni su quali componenti acquistare/costruire/riusare
- Proposta dell'architettura ad alto livello e dei componenti candidati
- Piano della prima iterazione
- Elenco di strumenti candidati

# Verso l'elaborazione

L'elaborazione è la serie iniziale di iterazioni durante le quali:

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

## Elaborazione

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

Nota: durante questa fase non vengono creati prototipi “usa e getta” ma codice e progettazione sono parti di qualità-produzione del sistema finale.

## Pianificazione dell'iterazione **e** successiva

I requisiti e le iterazioni sono organizzate in base al **rischio** (tecnico, incertezza dello sforzo, usabilità), **copertura** (le iterazioni iniziali devono coprire tutte le parti principali del sistema, implementazione in ampiezza e poco profonda) e **criticità** (le funzioni che il cliente considera di elevato valore di business).

La *classifica* viene stilata prima dell'iterazione 1, poi nuovamente prima dell'iterazione 2 e così via (per facilitare l'adattatività)

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 sottodomainio di sicurezza. ...
Basso	...	...

# Iterazione 1

## Iterazione 1

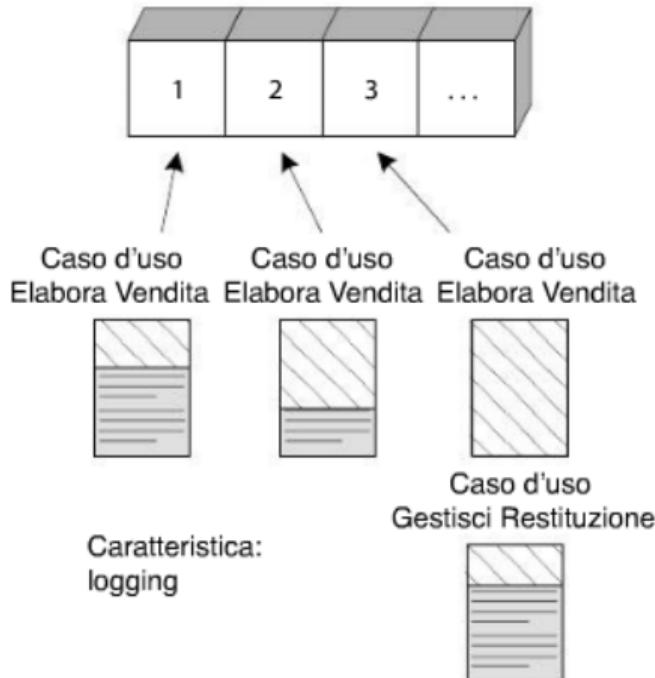
Nello sviluppo iterativo non si implementano tutti i requisiti in una sola volta: nell'iterazione 1 questi requisiti sono dei **sottoinsiemi** dei requisiti o dei casi d'uso completi.

## Iterazione 1

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

# Iterazione 1: sviluppo incrementale!

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



# Artefatti dell'elaborazione

Alcuni artefatti dell'elaborazione, esclusi quelli iniziati durante l'ideazione.

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.

## Non si è capita l'elaborazione se...

- Ha una durata superiore ad “alcuni” mesi per la maggior parte dei progetti
- Ha una sola iterazione
- La maggior parte dei requisiti è stata definita prima dell’elaborazione
- Gli elementi rischiosi e l’architettura non vengono affrontati
- Non si produce un’architettura eseguibile con progettazione e codice di qualità-produzione
- È considerata una fase di requisiti o di progettazione che precede una fase di implementazione nella costruzione
- Il feedback e l’adattamento sono minimi, gli utenti non vengono coinvolti continuamente
- Non vengono fatti test realistici fin dall’inizio
- L’architettura viene finalizzata in modo speculativo prima della programmazione
- È considerata un passo per fare programmazione proof-of-concept

# **05 . Modello di Dominio**

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

### **Si noti che**

questi lucidi sono basati sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016” e sul materiale fornito dai docenti Viviana Bono, Claudia Picardi e Gianluca Torta dell’Università degli Studi di Torino che hanno tenuto il corso negli anni accademici precedenti.

# Table of contents

1. Disciplina Modellazione del Business: Modello di Dominio
2. Creare un modello di dominio
3. Classi concettuali
4. Associazioni
5. Attributi
6. Ultime verifiche al modello

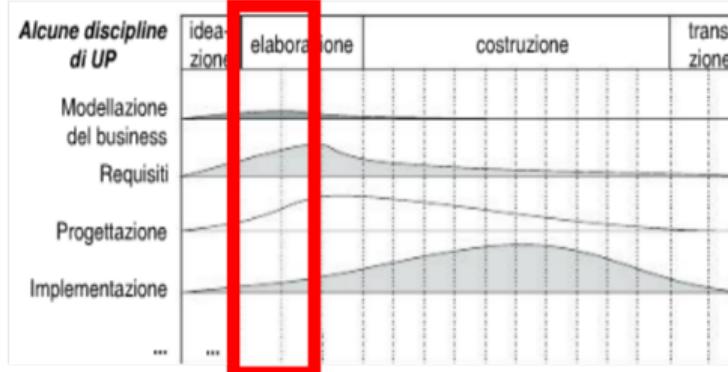
# **Disciplina Modellazione del Business: Modello di Dominio**

---

# UP maps

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

Disciplina	Pratica	Elaborato	Ideazione	Elaboraz.	Costr.	Transiz.
Modellazione del business	modellazione agile workshop requisiti	Modello di Dominio		i		
Progettazione	esercizio sulla visione votazione a punti	Casi d'Uso				
		Visione	i	r		
		Specifiche	i	r		
		Supplementare				
Implementazione	Glossario	Glossario	i	r		
		Modello di Progetto		i	r	
		Documento dell'Architettura		i		
		Software				
Gestione del progetto	Modello dei Dati	Modello dei Dati		i	r	
		...				
		...				
		...				

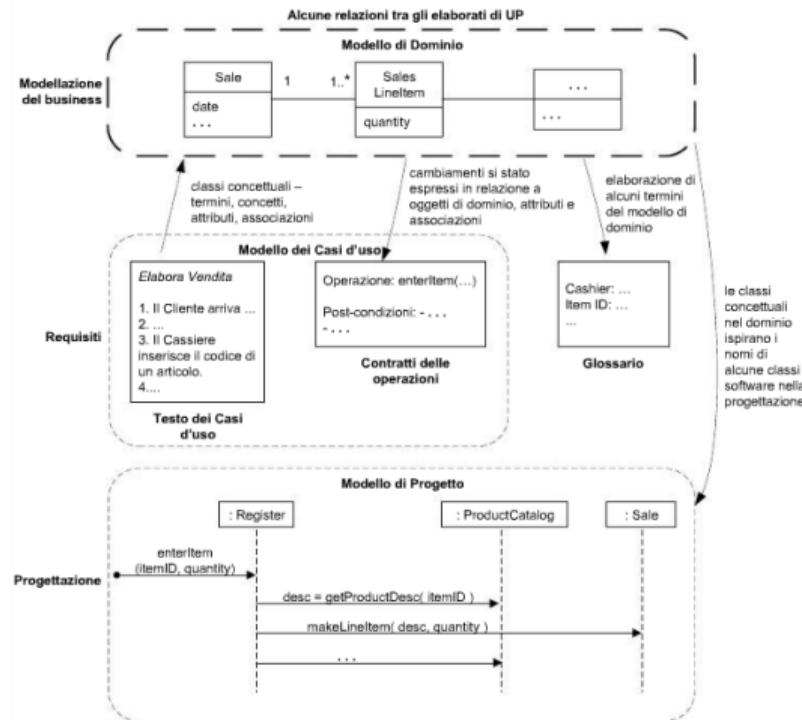


L'impegno relativo nelle discipline cambia a seconda delle fasi.

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

# Relazioni tra elaborati di UP

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

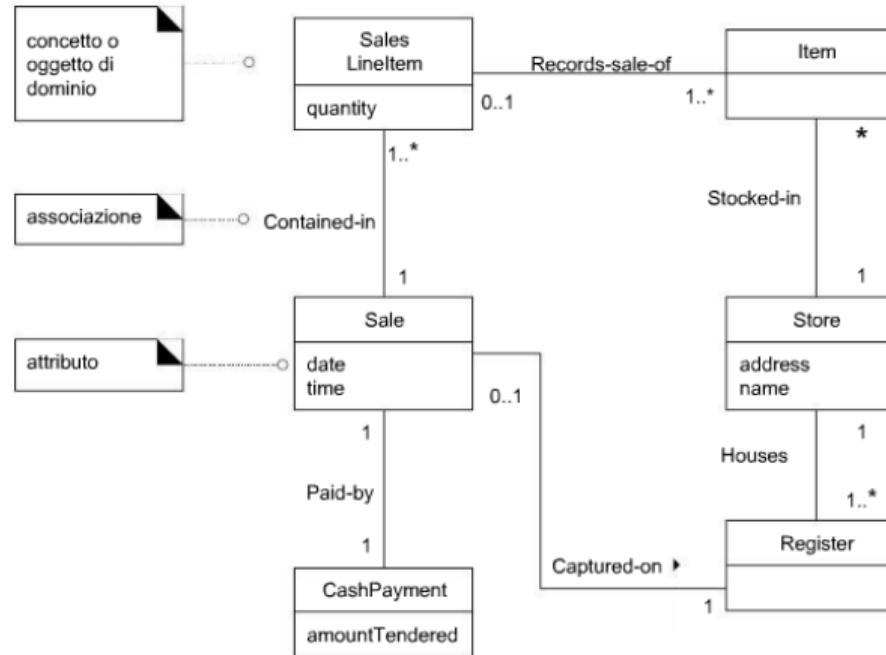


# Modello di Dominio

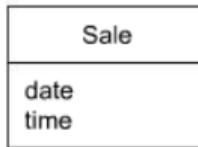
- Rappresentazione **visuale** delle **classi concettuali**, oggetti reali del dominio (non sono oggetti software!)
- Sviluppato nell'ambito della disciplina di *Modellazione del Business*
  - Specializzazione del “Business Object Model”
- Insieme di **diagrammi di classi UML**, includono:
  - **Oggetti** di dominio-classi concettuali
  - **Associazioni** tra classi concettuali
  - **Attributi** di classi concettuali
  - Non appaiono operazioni (*firma di metodi o responsabilità*)
- È un **dizionario visuale**
- Non è un modello dei dati
- Un modello di dominio è un modo particolare di utilizzare un diagramma delle classi di UML, con uno scopo specifico

# Esempio di Modello di Dominio di POS NextGEN

Un modello secondo un punto di vista concettuale. L'identificazione di un ricco insieme di classi concettuali è al centro dell'analisi OO.



# Classi concettuali **non** classi software!



visualizzazione di un concetto del  
mondo reale nel dominio di interesse  
*non* è la raffigurazione di una  
classe software

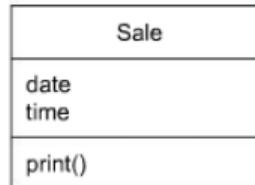
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

evitare



elemento software;  
non fa parte del modello di  
dominio

evitare



classe software;  
non fa parte del modello di  
dominio

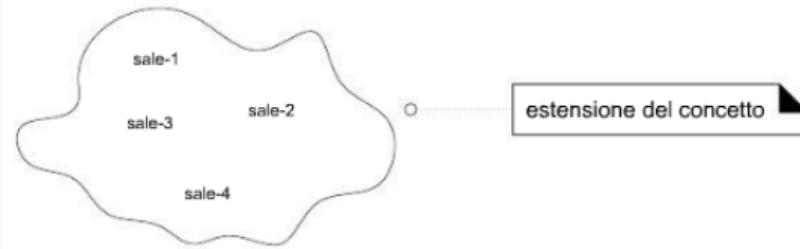
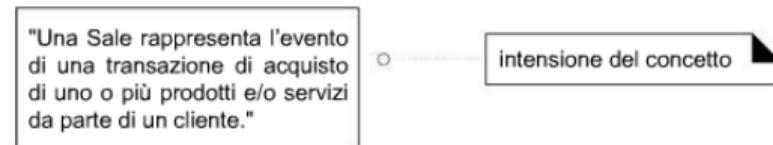
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Classi concettuali

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

## Una classe concettuale

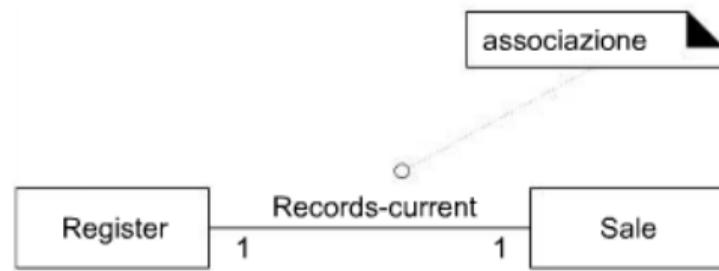
rappresenta un **concetto** del mondo reale o del dominio di interesse di un sistema che si sta modellando (modellazione *ontologica* VS modellazione *concettuale* o di *domino*).



## Una associazione

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

Se si pensa alle classi in maniera estensionale, un'associazione tra due classi è un insieme di coppie di oggetti dalle due classi.



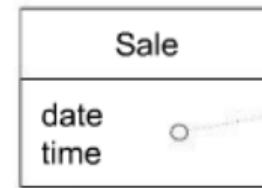
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

## Un attributo

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

Ciascun oggetto della classe ha un proprio valore, separato, per quella proprietà.

Una funzione che associa un valore a ciascun oggetto della classe.



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

# Riduzione del “gap di rappresentazione”

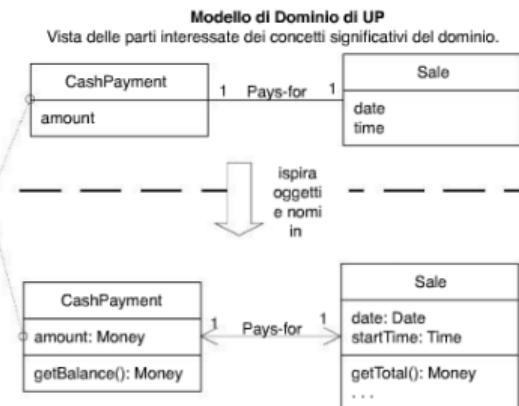
Comprendere i concetti chiave e la terminologia relativa al dominio del discorso del problema, più specificatamente:

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

CashPayment nel Modello di Dominio è un concetto, ma CashPayment nel Modello di Progetto è una classe software. Non sono la stessa cosa, ma il primo ha *ispirato* il nome e la definizione del secondo.

Questa scelta riduce il salto rappresentazionale.

Questa è una delle grandi idee della tecnologia a oggetti.



## **Creare un modello di dominio**

---

## Creare un modello di dominio

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

- Trovare le classi concettuali
- Disegnarle come classi in un diagramma delle classi UML
- Aggiungere le associazioni
- Aggiungere gli attributi

## **Classi concettuali**

---

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

- Riuso-modifica di modelli esistenti (pattern di analisi specifici in un determinato dominio di applicazione)
- Utilizzo di elenchi di categorie (preparando un elenco di classi candidate)
- Analisi linguistica (delle descrizioni testuali di un dominio):
  - Il mapping nome-classe non è automatico
  - Il linguaggio naturale è ambiguo
  - Fonti: casi d'uso descritti in formato dettagliato

## Esempio: elenco di categorie

Dal dominio POS, utilizziamo un elenco di categorie che enfatizzano le transazioni commerciali e le loro relazioni con le altre cose.

Categoria di classe concettuale	Esempi
<b>transazioni commerciali</b> <i>Linea guida:</i> sono aspetti critici (riguardano denaro), dunque si inizi con le transazioni.	<i>Sale, Payment (o CashPayment) Reservation</i>
<b>elementi/righe di transazioni</b> <i>Linea guida:</i> le transazioni spesso sono composte da righe per gli articoli correlati, quindi queste vanno considerate subito dopo le transazioni.	<i>SalesLineItem</i>
<b>prodotto o servizio correlato a una transazione o a una riga di transazione per articolo</b> <i>Linea guida:</i> le transazioni sono <i>per</i> qualcosa (un prodotto o un servizio). Vanno considerate subito dopo.	<i>Item Flight, Seat, Meal</i>
<b>dove viene registrata la transazione?</b> <i>Linea guida:</i> importante.	<i>Register, Ledger (o SalesLedger), FlightManifest</i>
<b>ruoli di persone o organizzazioni correlati alle transazioni; attori nei casi d'uso</b> <i>Linea guida:</i> normalmente dobbiamo sapere quali sono le parti coinvolte in una transazione.	<i>Cashier, Customer, Store MonopolyPlayer Passenger, Airline</i>

## Esempio: elenco di categorie

Dal dominio POS, utilizziamo un elenco di categorie che enfatizzano le transazioni commerciali e le loro relazioni con le altre cose.

<b>luogo della transazione; luogo del servizio</b>	<i>Store Airport, Plane, Seat</i>
<b>eventi significativi, spesso con un'ora o un luogo che è necessario ricordare</b>	<i>Sale, Payment (o CashPayment) MonopolyGame Flight</i>
<b>oggetti fisici</b> <i>Linea guida:</i> questo è particolarmente importante quando si crea software per il controllo di dispositivi, oppure simulazioni.	<i>Item, Register Board, Piece, Die Airplane</i>
<b>descrizioni di oggetti</b> <i>Linea guida:</i> vedere il Paragrafo 12.13 per una discussione.	<i>ProductDescription FlightDescription</i>
<b>cataloghi</b> <i>Linea guida:</i> le descrizioni sono spesso contenute in un catalogo.	<i>ProductCatalog FlightCatalog</i>
<b>contenitori di oggetti (fisici o informazioni)</b>	<i>Store, Bin Board Airplane</i>

## Esempio: elenco di categorie

Dal dominio POS, utilizziamo un elenco di categorie che enfatizzano le transazioni commerciali e le loro relazioni con le altre cose.

Categorie	
<b>oggetti in un contenitore</b>	<i>Item</i> <i>Square (in un Board)</i> <i>Passenger</i>
<b>altri sistemi che collaborano</b>	<i>CreditAuthorizationSystem</i> <i>AirTrafficControl</i>
<b>registrazioni di questioni finanziarie, di lavoro, contrattuali e legali</b>	<i>Receipt</i> , <i>Ledger</i> <i>MaintenanceLog</i>
<b>strumenti finanziari</b>	<i>Cash</i> , <i>Check</i> , <i>LineOfCredit</i> <i>TicketCredit</i>
<b>piani, manuali, documenti cui si fa regolarmente riferimento per eseguire il lavoro</b>	<i>DailyPriceChangeList</i> <i>RepairSchedule</i>

## Esempio: analisi linguistica

L'analisi linguistica è un'altra fonte di ispirazione e i casi d'uso formato dettagliato sono un'ottima descrizione a cui ispirarsi per questa analisi.

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

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

Il Cassiere ripete i passi 2-3 fino a che non indica che ha terminato.

## Esempio: analisi linguistica

L'analisi linguistica è un'altra fonte di ispirazione e i casi d'uso formato dettagliato sono un'ottima descrizione a cui ispirarsi per questa analisi.

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

## Esempio: analisi linguistica

L'analisi linguistica è un'altra fonte di ispirazione e i casi d'uso formato dettagliato sono un'ottima descrizione a cui ispirarsi per questa analisi.

Estensioni (o Flussi alternativi):

...

7a. Pagamento in contanti:

1. Il Cassiere inserisce l'**importo in contanti** presentato dal Cliente.
2. Il Sistema mostra il **resto dovuto** e apre il cassetto della **cassa**.
3. Il Cassiere deposita il contante presentato e restituisce il resto in contanti al Cliente.
4. Il Sistema registra il **pagamento in contanti**.

## Esempio: definizione delle classi

Un modello di dominio conterrà una collezione, in un certo senso arbitraria, di astrazioni e termini del dominio che i modellatori considerano significative.



## Esempio: definizione delle classi

Si consideri il concetto “*Ricevuta*”:

- Se le informazioni che contiene si possono derivare da altre fonti (altre classi concettuali) si può escludere dal modello di dominio
- Se ha un ruolo in termini di regole di business, business ad esempio conferisce il diritto al possessore di restituire oggetti acquistati acquistati, allora è bene includere il concetto nel modello di dominio

Nota: nell’iterazione corrente di POS la restituzione articoli non è considerata, quindi *Ricevuta* viene esclusa dal modello di dominio in questa iterazione

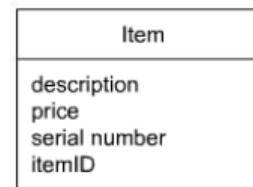
# Classi “Descrizione”

## Classe Descrizione

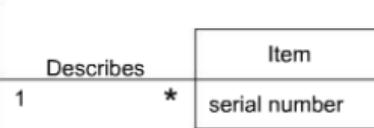
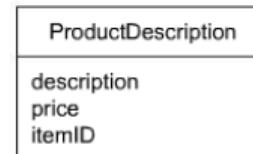
Una classe descrizione contiene informazioni che descrivono qualcos'altro.

È utile avere un'associazione che collega la classe e la sua classe descrizione (*pattern Item-Descriptor*)

- È necessaria una descrizione di un articolo o servizio indipendentemente dall'attuale esistenza di istanze dell'articolo o servizio
- L'eliminazione delle istanze di un articolo (esempio, Item) darebbe luogo ad una perdita di informazioni che è necessario conservare
- Si vogliono ridurre le informazioni ridondanti



Sconsigliato



Migliore

## **Associazioni**

---

# Associazioni

## Associazioni

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

### È utile includere:

- Associazioni la cui conoscenza della relazione deve essere conservata per una qualche durata (associazioni “*da ricordare*”)
- Associazioni derivate dall'elenco di associazioni comuni (si veda più avanti)

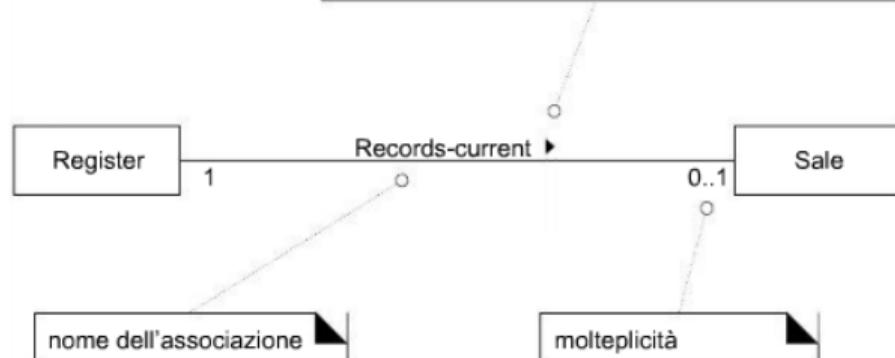
Nota: è utile introdurre solo le associazioni davvero rilevanti al dominio modellato.

## Caratterizzare una associazione con

- Nome significativo seguendo la traccia *NomeClasse-FraseVerbale-NomeClasse*
- Molteplicità e direzione di lettura

Nota: un'associazione è per natura **bidirezionale** (non è un'affermazione su collegamenti tra entità software). La direzione di lettura non è una specifica di visibilità.

► 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.



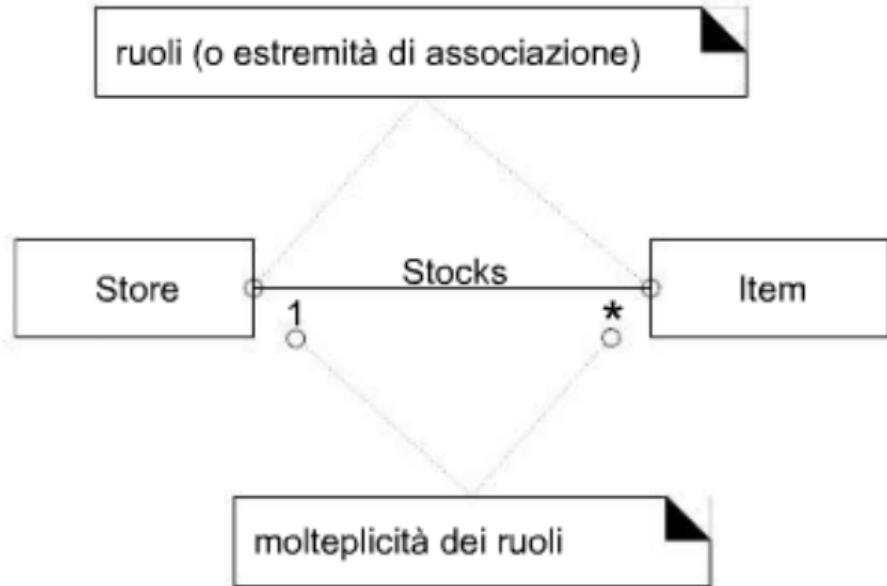
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Ruoli nelle associazioni

## Ruolo

Ciascuna estremità di una associazione è anche chiamata **ruolo**. I ruoli possono avere, optionalmente:

- espressione di **molteplicità**
- **nome**
- **navigabilità**



# Molteplicità delle associazioni

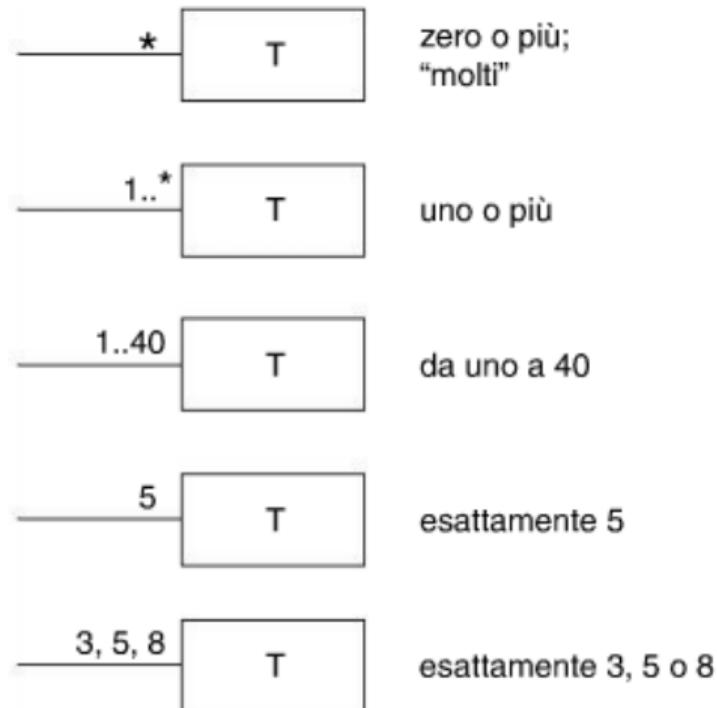
## Molteplicità

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

A..B vuol dire che A è la molteplicità minima (spesso 0 o 1) e B è la molteplicità massima (spesso 1 o \*).

I casi possibili in base alla molteplicità massima sono:

- *uno-a-molti*
- *molti-a-uno*
- *molti-a-molti*
- *uno-a-uno*



# Molteplicità delle associazioni

## Molteplicità

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



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

## Attenzione!

È possibile che uno specifico “*item*” possa essere in più “*store*” nel tempo  
**MA** in qualsiasi particolare momento l’*item* è in **un solo** *store*.

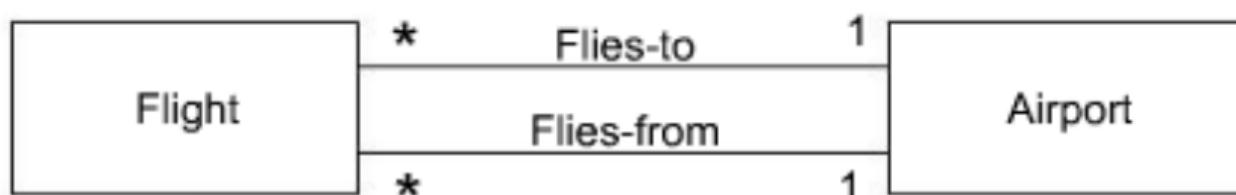
## Attenzione!

Il valore di una molteplicità dipende dall’interesse del modellatore e dello sviluppatore del software, perché comunica un vincolo di dominio che si rifletterà o potrebbe riflettersi nel software.

# Associazioni multiple

## Associazioni multiple tra due classi

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

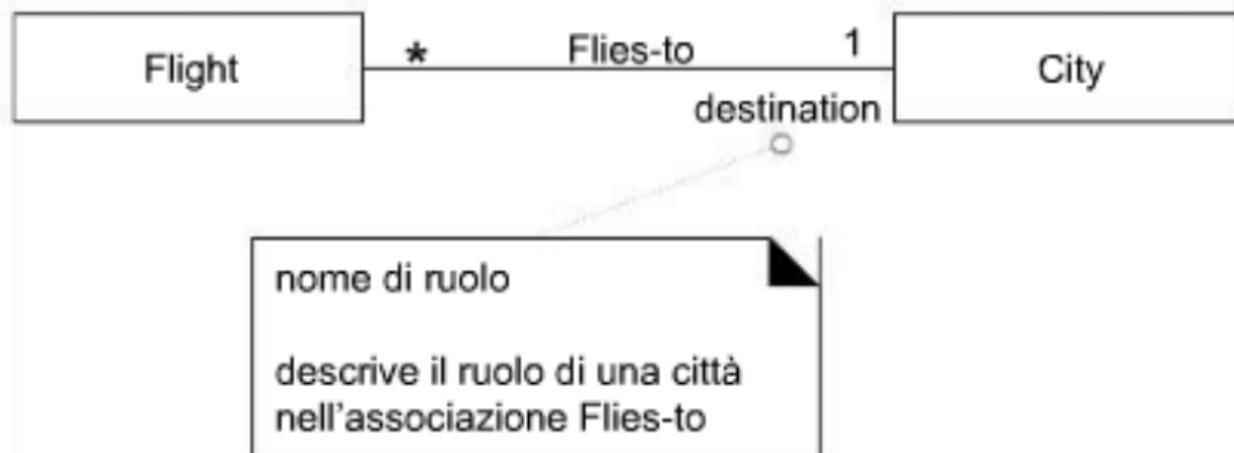


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Nomi di ruolo

## Nomi di ruolo

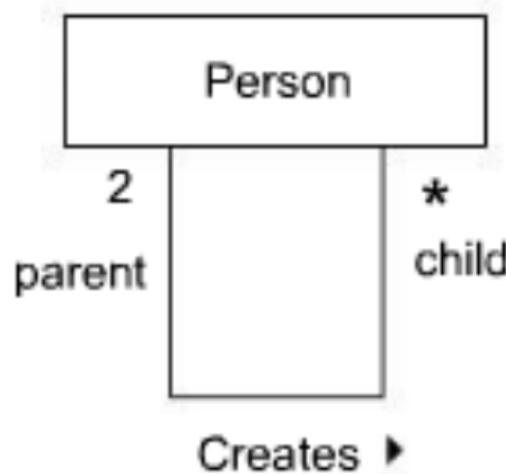
È possibile indicare il nome di un ruolo.



# Associazioni riflessive

## Associazioni riflessive

Una classe può anche avere un'associazione con se stessa. I nomi di ruolo sono in questo caso utili per indicare la molteplicità.

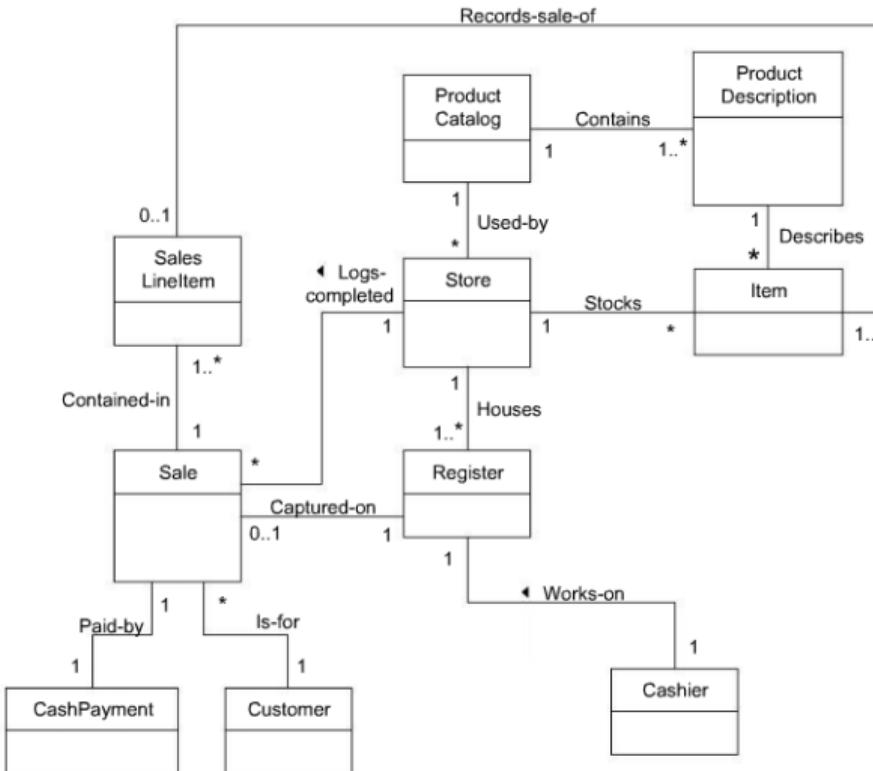


# Esempio: modello di dominio (parziale) per POS NextGen

Associazioni derivate dall'elenco di associazioni comuni per un sistema informativo.

Categoria	Esempi
A è una transazione correlata a un'altra transazione B	CashPayment—Sale Cancellation—Reservation
A è un elemento/riga di una transazione B	SalesLineItem—Sale
A è un prodotto o servizio per una transazione (o riga per l'articolo) B	Item—SalesLineItem (o Sale) Flight—Reservation
A è un ruolo relativo a una transazione B	Customer—Payment Passenger—Ticket
A è una parte fisica o logica di B	Drawer—Register Square—Board Seat—Airplane
A è contenuto fisicamente o logicamente in B	Register—Store, Item—Shelf Square—Board Passenger—Airplane
A è una descrizione per B	ProductDescription—Item FlightDescription—Flight
A è noto/registrato/memorizzato/riportato/acquisito in B	Sale—Register Piece—Square Reservation—FlightManifest
A è un membro di B	Cashier—Store Player—MonopolyGame Pilot—Airline
A è una sottounità organizzativa di B	Department—Store Maintenance—Airline
A utilizza o gestisce o possiede B	Cashier—Register Player—Piece Pilot—Airplane
A è vicino/prossimo a B	SalesLineItem—SalesLineItem Square—Square City—City

# Esempio: modello di dominio (parziale) per POS NextGen



# Composizione

L'aggregazione è, in UML, un tipo di associazione che suggerisce una relazione intero-parte.  
Un esempio di aggregazione è la relazione fra un'automobile e le sue ruote.

## Composizione

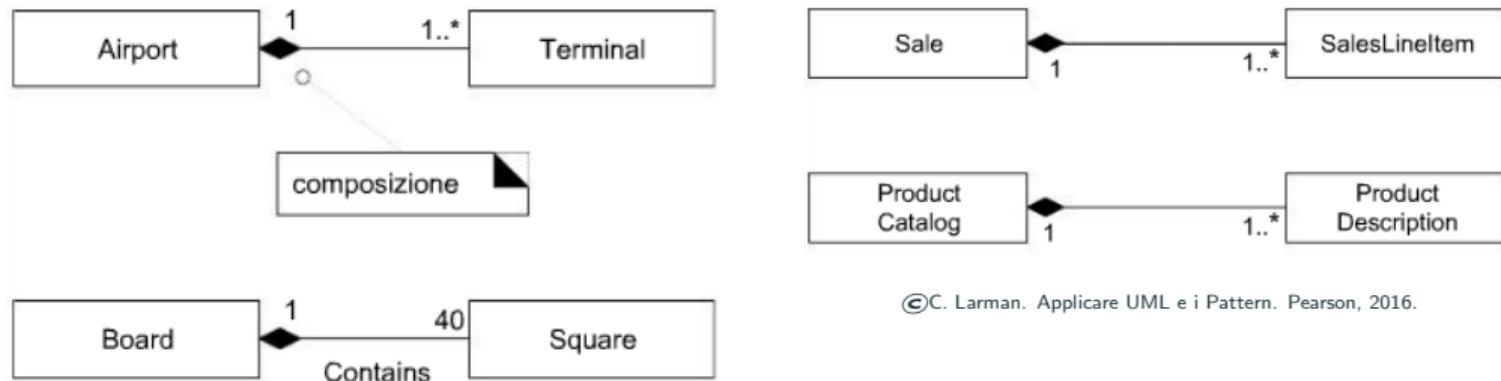
La **composizione**, o *aggregazione composta*, è un tipo di **forte di aggregazione intero-parte**:

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

L'aggregazione fra automobile e le sue ruote non è una composizione.

# Notazione della composizione

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



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

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

## **Attributi**

---

# Attributi

## Attributi

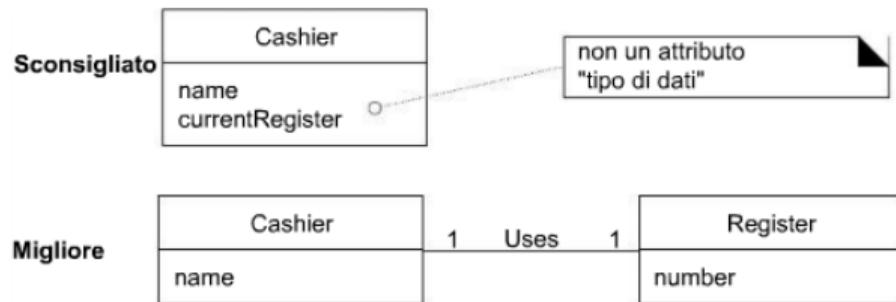
Un **attributo** di una classe rappresenta un *valore* (un dato) degli oggetti di quella classe

Aggiungere attributi che:

- Sono tipi di dati primitivi (boolean, date, number, character, string, ecc.)
- Sono tipi enumerativi (es. tipo servizio: gold, silver)

Caratterizzare un attributo con:

- Origine: derivato/non derivato
- Tipo di dato, ovvero un vincolo sui valori del dominio



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

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

# Attributi

## Attributi

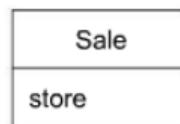
Un **attributo** di una classe rappresenta un *valore* (un dato) degli oggetti di quella classe

Aggiungere attributi che:

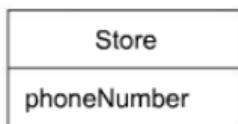
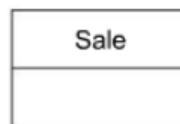
- Sono tipi di dati primitivi (boolean, date, number, character, string, ecc.)
- Sono tipi enumerativi (es. tipo servizio: gold, silver)

Caratterizzare un attributo con:

- Origine: derivato/non derivato
- Tipo di dato, ovvero un vincolo sui valori del dominio



oppure... ?



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

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

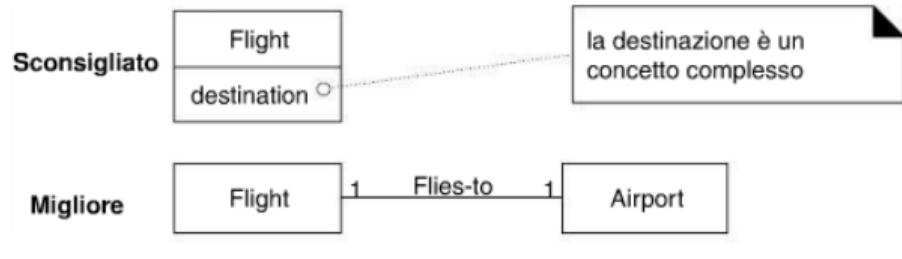
# Attributi

## Attributi

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

Aggiungere attributi che:

- Sono tipi di dati primitivi (boolean, date, number, character, string, ecc.)
- Sono tipi enumerativi (es. tipo servizio: gold, silver)



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

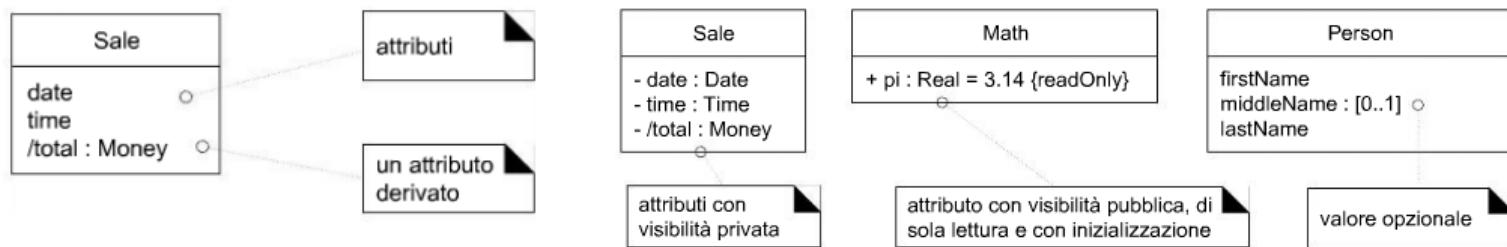
Classi concettuali vanno correlate con un'associazione,  
non con un attributo.

Caratterizzare un attributo con:

- Origine: derivato/non derivato
- Tipo di dato, ovvero un vincolo sui valori del dominio

# Notazione per gli attributi

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

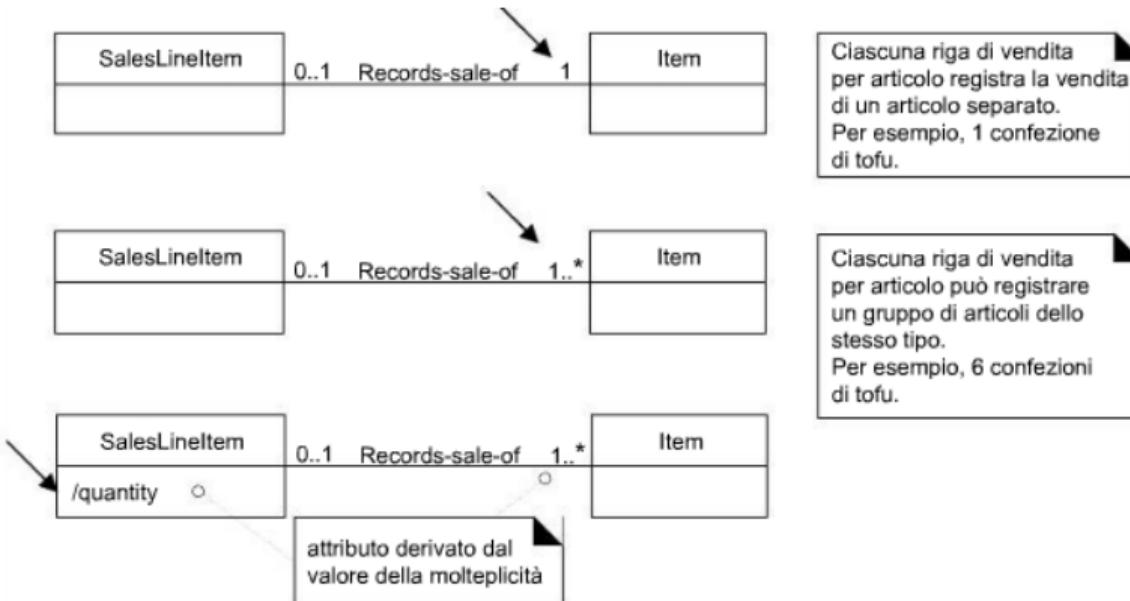


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

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

# Attributi derivati

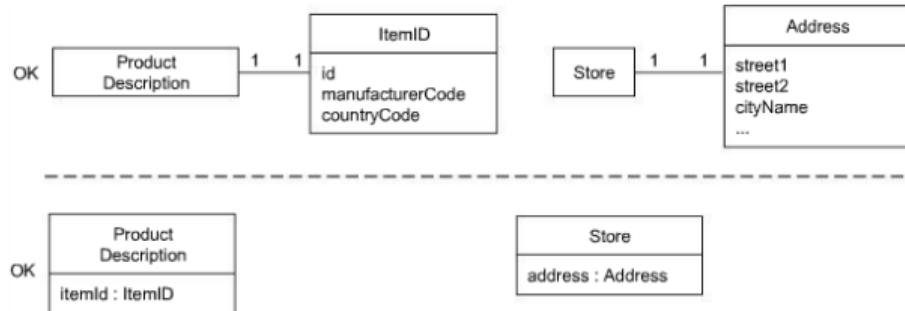
Un attributo può essere calcolato o derivato dalle informazioni contenute negli oggetti associati.  
Un attributo derivato può essere indicato dal simbolo “/” prima del nome dell'attributo.



# Classi tipo di dato

Quando introdurle?

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



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

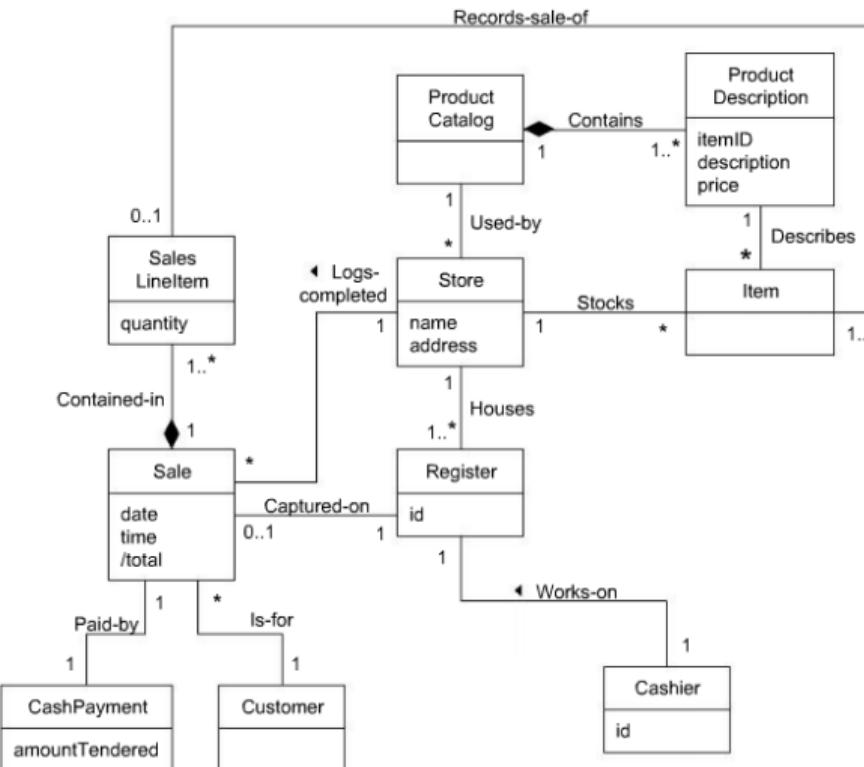
## **Ultime verifiche al modello**

---

# Ultime verifiche

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

# Esempio: modello di dominio (parziale) per POS NextGen



## Modello di dominio in UP

Il modello di dominio in UP viene creato in primo luogo durante le iterazioni dell'elaborazione, quando vi è la massima necessità di capire i concetti significativi e di rappresentarne alcuni come classi software durante il lavoro di progettazione.

Disciplina	Elaborato Iterazione →	Ideaz. I1	Elab. E1..En	Costr. C1..Cn	Trans. T1..T2
Modellazione del business	<b>Modello di Dominio</b>		i		
Requisiti	Modello dei Casi d'Uso (SSD) Visione Specifiche Supplementari Glossario	i i i i	r r r r		
Progettazione	Modello di progetto Documento dell'Architettura Software Modello dei Dati		i i i	r r	

# Generalizzazione

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

...  
**Estensioni:**

7b. Pagamento con carta di credito:

1. Il Cliente inserisce le informazioni sulla propria **carta di credito**.
2. Il Sistema invia una **richiesta di autorizzazione al pagamento** a un sistema esterno di **Servizio di Autorizzazione al Pagamento** per richiedere l'**approvazione del pagamento**.

2a. Il Sistema rileva un problema nella comunicazione con il sistema esterno:

1. Il Sistema segnala l'errore al Cassiere.
2. Il Cassiere chiede al Cliente un metodo di pagamento alternativo.

3. Il Sistema riceve l'**approvazione del pagamento** e segnala l'approvazione al Cassiere.

3a. Il Sistema riceve un **rifiuto per il pagamento**:

1. Il Sistema segnala il rifiuto al Cassiere.
2. Il Cassiere chiede al Cliente un metodo di pagamento alternativo.

4. Il Sistema registra il **pagamento con carta di credito**, che comprende l'approvazione del pagamento.

5. Il Sistema presenta il meccanismo di inserimento della firma per il pagamento.

6. Il Cassiere chiede al Cliente la firma per la ricevuta del pagamento con carta di credito. Il Cliente inserisce la firma.

7c. Pagamento con assegno:

1. Il Cliente compila un **assegno** e lo consegna, insieme alla **carta d'identità**, al Cassiere.
2. Il Cassiere scrive il numero della carta d'identità, lo inserisce e richiede un'**autorizzazione al pagamento con assegno**.

3. Il Sistema genera una **richiesta di pagamento con assegno** e la invia a un **Servizio di Autorizzazione Assegni esterno**.

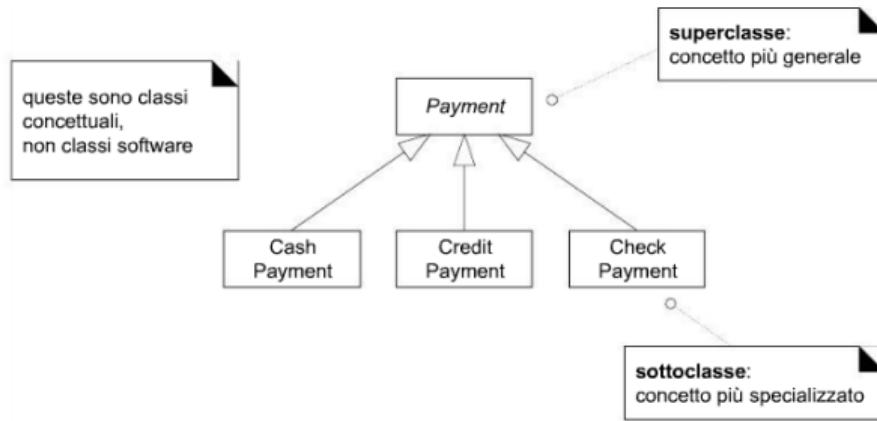
4. Il Sistema riceve un'approvazione del pagamento con assegno e segnala l'approvazione al Cassiere.

5. Il Sistema registra il **pagamento con assegno**, che comprende l'approvazione del pagamento.

...

# Generalizzazione

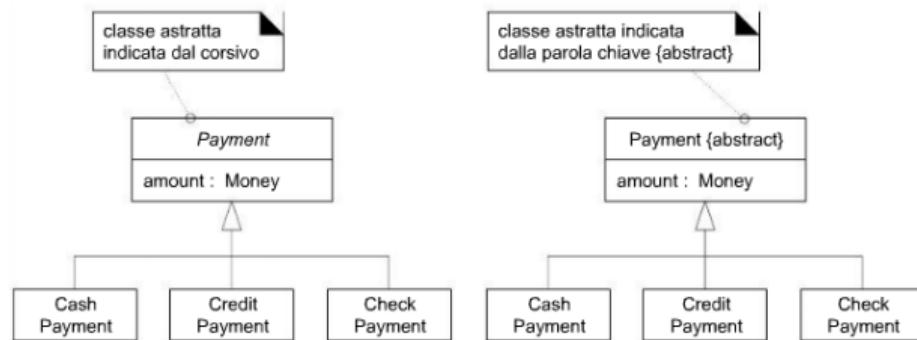
La **generalizzazione** è un'astrazione basata sull'identificazione di *caratteristiche comuni tra concetti*, che porta a definire una relazione tra un concetto più generale (*superclasse*) e un concetto più specializzato o specifico (*sottoclasse*). Vale il principio di **sostituibilità**.



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

# Generalizzazione

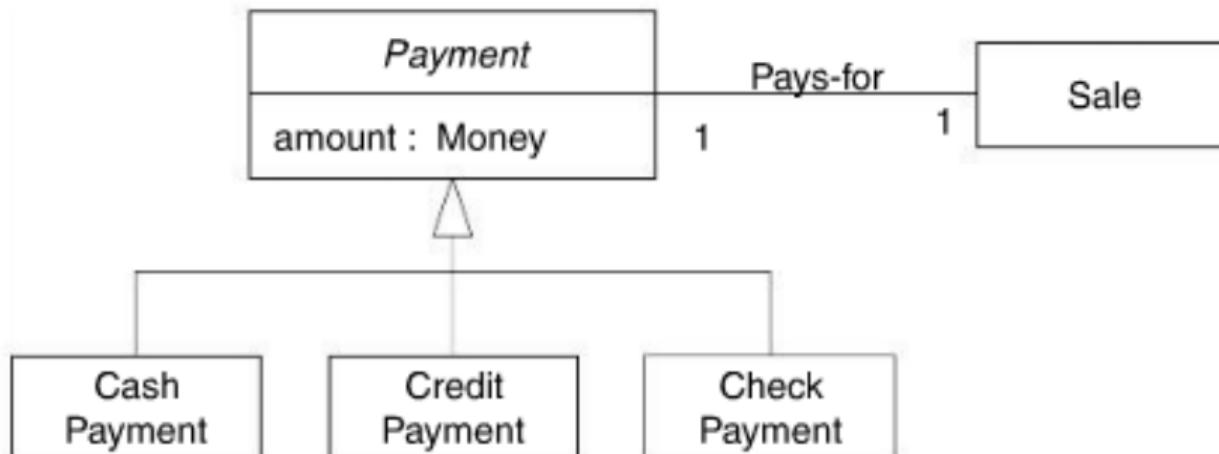
La **generalizzazione** è un'astrazione basata sull'identificazione di *caratteristiche comuni tra concetti*, che porta a definire una relazione tra un concetto più generale (*superclasse*) e un concetto più specializzato o specifico (*sottoclasse*). Vale il principio di **sostituibilità**.



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

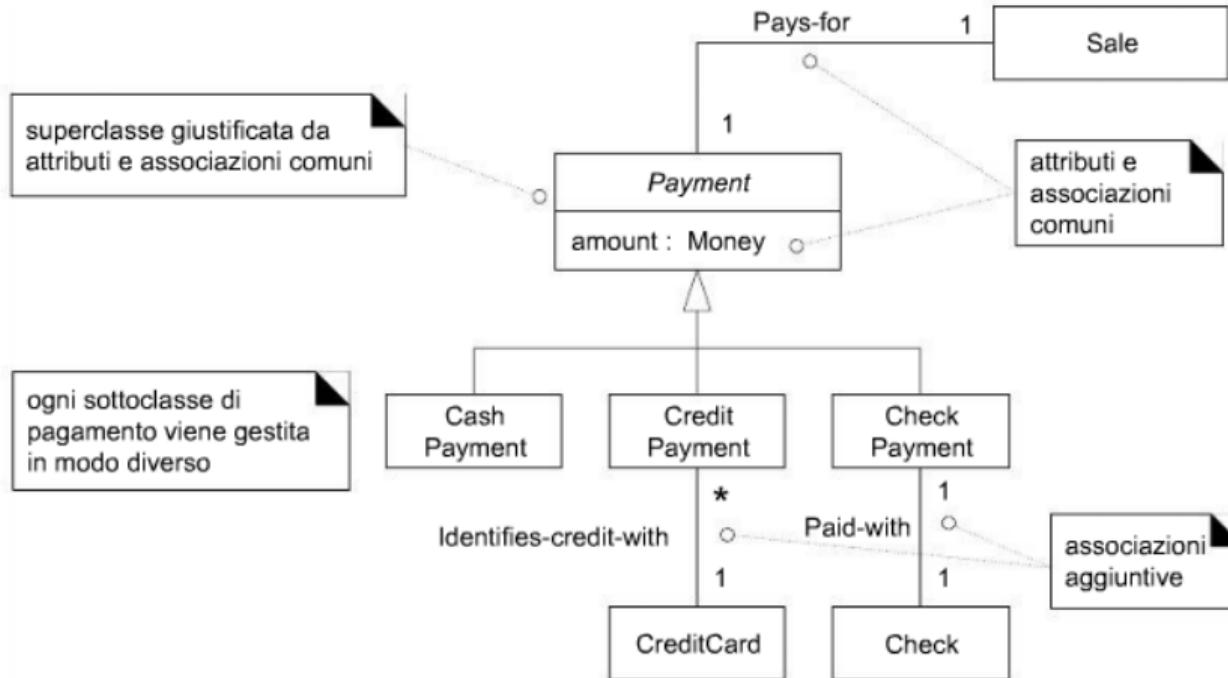
Una classe concettuale è **astratta** se ciascun suo elemento è anche elemento di una delle sue sottoclassi.

## Conformità di una sottoclassificazione concettuale



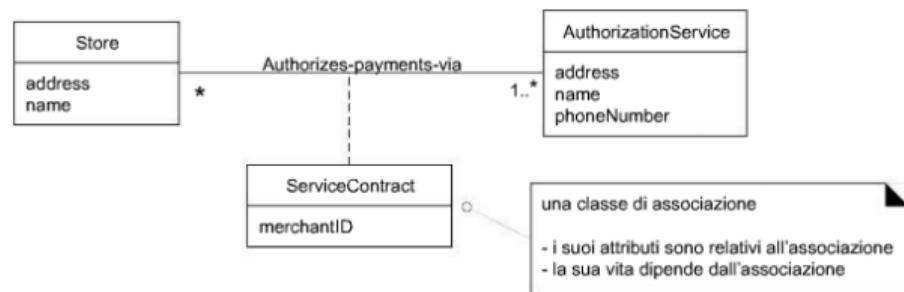
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Associazioni aggiuntive in una sottoclasse concettuale



# Classi di associazioni

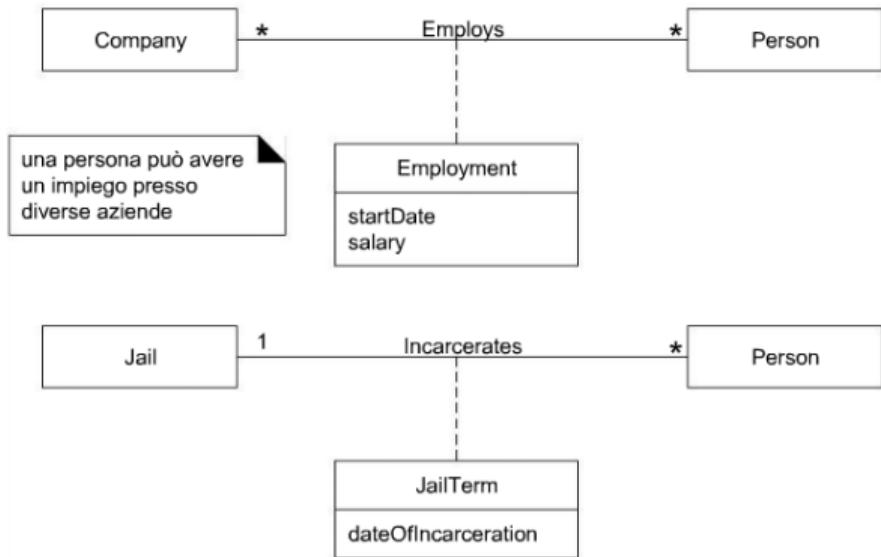
Permettono di aggiungere attributi e altre caratteristiche alle associazioni.



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

# Classi di associazioni

Permettono di aggiungere attributi e altre caratteristiche alle associazioni.



# 06 . Diagrammi di Sequenza di Sistema (SSD)

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

### **Si noti che**

questi lucidi sono basati sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016” e sul materiale fornito dai docenti Viviana Bono, Claudia Picardi e Gianluca Torta dell’Università degli Studi di Torino che hanno tenuto il corso negli anni accademici precedenti.

# Table of contents

---

1. Disciplina dei requisiti: Diagrammi di Sequenza di Sistema
2. Un po' di notazione



# **Disciplina dei requisiti: Diagrammi di Sequenza di Sistema**

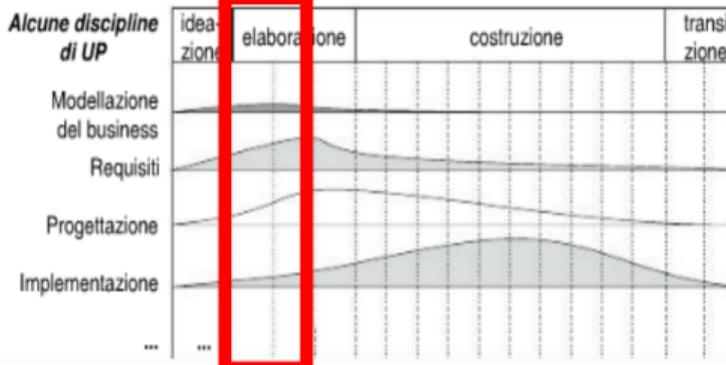
---

# UP maps

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

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

## Alcune discipline di UP



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

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

# Diagramma di sequenza di sistema

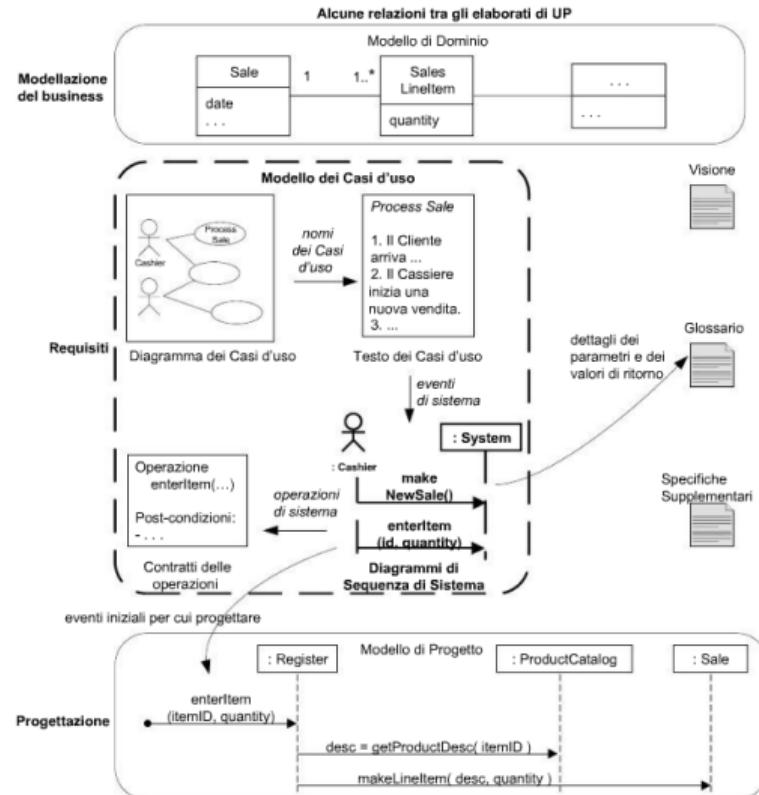
## Il Diagramma di sequenza di sistema (SSD)

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

Nota: non è menzionato esplicitamente in UP.

- I diagrammi di sequenza di sistema sono espressi attraverso i **diagrammi di sequenza di UML**
- Il sistema è modellato come una “*scatola nera*”
- Usualmente si modella un SSD per ogni caso d'uso per lo scenario principale e per ogni scenario alternativo
- Lo SSD costituisce un input per i **contratti** delle operazioni e, soprattutto, per la progettazione degli oggetti

# Relazioni tra elaborati di UP



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

## 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 è qualcosa di importante o degno di nota che avviene durante l'esecuzione di un sistema
- Un evento di sistema è un evento esterno al sistema, di input, di solito generato da un attore per interagire con il sistema

# Diagrammi di sequenza di sistema

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

## Diagrammi di sequenza di sistema

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

Nota: la qualifica “di sistema” enfatizza l’applicazione dei diagrammi di sequenza UML ai sistemi, considerati a *scatola nera*.

# Eventi di un sistema software

Un sistema reagisce a tre cose:

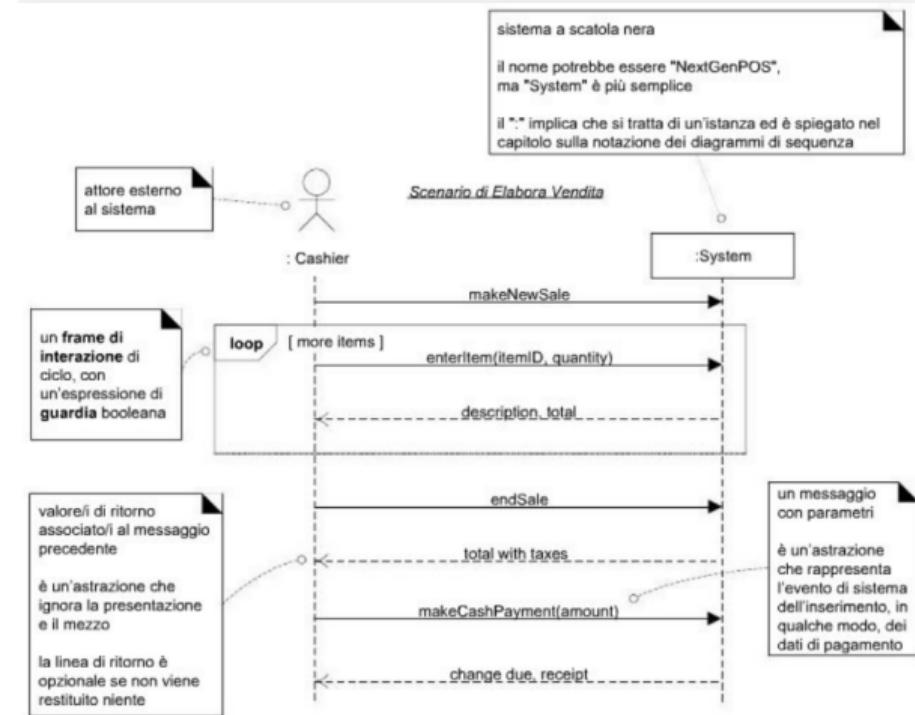
- **eventi esterni** da parte di attori umani o sistemi informatici
- **eventi temporali**
- **guasti o eccezioni**

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

# Esempio di SSD

Eventi di sistema:

- *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



## SSD e casi d'uso

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

Un SSD mostra:

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

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

# Gli SSD derivano dai casi d'uso

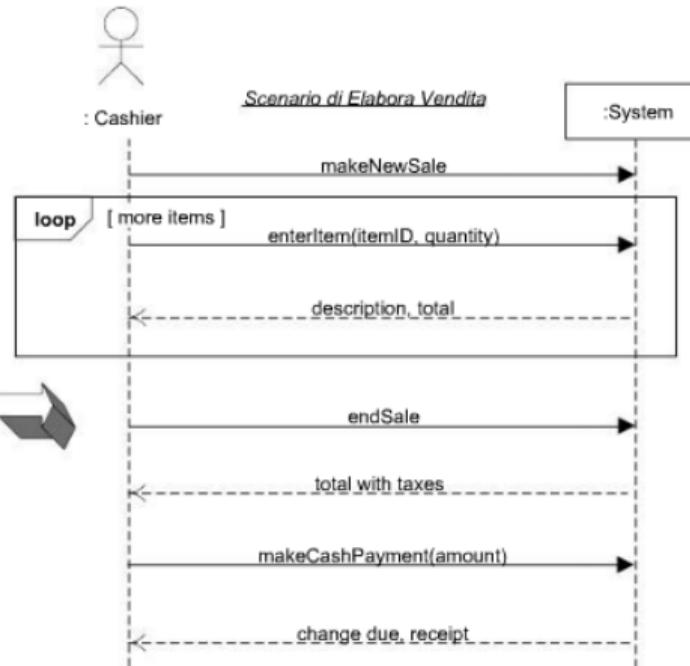
Scenario di base di *Elabora Vendita*, con pagamento in contanti:

1. Il Cliente arriva alla cassa POS con gli articoli e/o i servizi da acquistare.
2. Il Cassiere inizia una nuova vendita.
3. Il Cassiere inserisce il codice identificativo di un articolo.
4. Il Sistema registra la riga di vendita per l'articolo e mostra una descrizione dell'articolo, il suo prezzo e il totale parziale.  
*Il Cassiere ripete i passi 3-4 fino a che non indica che ha terminato.*
5. Il Sistema mostra il totale.
6. Il Cassiere riferisce il totale al Cliente, e richiede il pagamento.
7. Il Cliente paga (in contanti) e il sistema gestisce il pagamento.
8. Il Sistema registra la vendita completata.
9. Il Sistema genera la ricevuta.
10. Il Cliente va via con la ricevuta e gli articoli acquistati.

# Gli SSD derivano dai casi d'uso

## Scenario di base di Elabora Vendita, con pagamento in contanti:

1. Il Cliente arriva alla cassa POS con gli articoli e/o i servizi da acquistare.
2. Il Cassiere inizia una nuova vendita.
3. Il Cassiere inserisce il codice identificativo di un articolo.
4. Il Sistema registra la riga di vendita per l'articolo e mostra una descrizione dell'articolo, il suo prezzo e il totale parziale.  
*Il Cassiere ripete i passi 3-4 fino a che non indica che ha terminato.*
5. Il Sistema mostra il totale.
6. Il Cassiere riferisce il totale al Cliente, e richiede il pagamento.
7. Il Cliente paga (in contanti) e il sistema gestisce il pagamento.
8. Il Sistema registra la vendita completata.
9. Il Sistema genera la ricevuta.
10. Il Cliente va via con la ricevuta e gli articoli acquistati.

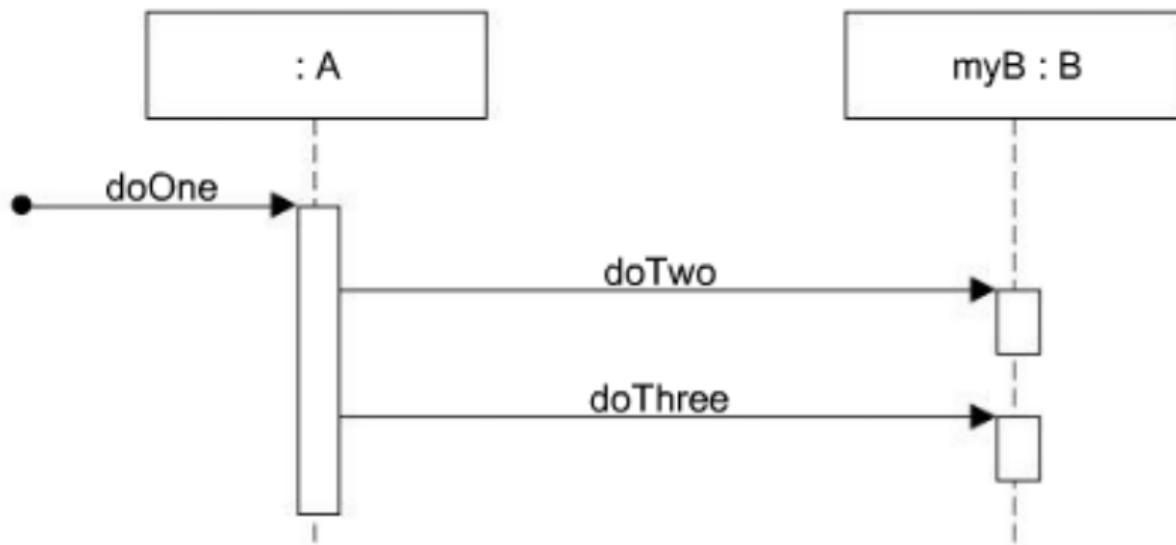




## **Un po' di notazione**

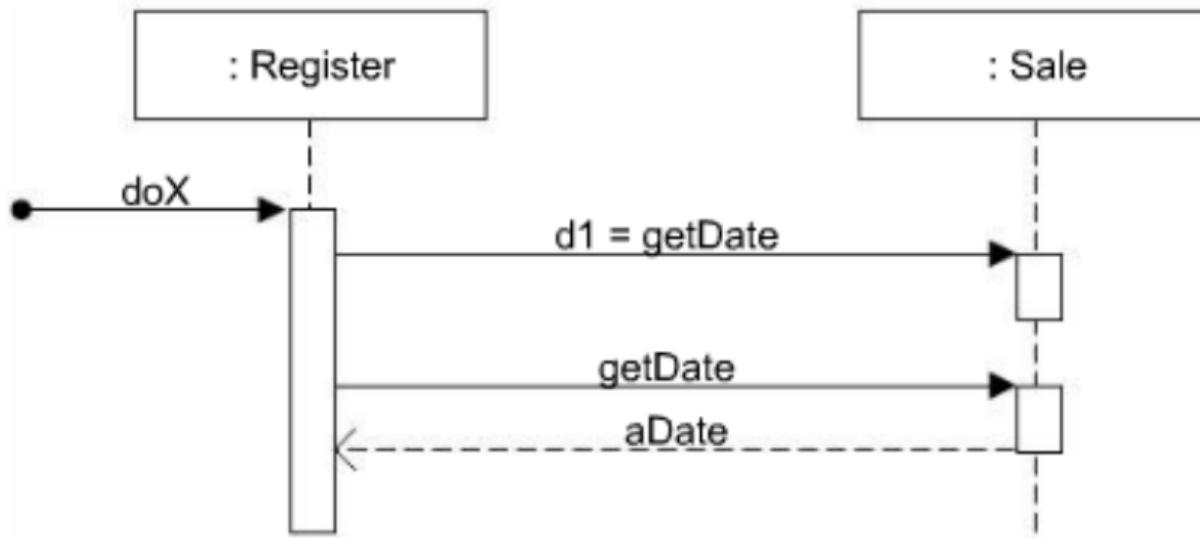
---

## Semplice diagramma di sequenza



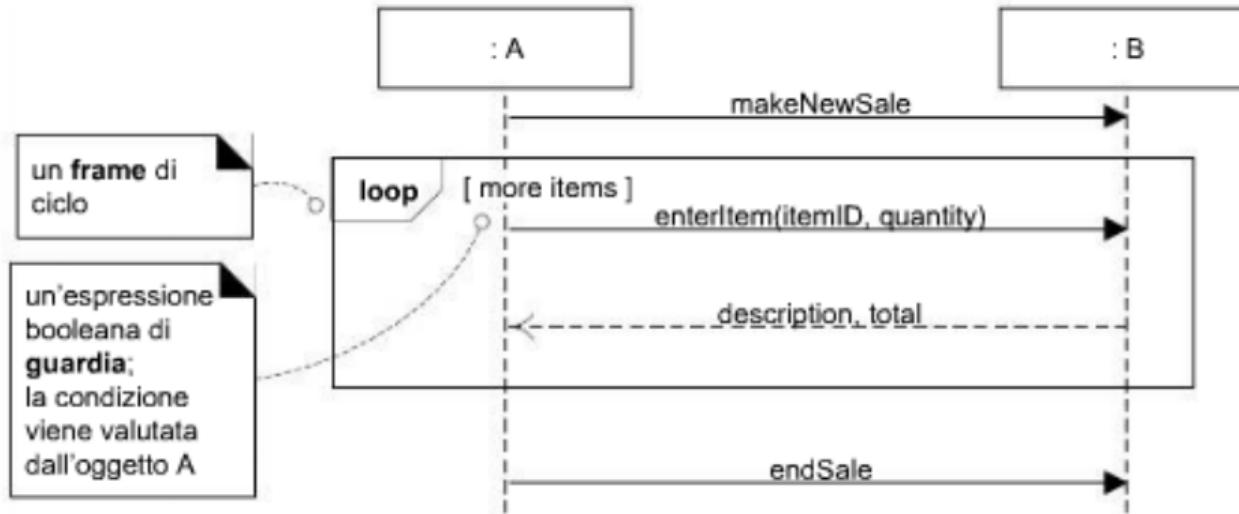
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

## Due modi per mostrare un risultato di ritorno da un messaggio



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

# Esempio di frame di UML

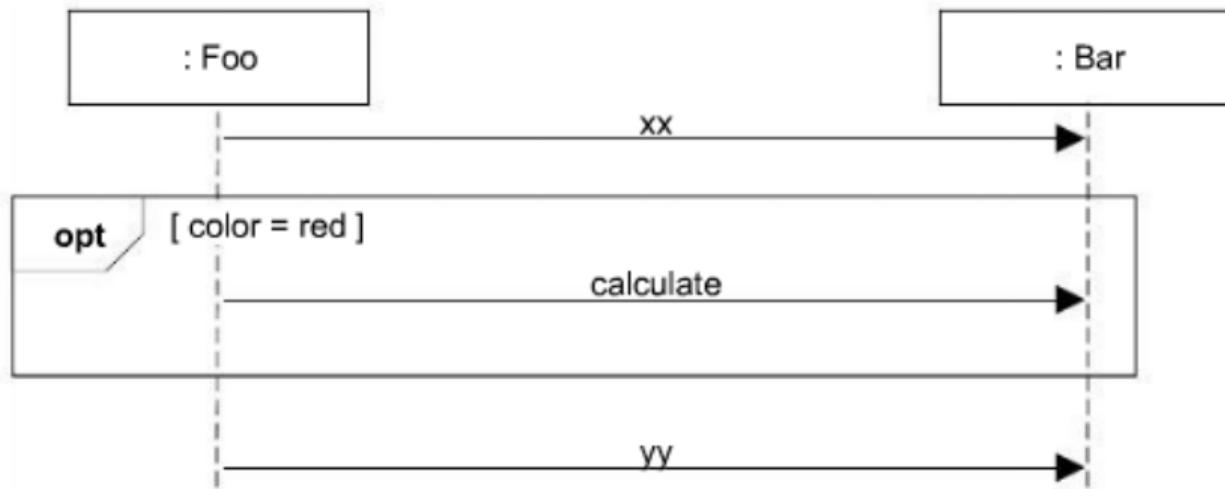


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Operatori comuni per i frame di UML

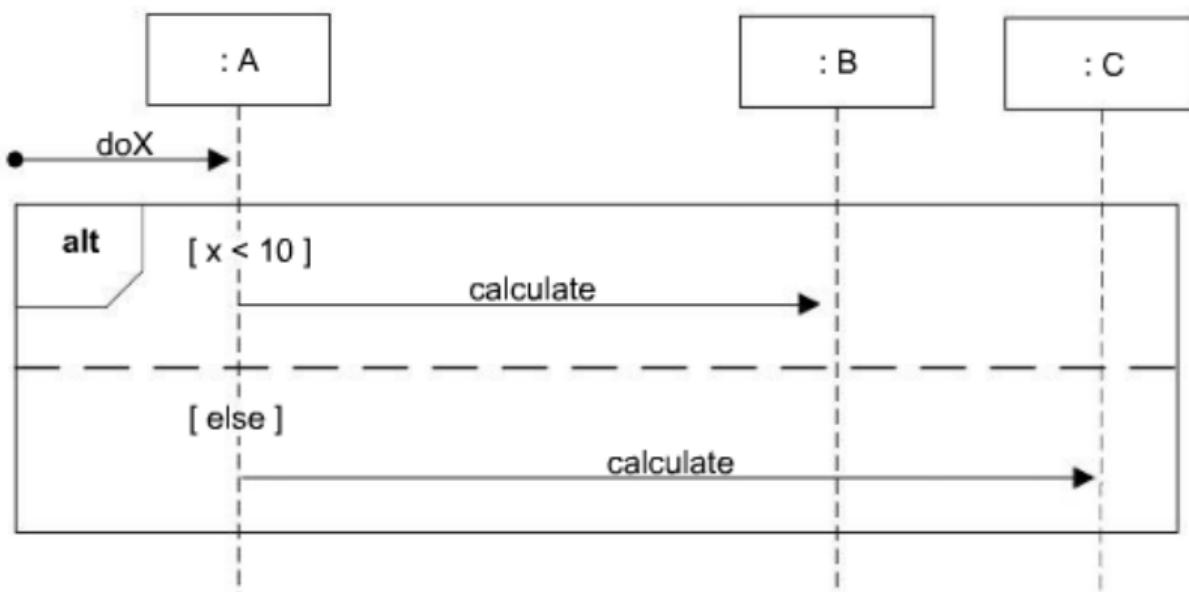
Operatore frame	Significato
alt	Frammento alternativo per logica mutuamente espressa nella guardia (un'istruzione <i>if-else</i> di Java o del C).
opt	Frammento opzionale che viene eseguito se la guardia è vera (un'istruzione <i>if</i> ).
loop	Frammento da eseguire ripetutamente finché la guardia è vera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

# Un messaggio condizionale



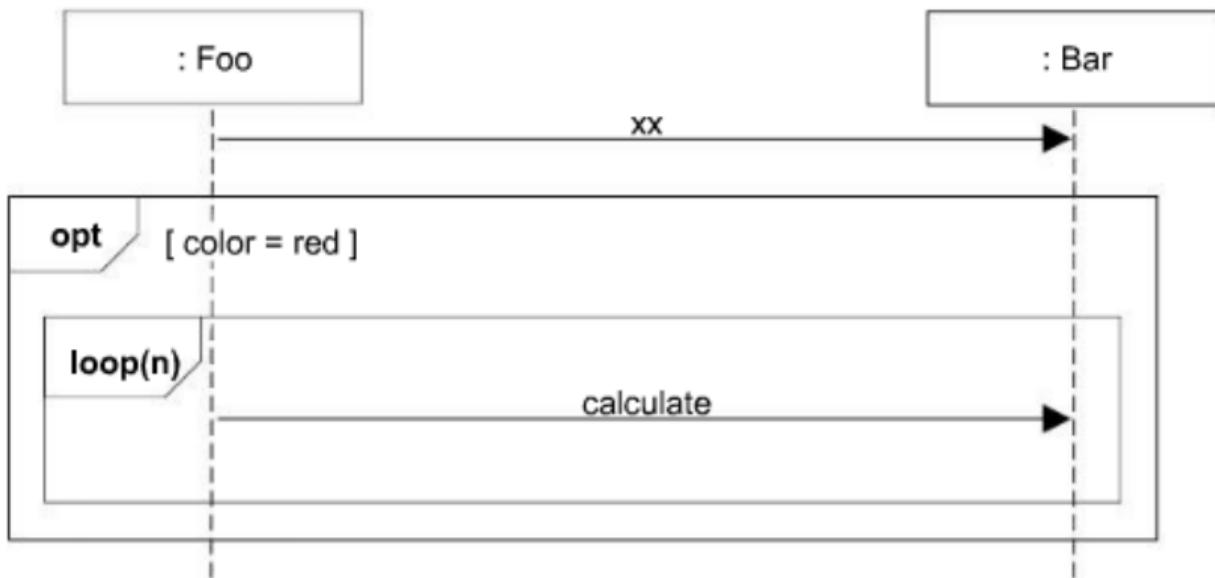
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Messaggi condizionali mutuamente esclusivi



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

# Annidamento di frame



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

# 07 . Contratti

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

### **Si noti che**

questi lucidi sono basati sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016” e sul materiale fornito dai docenti Viviana Bono, Claudia Picardi e Gianluca Torta dell’Università degli Studi di Torino che hanno tenuto il corso negli anni accademici precedenti.

# Table of contents

---

- 
1. Disciplina dei requisiti: Contratti
  2. Post-condizioni e pre-condizioni
  3. Scrivere contratti



## **Disciplina dei requisiti: Contratti**

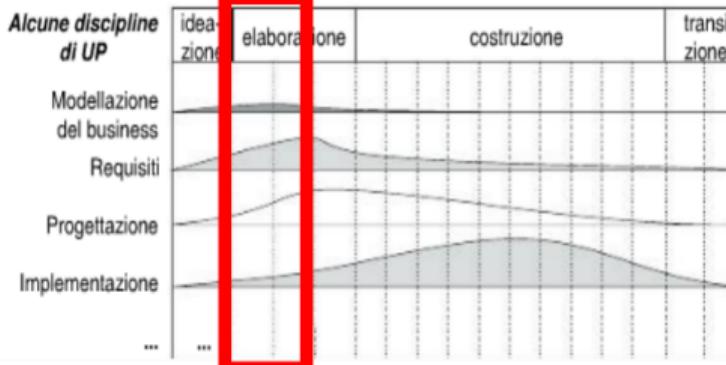
---

# UP maps

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

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

## Alcune discipline di UP



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

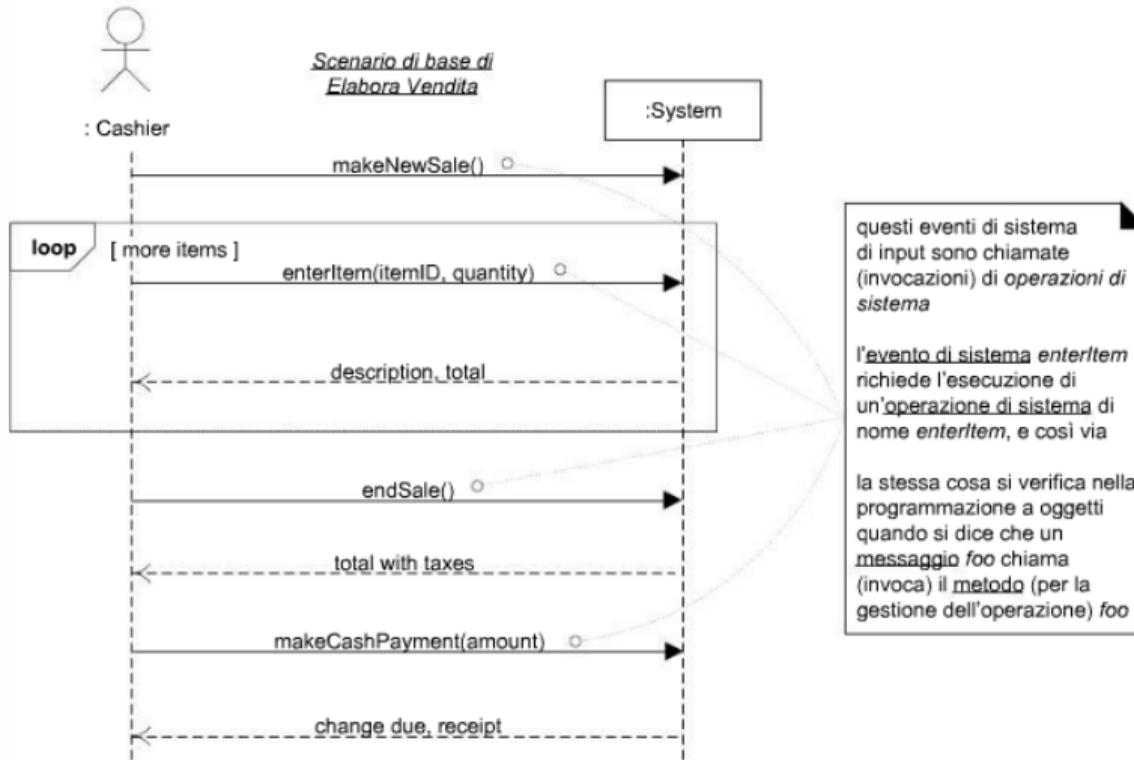
Questo esempio è solo un suggerimento, non è da prendere alla lettera.

- Le operazioni di sistema sono operazioni che il sistema, considerato come un componente a scatola nera, offre nella sua interfaccia pubblica.
- Le operazioni di sistema possono essere identificate mentre si abbozzano gli SSD. Gli SSD mostrano eventi di sistema.

## Eventi e operazioni

Un evento di sistema implica che il sistema definisca un'operazione di sistema per gestire quell'evento.

# Eventi e operazioni di sistema



## I contratti delle operazioni

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

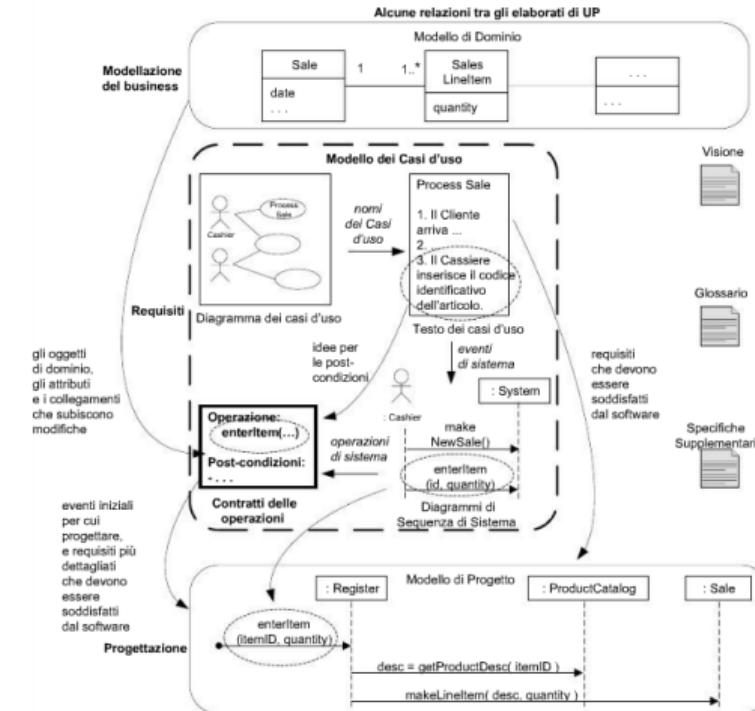
Nota: possono essere considerati parte del Modello dei Casi d'Uso, poiché forniscono maggiori dettagli dell'analisi sull'effetto delle operazioni di sistema implicate dai casi d'uso.

Non sono menzionati esplicitamente in UP.

# Relazioni tra elaborati di UP

I principali input per i contratti sono le operazioni di sistema identificate negli SSD, il modello di dominio, e l'esame del dominio da parte degli esperti.

I contratti servono come input per la progettazione a oggetti.



### 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*).

## Sezioni di un contratto - Template

- **Operazione:** Nome e parametri (*firma*) dell'operazione.
- **Riferimenti:** Casi d'uso in cui può verificarsi questa operazione.
- **Pre-condizioni:** Ipotesi significative sullo stato del sistema o degli oggetti nel Modello di Dominio prima dell'esecuzione dell'operazione. Si tratta di ipotesi non banali, che dovrebbero essere comunicate al lettore.
- **Post-condizione:** È la sezione più importante. Descrive i cambiamenti di stato degli oggetti nel Modello di Dominio dopo il completamento dell'operazione.

## **Post-condizioni e pre-condizioni**

---

# Definizione di post-condizione

## Le post-condizioni

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

Le post-condizioni **non sono azioni da eseguire nel corso dell'operazione**; si tratta di **osservazioni** sugli oggetti del modello di dominio che risultano al **termine** dell'operazione.

# Definizione di post-condizione

## Le post-condizioni

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

Le post-condizioni **non sono azioni da eseguire nel corso dell'operazione**; si tratta di **osservazioni** sugli oggetti del modello di dominio che risultano al **termine** dell'operazione.

Nota: è un punto di vista concettuale, non di implementazione: si possono creare gli oggetti corrispondenti alle classi del **Modello di Dominio**, come si possono formare i collegamenti corrispondenti alle associazioni del **Modello di Dominio**.

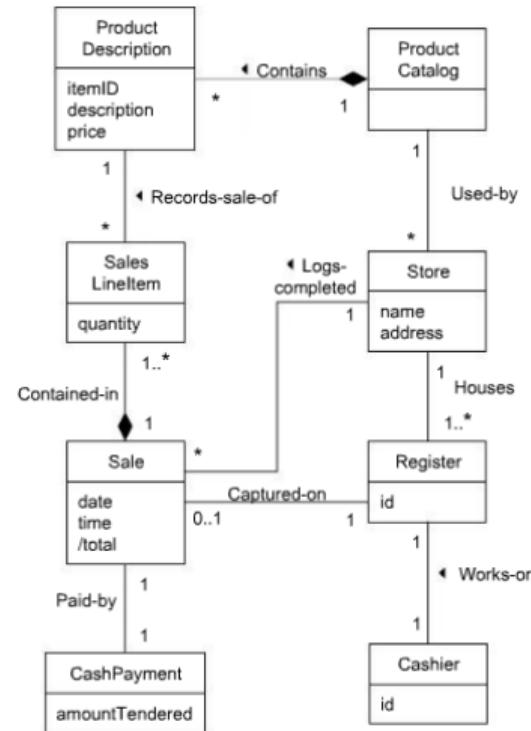
# Adottare un punto di vista concettuale e della conoscenza

## Il contratto per un'operazione di sistema

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

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

A lato il Modello di Dominio di riferimento negli esempi di seguito.



## Un contratto

è uno strumento ottimo nell'analisi dei requisiti o nell'analisi orientata agli oggetti per descrivere in modo molto dettagliato i cambiamenti richiesti dall'esecuzione di una operazione di sistema (in termini di oggetti del Modello di Dominio), senza però descrivere come devono essere ottenuti questi cambiamenti.

La progettazione si può rimandare e ci si può concentrare sull'analisi di che cosa deve accadere, anziché su come deve essere ottenuto.

# Contratti: che cosa vs come

## Contratto CO2: enterItem

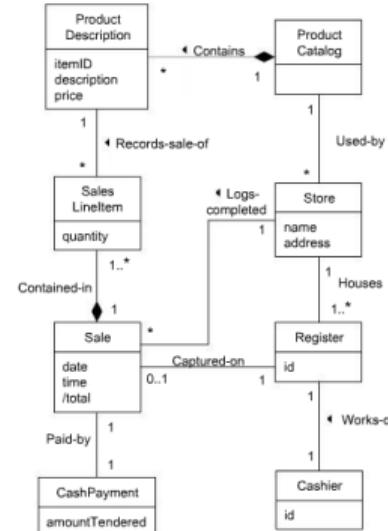
**Operazione:** enterItem(itemID: ItemID, quantity: integer)

**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*).



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

## Si noti che

non viene fatto alcun commento su come venga creata l'istanza *SalesLineItem* o su come venga associata a una *Sale*.

# Contratti: che cosa vs come

## Contratto CO2: enterItem

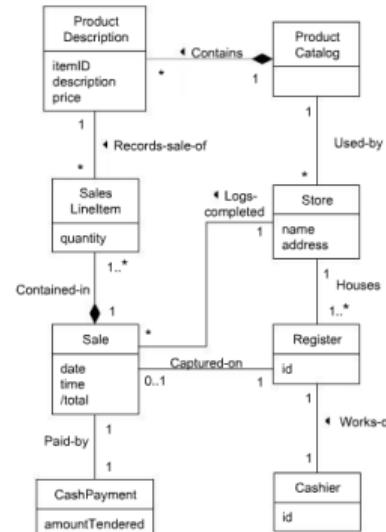
**Operazione:** enterItem(itemID: ItemID, quantity: integer)

**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*).



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

**Le post-condizioni vanno espresse con il tempo del verbo al passato**

“È stata creata una SalesLineItem” piuttosto che “Creare una SalesLineItem” oppure “Viene creata una SalesLineItem”.

# Contratti: che cosa vs come

## Contratto CO2: enterItem

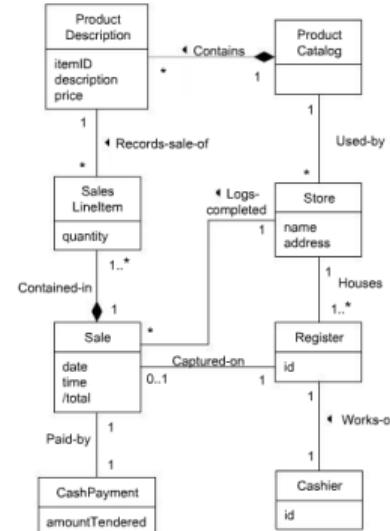
**Operazione:** enterItem(itemID: ItemID, quantity: integer)

**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*).



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

## Si noti l'assegnazione

di un nome all'istanza, questo permetterà di fare riferimento alla nuova istanza nelle altre post-condizioni.

# Contratti: che cosa vs come

## Contratto CO2: enterItem

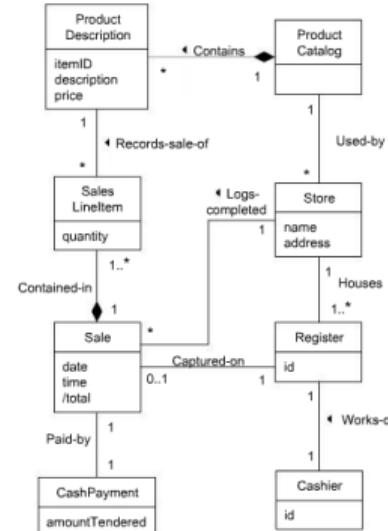
**Operazione:** enterItem(itemID: ItemID, quantity: integer)

**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*).



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

## Formazione o rottura di collegamenti

La nuova *SalesLineItem* deve essere stata collegata alla sua *Sale* corrente e collegata anche a una sua *ProductDescription*.

# Contratti: che cosa vs come

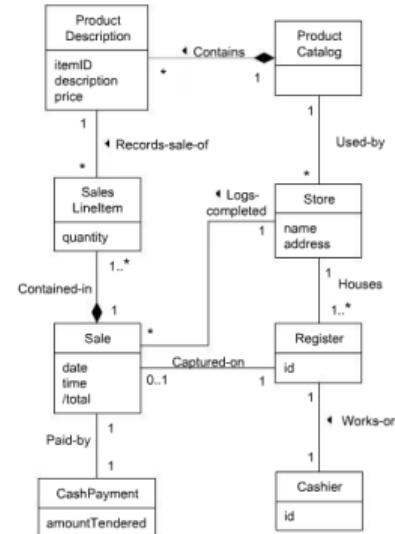
## Contratto CO2: enterItem

**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*).



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

## Modifica di attributi

La *quantity* della *SalesLineItem* creata deve essere diventata pari al parametro *quantity*.

# Definizione di pre-condizione

## Le pre-condizioni

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

Una descrizione sintetica dello stato di avanzamento del caso d'uso.

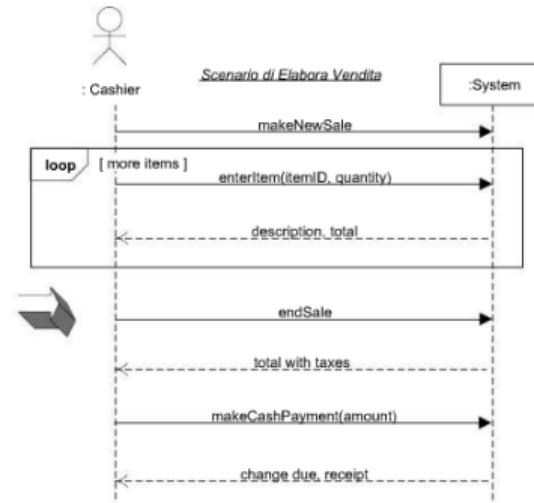
# Definizione di pre-condizione

## Le pre-condizioni

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

Una descrizione sintetica dello stato di avanzamento del caso d'uso.

Per esempio, l'operazione *enterItem* viene eseguita quando una vendita è già iniziata ma non è ancora conclusa. Lo stato di avanzamento del caso d'uso si può riassumere con la pre-condizione “*è in corso una vendita*”.



# Definizione di pre-condizione

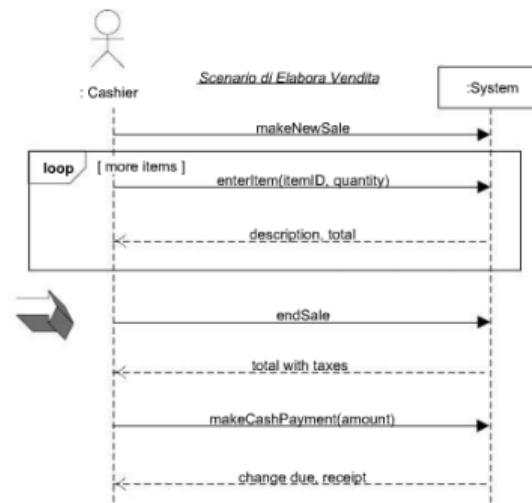
## Le pre-condizioni

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

Una descrizione sintetica dello stato di avanzamento del caso d'uso.

Per esempio, l'operazione *enterItem* viene eseguita quando una vendita è già iniziata ma non è ancora conclusa. Lo stato di avanzamento del caso d'uso si può riassumere con la pre-condizione “*è in corso una vendita*”.

Nota: nelle pre-condizioni è utile indicare gli *oggetti rilevanti a quel punto* del caso d'uso, in particolare quelli che si vogliono citare nelle post-condizioni. Ad esempio, “*sì è stata associata con la Sale corrente s*”, la vendita corrente *s* va citata nelle pre-condizioni.



## **Scrivere contratti**

---

# Come creare e scrivere i contratti

---

Si proceda come segue:

1. Identificare le operazioni di sistema dagli SSD
2. Creare un contratto per le operazioni di sistema complesse o i cui effetti sono probabilmente sottili, o che non sono chiare dai casi d'uso
3. Per descrivere le post-condizioni si utilizzino le seguenti sotto-categorie:
  - creazione o cancellazione di oggetto (o istanza)
  - formazione o rottura di collegamento
  - modifica di attributo

# Esempio: contratti per POS NextGen, caso d'uso Elabora Vendita

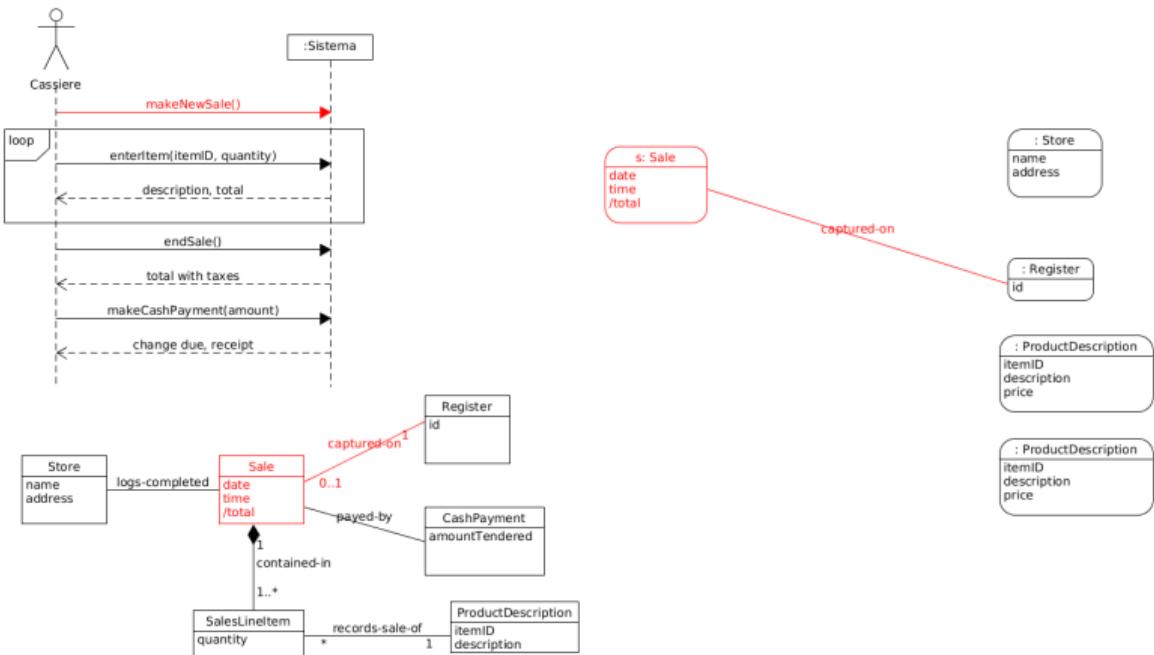
## Contratto CO1: makeNewSale

Operazione: makeNewSale()  
 Riferimenti: casi d'uso: Elabora Vendita  
 Pre-condizioni: nessuna

Post-condizioni: – è stata creata un'istanza s di Sale (creazione di oggetto).  
 – s è stata associata con il Register (formazione di collegamenti)  
 – gli attributi di s sono stati inizializzati (modifica di attributi).

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

2016.



# Esempio: contratti per POS NextGen, caso d'uso Elabora Vendita

## Contratto C02: enteritem

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

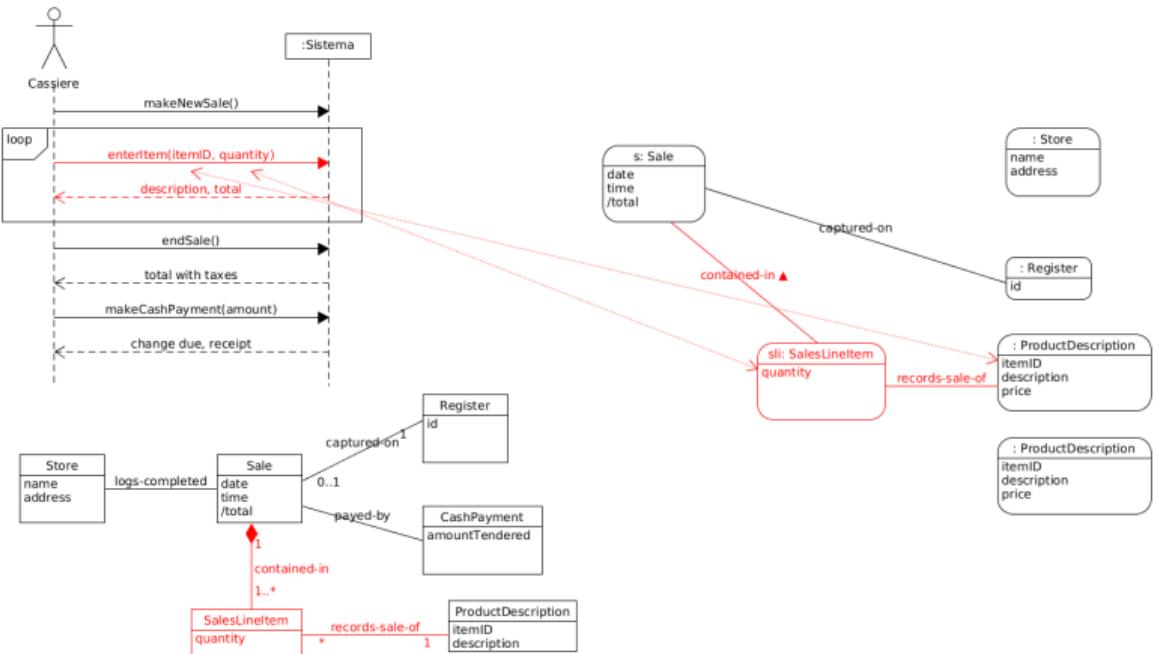
**Riferimenti:** casi d'uso: Elabora Vendita

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

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

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

2016.



# Esempio: contratti per POS NextGen, caso d'uso Elabora Vendita

## Contratto C02: enteritem

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

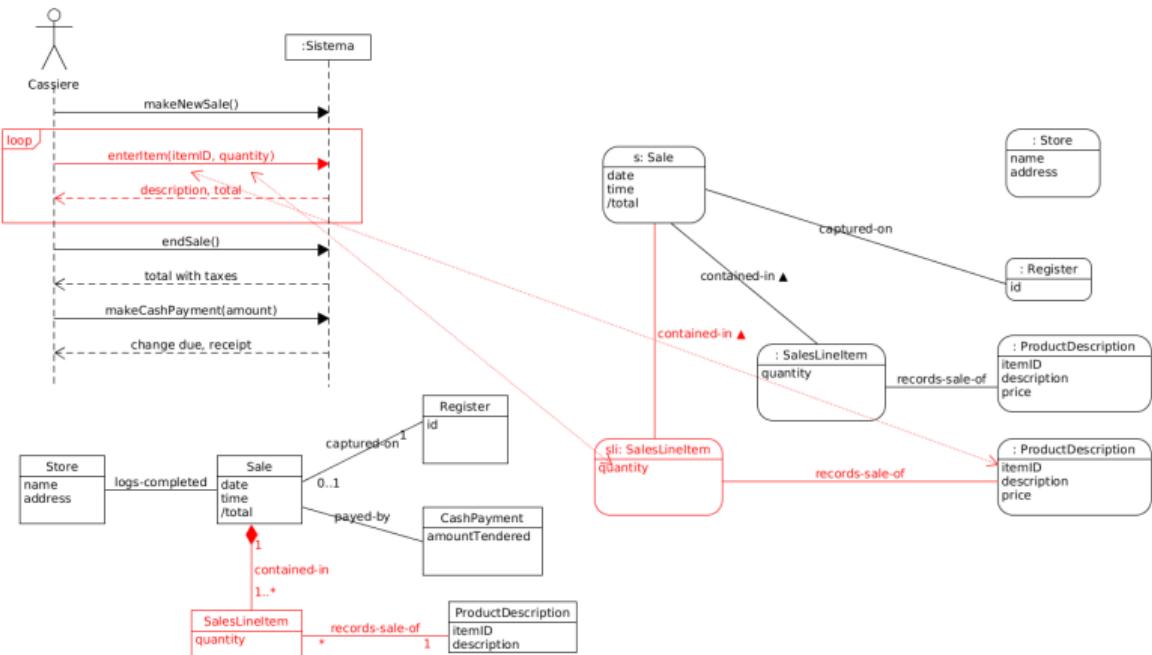
**Riferimenti:** casi d'uso: Elabora Vendita

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

- Post-condizioni:**
- è stata creata un'istanza sli di SalesLineItem (creazione di q)
  - 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).

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

2016.



# Esempio: contratti per POS NextGen, caso d'uso Elabora Vendita

*Di questo contratto ne  
parliamo tra poco...*

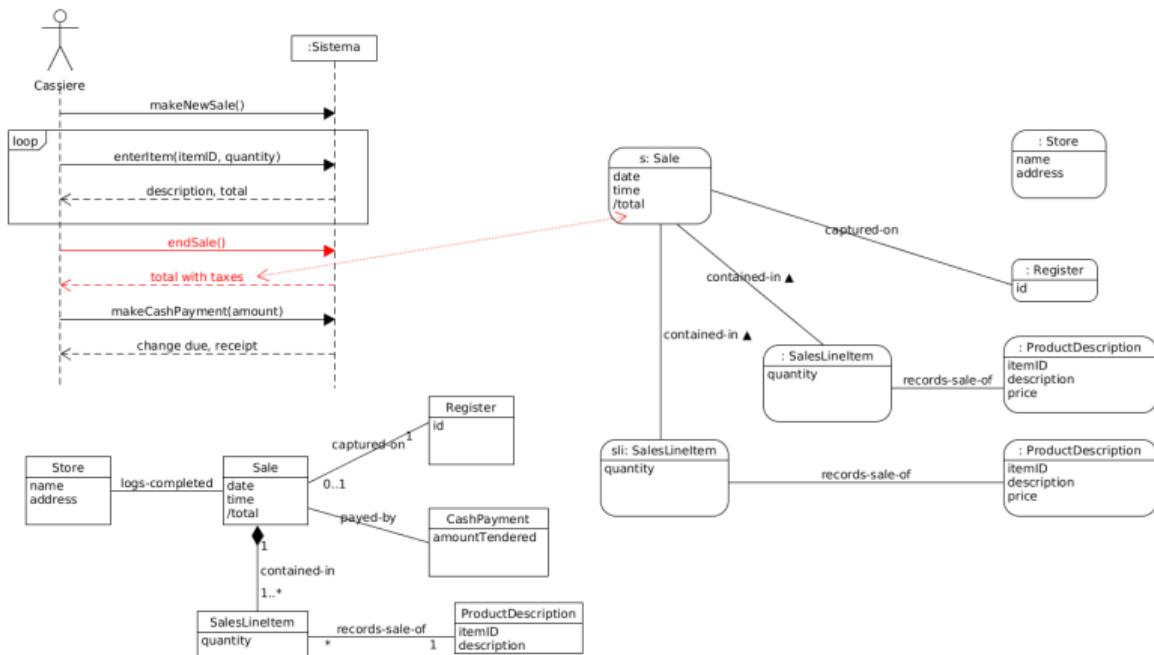
## Contratto C03: endSale

Operazione: endSale()  
Riferimenti: casi d'uso: Elabora Vendita  
Pre-condizioni: è in corso una vendita s.

Post-condizioni: – nessuna.

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

2016.



# Esempio: contratti per POS NextGen, caso d'uso Elabora Vendita

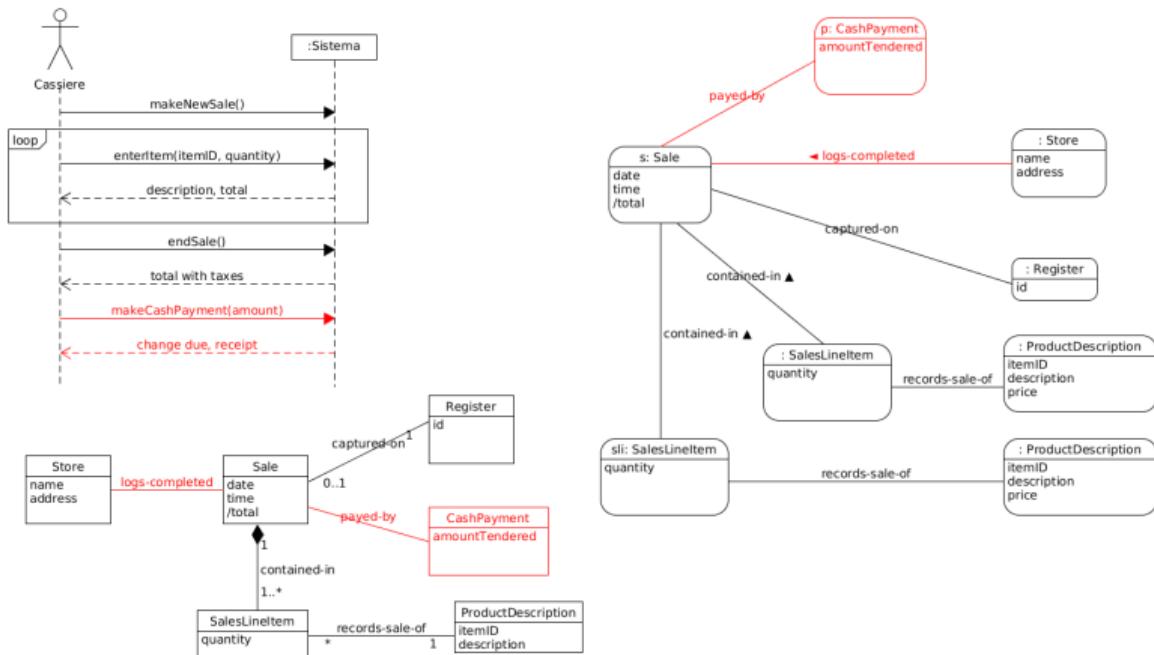
## Contratto CO4: makeCashPayment

**Operazione:** makeCashPayment( amount: Money )

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

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

- Post-condizioni:**
- è stata creata un'istanza p di CashPayment (*creazione di oggetto*).
  - p è stata associata con la Sale (vendita) corrente s (*formazione collegamento*).
  - p.amountTendered è diventato amount (*modifica di attributo*).
  - la Sale corrente s è stata associata con lo Store (*formazione collegamento*); (s è stata aggiunta al registro storico delle vendite).



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

2016.

## Trasformazione e interrogazione

---

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

# Trasformazione e interrogazione

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

## Un'operazione di sistema

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

# Esempio: contratti per POS NextGen, caso d'uso Elabora Vendita

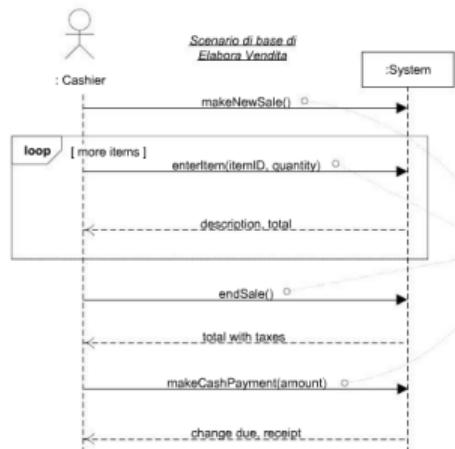
## Contratto CO3: endSale

**Operazione:** endSale()

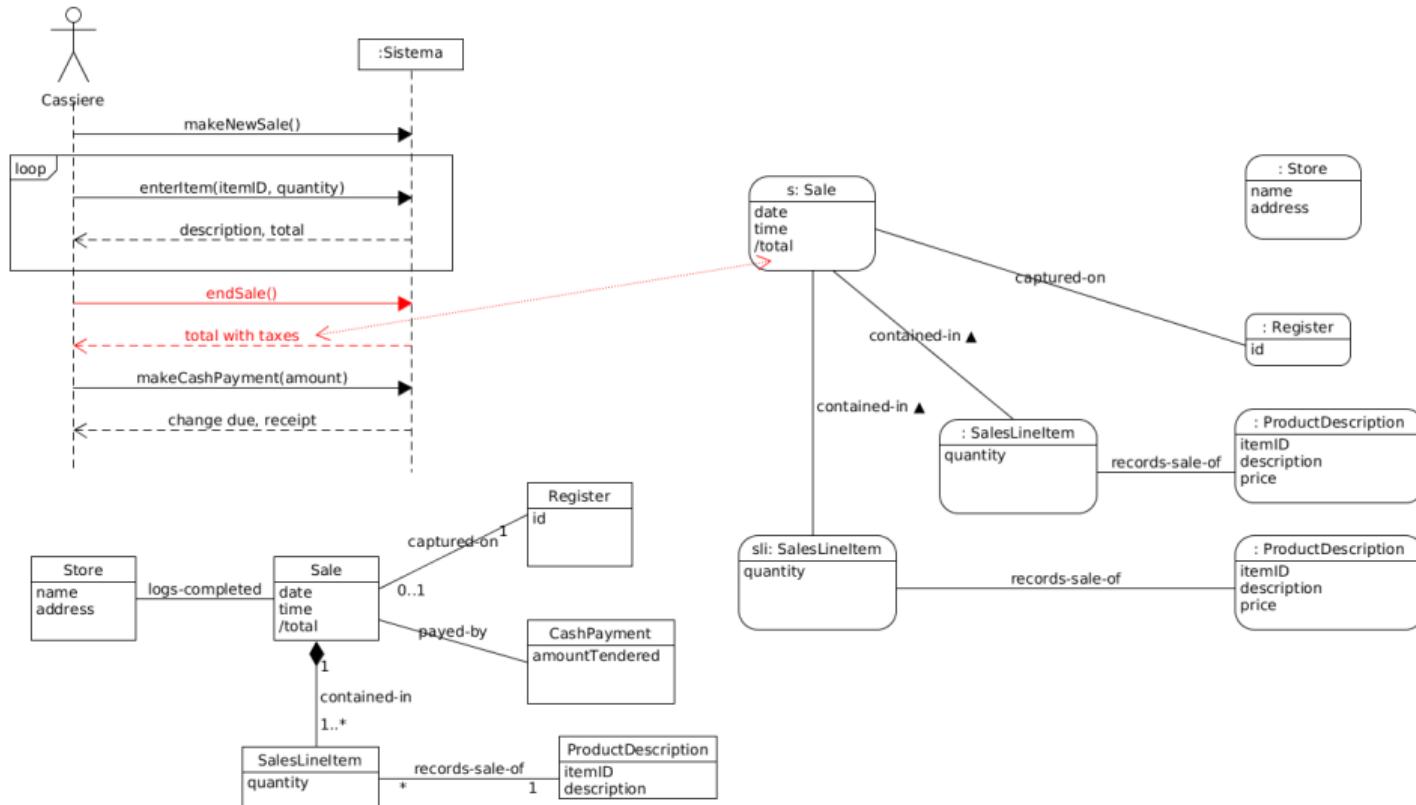
**Riferimenti:** casi d'uso: Elabora Vendita

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

**Post-condizioni:** – nessuna.



# Esempio: contratti per POS NextGen, caso d'uso Elabora Vendita



## Come creare e scrivere i contratti

- Bisogna scrivere un contratto per ogni evento di sistema trovato nel SSD?
- Se si scoprono nuove classi, attributi, si possono aggiungere nel modello di dominio?
- Le post-condizioni devono essere in ogni momento le più complete possibili ?

## Come creare e scrivere i contratti

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

## Come creare e scrivere i contratti

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

## Come creare e scrivere i contratti

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

# **08 . Architettura logica e organizzazione in layer**

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

### **Si noti che**

questi lucidi sono basati sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016” e sul materiale fornito dai docenti Viviana Bono, Claudia Picardi e Gianluca Torta dell’Università degli Studi di Torino che hanno tenuto il corso negli anni accademici precedenti.

# Table of contents

---

1. Architettura logica
2. Strato del dominio
3. Principio di separazione Modello-Vista

## Architettura logica

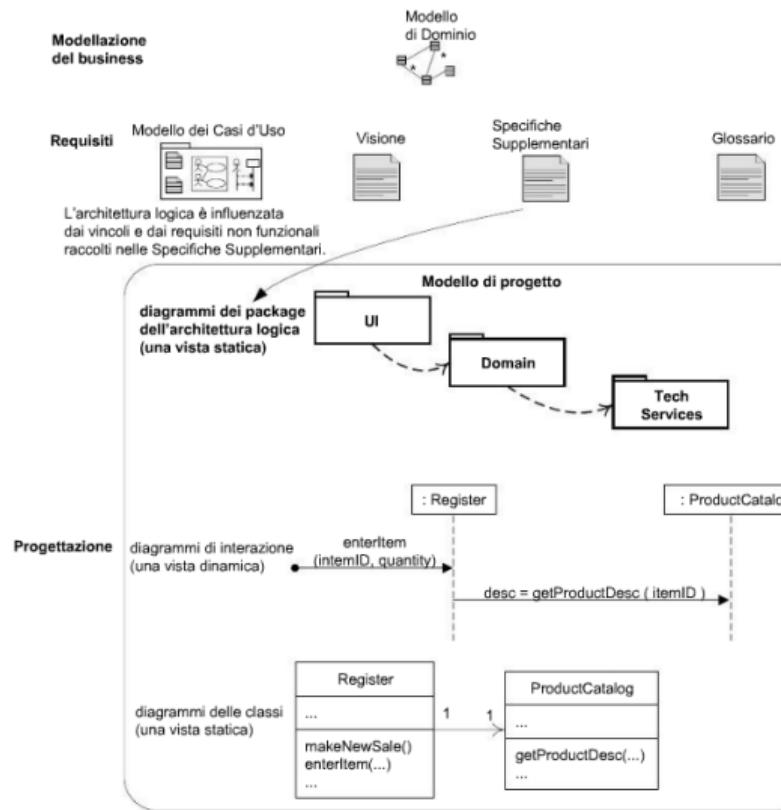
---

## Architettura

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

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

# Alcune relazioni tra gli elaborati di UP



## Architettura logica

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

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

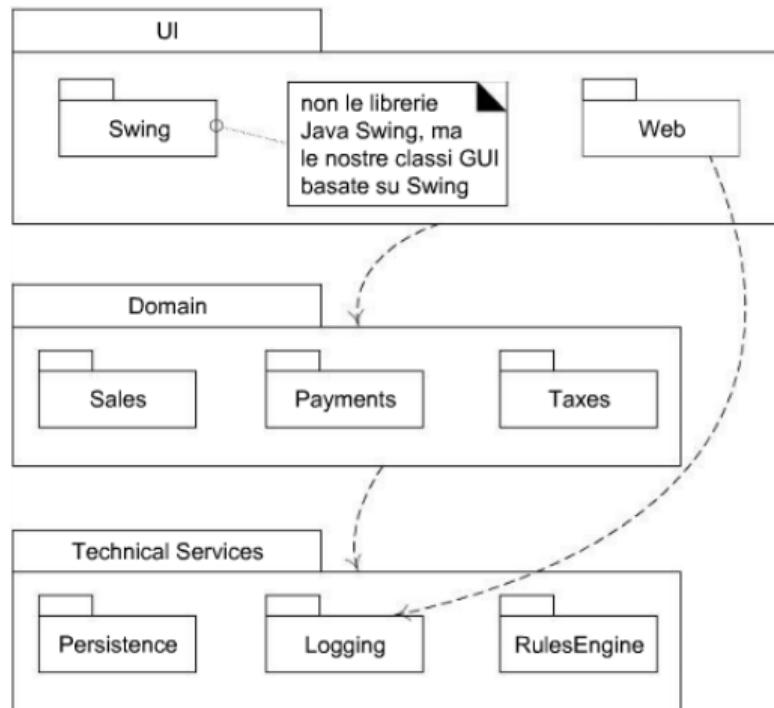
## Layer o strato

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

Gli strati di un'applicazione software comprendono normalmente:

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

# Esempio dell'architettura per POS NextGen in diagrammi di package 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

## Architettura a strati

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

## Architettura a strati

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

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

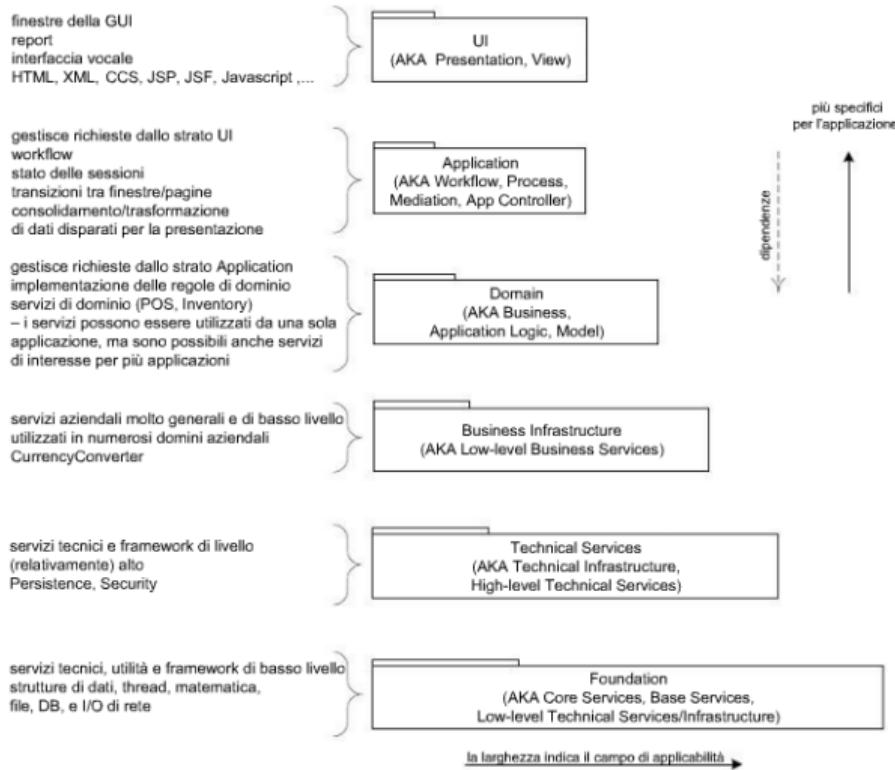
## Architettura a strati

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

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

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

# Una tipica architettura a strati



## **Strato del dominio**

---

## Strato del dominio, strato della logica applicativa e modello di dominio

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

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

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

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

Un oggetto software di questo tipo è chiamato un **oggetto di dominio**, rappresenta una cosa nello spazio di dominio del problema e ha una logica applicativa o di business correlata (es. un oggetto *Sale* è in grado di calcolare il suo totale).

## Strato del dominio, strato della logica applicativa e modello di dominio

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

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

Un oggetto software di questo tipo è chiamato un **oggetto di dominio**, rappresenta una cosa nello spazio di dominio del problema e ha una logica applicativa o di business correlata (es. un oggetto *Sale* è in grado di calcolare il suo totale).

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

# Strato del dominio, strato della logica applicativa e modello di dominio

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

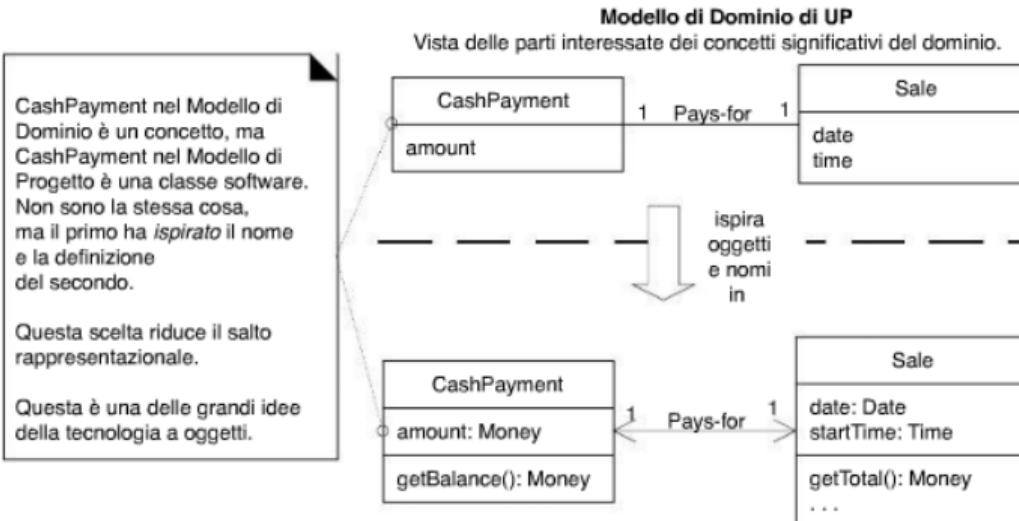
L'approccio consigliato consiste nel creare degli oggetti software con nomi e informazioni simili al dominio del mondo reale, e assegnare a essi responsabilità della logica applicativa.

Un oggetto software di questo tipo è chiamato un **oggetto di dominio**, rappresenta una cosa nello spazio di dominio del problema e ha una logica applicativa o di business correlata (es. un oggetto *Sale* è in grado di calcolare il suo totale).

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

Creando uno strato del domino ispirandosi al modello di dominio, si ottiene un “*salto rappresentazionale basso*” tra dominio del mondo reale e il progetto software.

# Strato del dominio e modello di dominio



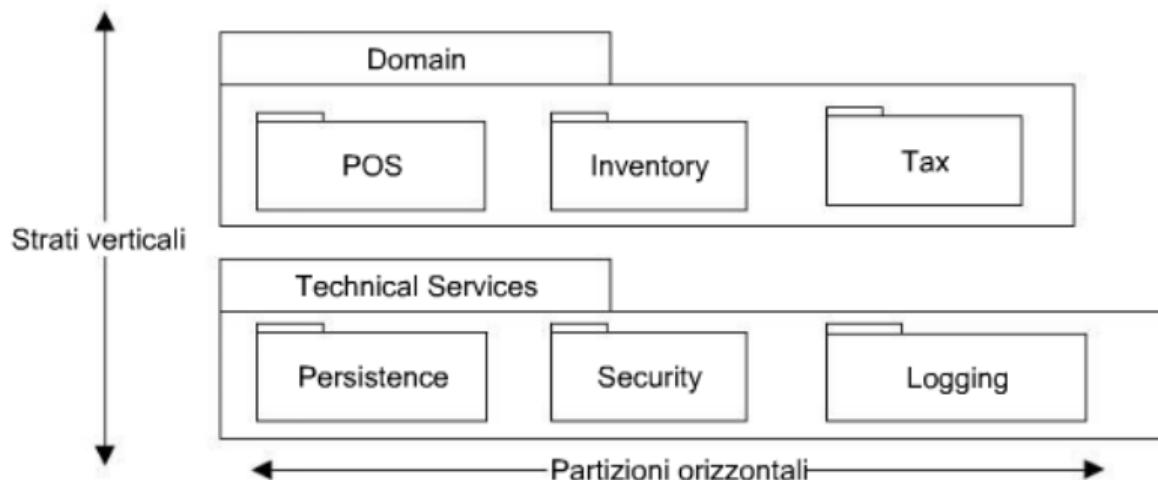
**Modello di Progetto di UP**  
Lo sviluppatore orientato agli oggetti ha tratto ispirazione dal dominio del mondo reale per la creazione di classi software.

Pertanto, il salto rappresentazionale tra il modo in cui le parti interessate concepiscono il dominio e la sua rappresentazione nel software è stato tenuto basso.

# Livelli, strati e partizioni

La nozione originaria di livello è di strato logico.

Si dice che gli **strati** di un'architettura rappresentano le *sezioni verticali*, mentre le **partizioni** rappresentano una *divisione orizzontale* di sottosistemi relativamente paralleli di uno strato.



# **Principio di separazione Modello-Vista**

---

# Principio di separazione Modello-Vista

## Principio di separazione Modello-Vista

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

## Principio di separazione Modello-Vista

---

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

# Principio di separazione Modello-Vista

**Modello = strato di dominio, Vista = strato UI**

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

# Principio di separazione Modello-Vista

## Modello = strato di dominio, Vista = strato UI

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

## Principio di separazione Modello-Vista

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

È un principio fondamentale del pattern **Model-View-Controller (MVC)**. MVC era un pattern di *Smalltalk-80* relativo a oggetti dati (*modelli*), elementi della GUI (*vista*) e gestori degli eventi del mouse e della tastiera (*controller*).

## Principio di separazione Modello-Vista

---

- Le classi di dominio **incapsulano** le informazioni e il comportamento relativi alla logica applicativa
- Le classi della vista sono relativamente leggere, esse sono **responsabili** dell'input e dell'output e di catturare gli eventi della GUI ma **non mantengono** i dati dell'applicazione né forniscono direttamente la logica applicativa

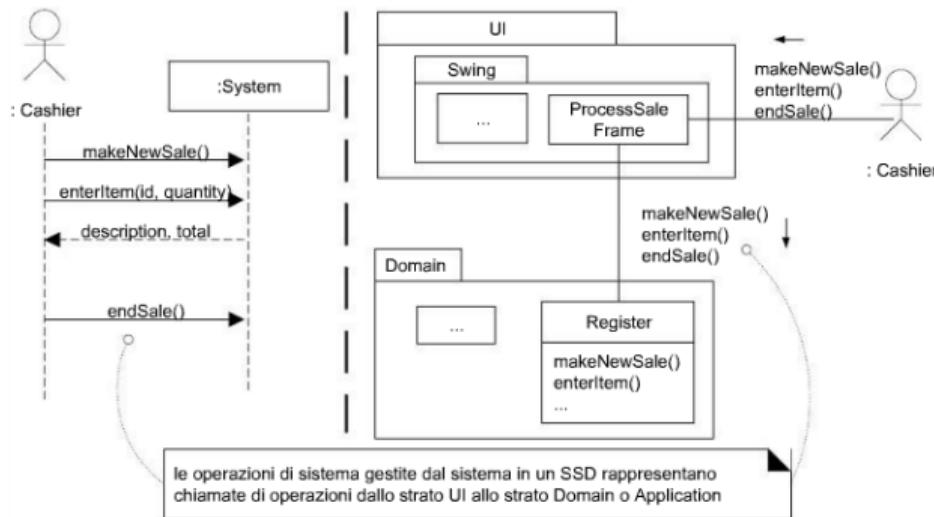
## Vantaggi del principio di separazione Modello-Vista

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

# SSD, operazioni di sistema e strati

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

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



# **09 . Verso la progettazione ad oggetti: modellazione dinamica e statica con UML**

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

### **Si noti che**

questi lucidi sono basati sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016” e sul materiale fornito dai docenti Viviana Bono, Claudia Picardi e Gianluca Torta dell’Università degli Studi di Torino che hanno tenuto il corso negli anni accademici precedenti.

# Table of contents

1. Dai requisiti alla progettazione
2. Verso la progettazione ad oggetti
3. Diagrammi di interazione
4. Diagrammi delle classi

## Dai requisiti alla progettazione

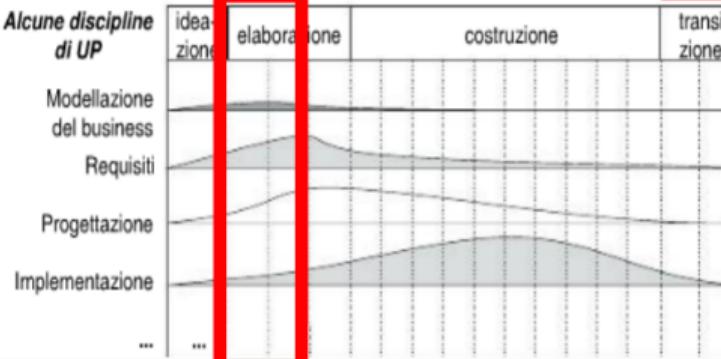
---

# UP maps

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

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

## Alcune discipline di UP



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

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

# Dove siamo?

- Il 10% dei requisiti è stato esaminato durante l'ideazione
- Un'indagine un poco più approfondita è stata iniziata nella prima iterazione dell'elaborazione

## Progettazione

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

# Dove siamo?

- Il 10% dei requisiti è stato esaminato durante l'ideazione
- Un'indagine un poco più approfondita è stata iniziata nella prima iterazione dell'elaborazione

## Progettazione

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

Nota: è naturale scoprire e modificare alcuni requisiti durante il lavoro di progettazione e di implementazione, soprattutto nelle iterazioni iniziali. La programmazione dall'inizio, i test e le demo aiutano a provocare presto questi cambiamenti inevitabili. È lo scopo dello sviluppo iterativo.

## **Verso la progettazione ad oggetti**

---

# Progettare a oggetti

Come progettare a oggetti?

- **Codifica.** Progettare mentre si codifica.
- **Disegno, poi codifica.** Disegnare alcuni diagrammi UML, poi passare alla codifica.
- **Solo disegno.** Lo strumento genera ogni cosa dai diagrammi.

# Progettare a oggetti

Come progettare a oggetti?

- **Codifica.** Progettare mentre si codifica.
- **Disegno, poi codifica.** Disegnare alcuni diagrammi UML, poi passare alla codifica.
- **Solo disegno.** Lo strumento genera ogni cosa dai diagrammi.

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

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

- Modellare insieme agli altri, modellazione in gruppo
- Creare diversi modelli in parallelo (sia *dinamici* che *statici*)

# Modellazione statica e dinamica

Ci sono due tipi di modelli per gli oggetti:

- **dinamici**
- **statici**

# Modellazione statica e dinamica

Ci sono due tipi di modelli per gli oggetti:

- **dinamici**
- **statici**

## Modelli dinamici (es., diagrammi di interazione UML)

Rappresentano il comportamento del sistema, la collaborazione tra oggetti software per realizzare (uno/più scenari di) un caso d'uso, i metodi di classi software.

La modellazione a oggetti dinamica più comune è quella con i diagrammi di sequenza di UML.

# Modellazione statica e dinamica

Ci sono due tipi di modelli per gli oggetti:

- **dinamici**
- **statici**

## Modelli dinamici (es., diagrammi di interazione UML)

Rappresentano il comportamento del sistema, la collaborazione tra oggetti software per realizzare (uno/più scenari di) un caso d'uso, i metodi di classi software.

La modellazione a oggetti dinamica più comune è quella con i diagrammi di sequenza di UML.

## Modelli statici (es., diagrammi di classe UML)

Servono per definire i package, i nomi delle classi, gli attributi, le firme di operazioni.

La modellazione a oggetti statica più comune è quella con i diagrammi delle classi di UML.

# Modellazione statica e dinamica

Ci sono due tipi di modelli per gli oggetti:

- **dinamici**
- **statici**

## Modelli dinamici (es., diagrammi di interazione UML)

Rappresentano il comportamento del sistema, la collaborazione tra oggetti software per realizzare (uno/più scenari di) un caso d'uso, i metodi di classi software.

La modellazione a oggetti dinamica più comune è quella con i diagrammi di sequenza di UML.

## Modelli statici (es., diagrammi di classe UML)

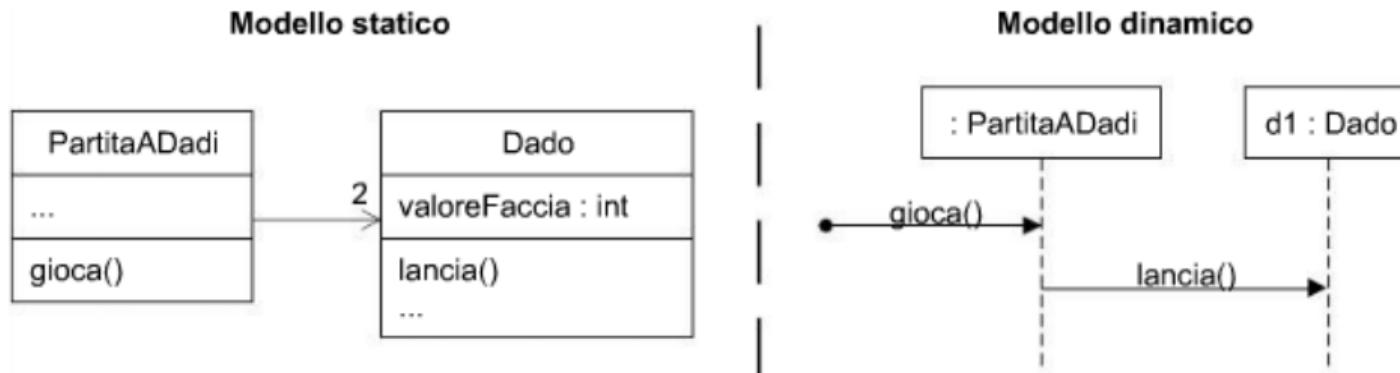
Servono per definire i package, i nomi delle classi, gli attributi, le firme di operazioni.

La modellazione a oggetti statica più comune è quella con i diagrammi delle classi di UML.

*Sono tra loro relazionati ed è per questa ragione che si consiglia di crearli in parallelo.*

# Modellazione statica e dinamica

- I messaggi nel diagramma di sequenza indicano operazioni nelle classi che ricevono il messaggio del diagramma delle classi.
- Le linee di vita nel diagramma di sequenza rappresentano oggetti di classi del diagramma delle classi



Un diagramma delle classi di UML.

Un diagramma di sequenza di UML.

## Modellazione statica e dinamica

---

- La maggior parte del lavoro di progettazione difficile, interessante e utile avviene mentre si disegnano i diagrammi di interazione, che rappresentano una vista dinamica
- Durante la modellazione a oggetti dinamica si pensa in modo dettagliato e preciso a quali oggetti devono esistere e come questi collaborano attraverso messaggi e metodi
- Durante la modellazione dinamica si applica la progettazione guidata dalle **responsabilità** e i principi **GRASP**

# Progettazione a oggetti e UML

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

# Progettazione a oggetti e UML

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

La progettazione a oggetti richiede soprattutto la conoscenza di:

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

## Diagrammi di interazione

---

# Diagrammi di interazione e modellazione dinamica

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

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

## Interazione

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

Il termine diagramma di interazione è una generalizzazione dei tipi più specifici di diagrammi UML di **sequenza** e di **comunicazione**. Noi ci concentreremo su quelli di sequenza.

# Interazione

---



Un'interazione è motivata dalla necessità di eseguire un determinato **compito**.

Un' **interazione** è motivata dalla necessità di eseguire un determinato **compito**.

Un **compito** è rappresentato da un **messaggio** che dà inizio all'interazione (messaggio trovato).

Un' **interazione** è motivata dalla necessità di eseguire un determinato **compito**.

Un **compito** è rappresentato da un **messaggio** che dà inizio all'interazione (messaggio trovato).

Il **messaggio** è inviato a un oggetto designato come **responsabile** per questo compito.

Un'interazione è motivata dalla necessità di eseguire un determinato **compito**.

Un **compito** è rappresentato da un **messaggio** che dà inizio all'interazione (messaggio trovato).

Il **messaggio** è inviato a un oggetto designato come **responsabile** per questo compito.

L'**oggetto responsabile** collabora/interagisce con altri oggetti (**partecipanti**) per svolgere il compito.

Un'interazione è motivata dalla necessità di eseguire un determinato **compito**.

Un **compito** è rappresentato da un **messaggio** che dà inizio all'interazione (messaggio trovato).

Il **messaggio** è inviato a un oggetto designato come **responsabile** per questo compito.

L'**oggetto responsabile** collabora/interagisce con altri oggetti (**partecipanti**) per svolgere il compito.

Ciascun **partecipante** svolge un proprio **ruolo** nell'ambito della **collaborazione**.

Un'interazione è motivata dalla necessità di eseguire un determinato **compito**.

Un **compito** è rappresentato da un **messaggio** che dà inizio all'interazione (messaggio trovato).

Il **messaggio** è inviato a un oggetto designato come **responsabile** per questo compito.

L'**oggetto responsabile** collabora/interagisce con altri oggetti (**partecipanti**) per svolgere il compito.

Ciascun **partecipante** svolge un proprio **ruolo** nell'ambito della **collaborazione**.

La **collaborazione** avviene mediante **scambio di messaggi (interazione)**.

Un'interazione è motivata dalla necessità di eseguire un determinato **compito**.

Un **compito** è rappresentato da un **messaggio** che dà inizio all'interazione (messaggio trovato).

Il **messaggio** è inviato a un oggetto designato come **responsabile** per questo compito.

L'**oggetto responsabile** collabora/interagisce con altri oggetti (**partecipanti**) per svolgere il compito.

Ciascun **partecipante** svolge un proprio **ruolo** nell'ambito della **collaborazione**.

La **collaborazione** avviene mediante **scambio di messaggi (interazione)**.

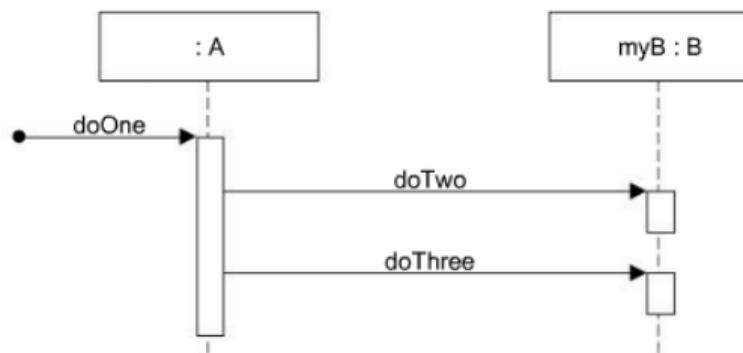
Ciascun **messaggio** è una richiesta che un oggetto fa a un altro oggetto di eseguire un'**operazione**.

# Diagrammi di sequenza

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

Vantaggi: mostrano chiaramente la sequenza dell'ordinamento temporale dei messaggi.

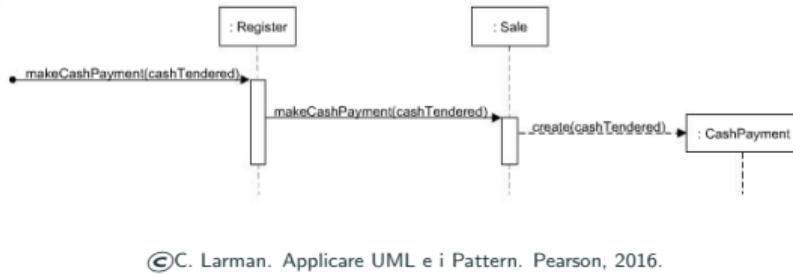
Svantaggi: costringono a estendersi verso destra quando si aggiungono nuovi oggetti.



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

# Diagrammi di sequenza

- Il messaggio *makeCashPayment* viene inviato a un'istanza di *Register*. Il mittente non è identificato.
- L'istanza di *Register* invia il messaggio *makeCashPayment* a un'istanza di *Sale*
- L'istanza di *Sale* crea un'istanza di *CashPayment*



```
public class Sale {  
    private CashPayment payment;  
  
    public void makeCashPayment( Money cashTendered ) {  
        payment = new CashPayment( cashTendered );  
        ...  
    }  
    ...  
}
```

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

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

# Diagrammi di sequenza in UP

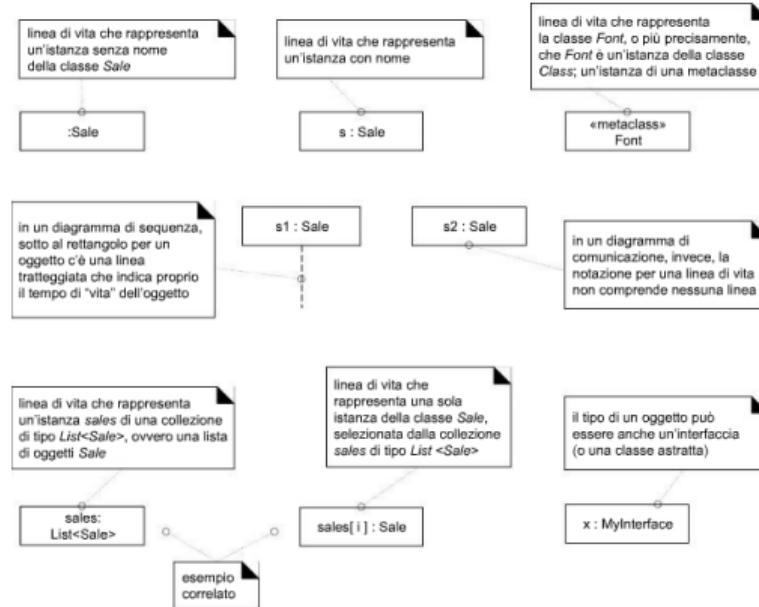
## Design Sequence Diagram (DSD)

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

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

# Diagrammi di sequenza: partecipanti

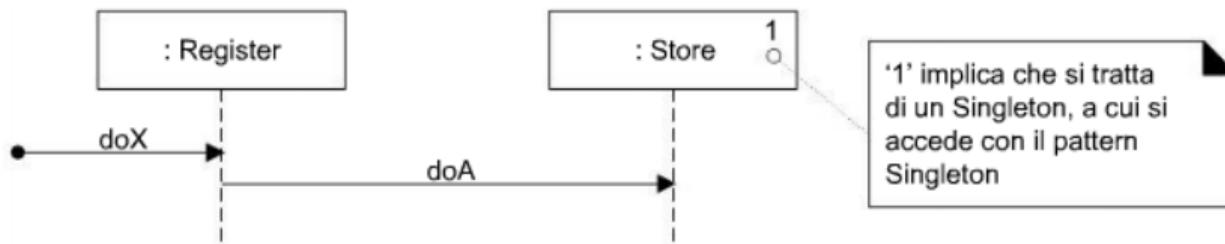
I rettangoli sono chiamati **linee di vita** (*lifeline*). Rappresentano i **partecipanti** all'interazione, ovvero le parti correlate definite nel contesto di un qualche diagramma strutturale.



# Diagrammi di sequenza: singleton

*Singleton*: pattern nel quale da una classe viene istanziata una sola istanza, mai due o più.

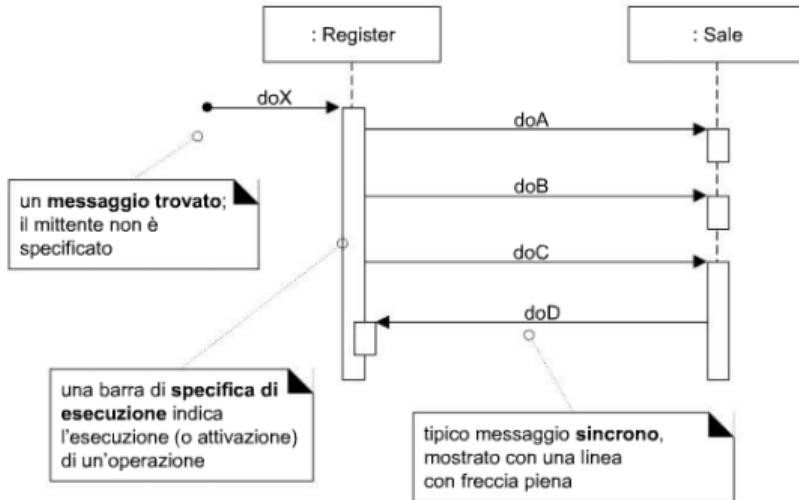
In un diagramma di interazione, un tale oggetto “singleton” è contrassegnato da un ‘1’ nell'angolo superiore destro della linea di vita.



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

# Diagrammi di sequenza: linee di vita e messaggi

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

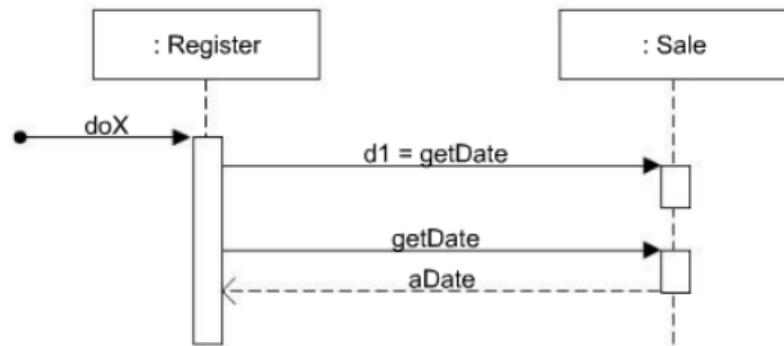


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Diagrammi di sequenza: risposte o ritorni

Ci sono due modi per mostrare il risultato di ritorno:

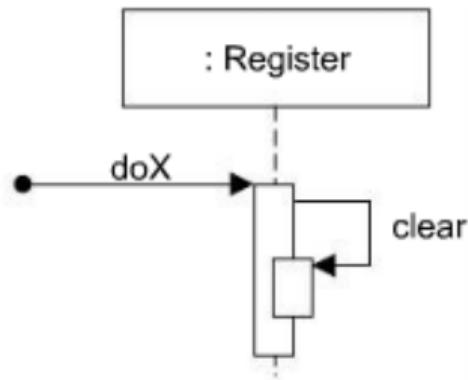
- utilizzando la sintassi `returnVar = message(parametri)`
- utilizzando una linea di messaggio di risposta (o ritorno) alla fine della barra di specifica di esecuzione



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

## Diagrammi di sequenza: self o this

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



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

# Diagrammi di sequenza: creazione di istanze

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

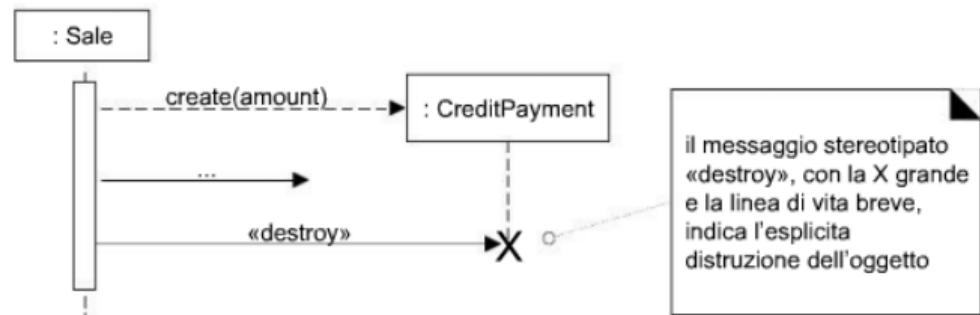
Si noti che i nuovi oggetti creati sono posizionati all' "altezza" della loro creazione.



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

# Diagrammi di sequenza: distruzione di oggetti

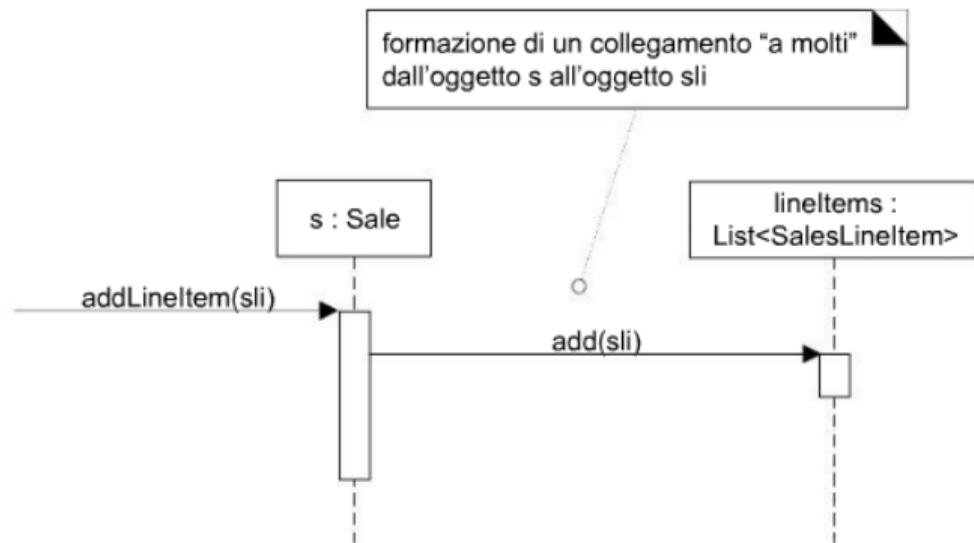
Una distruzione esplicita di un oggetto.



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

# Diagrammi di sequenza: formazione di collegamenti

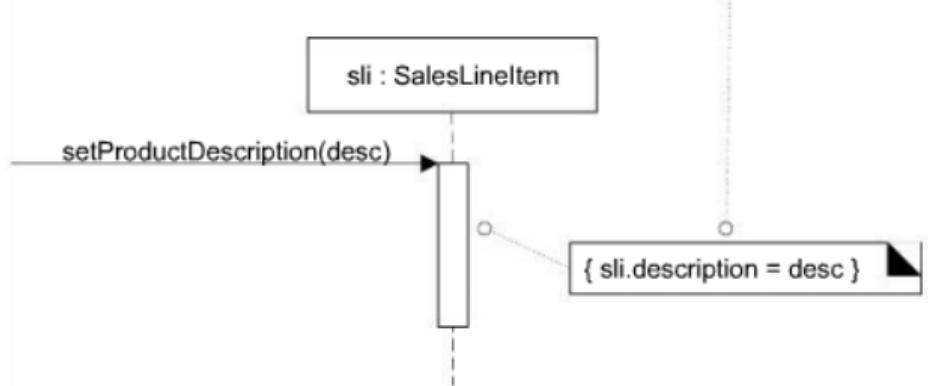
Formazione di un collegamento di un'associazione “*a molti*”.



# Diagrammi di sequenza: formazione di collegamenti

Formazione di un collegamento di un'associazione “*a uno*”.  
Il vincolo va interpretato come una post-condizione dell'operazione, ovvero una condizione che deve risultare vera al termine della sua esecuzione.

vincolo che indica la formazione di un collegamento “*a uno*” dall'oggetto sli all'oggetto desc mediante l'attributo description

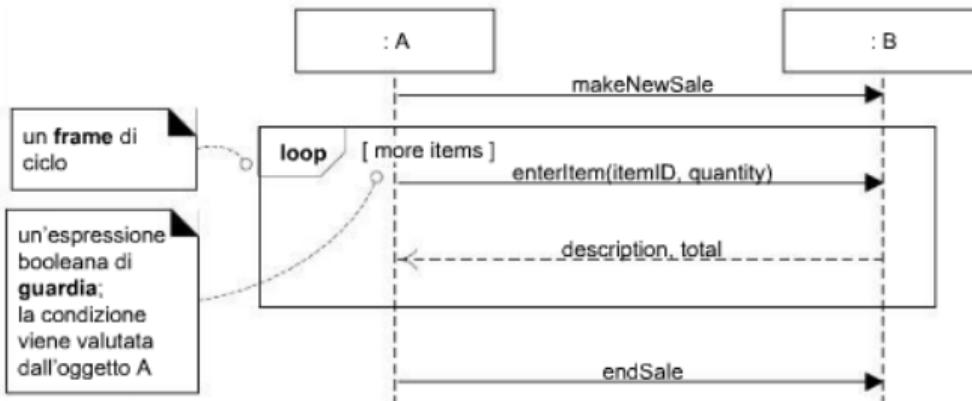


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Diagrammi di sequenza: frame

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

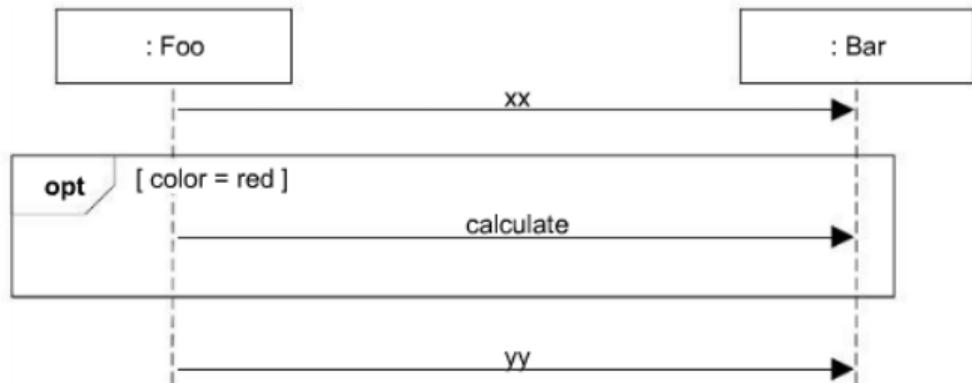
I frame sono regioni o frammenti  
dei diagrammi, hanno un  
operatore (etichetta) e una  
guardia (condizione o test  
booleano) che va posizionata  
sopra la linea di vita che deve  
valutare tale condizione.



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

# Diagrammi di sequenza: messaggi condizionali

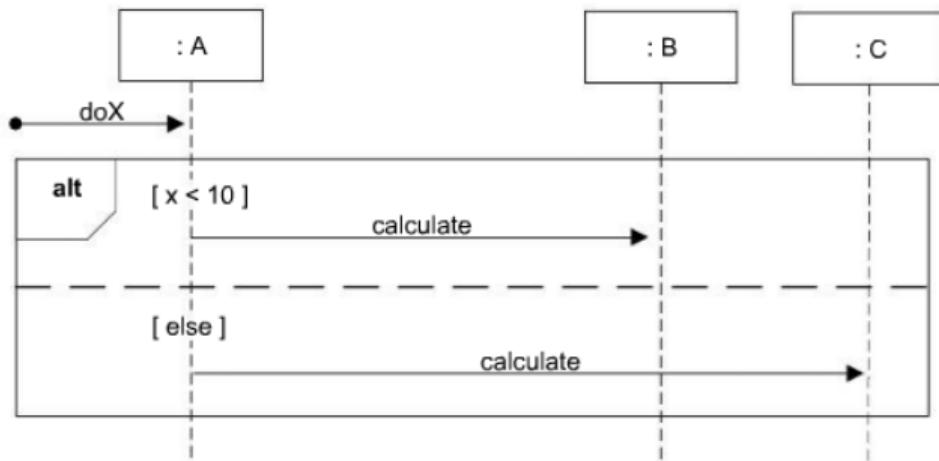
Un frame **OPT** è posizionato attorno a uno o più messaggi.



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

# Diagrammi di sequenza: messaggi condizionali mutuamente esclusivi

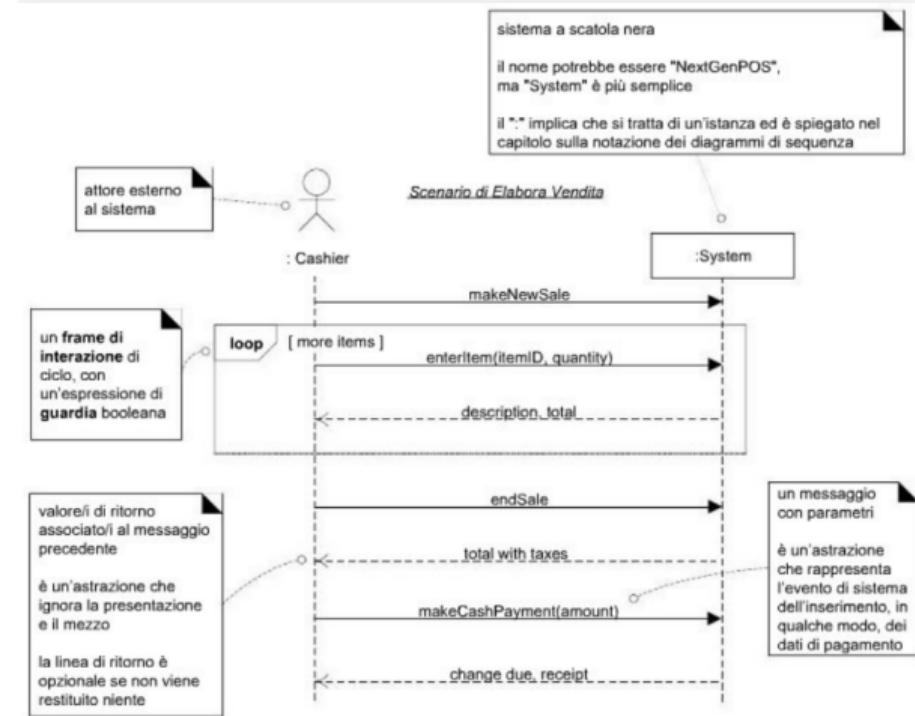
Un frame **ALT** è posizionato attorno alle alternative mutuamente esclusive.



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

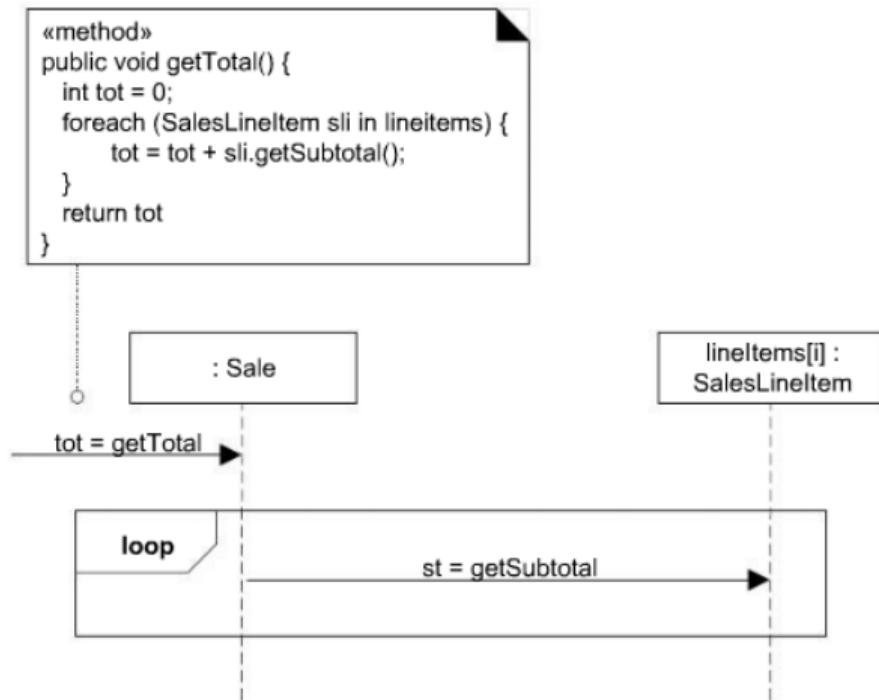
# Diagrammi di sequenza: iterazione

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



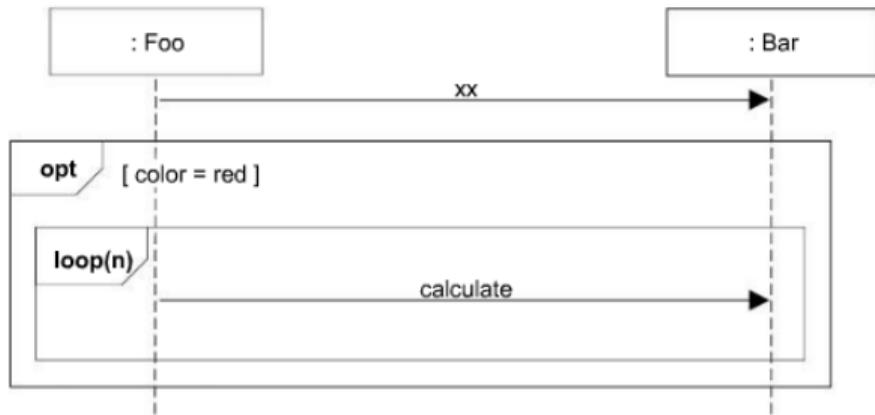
# Diagrammi di sequenza: iterazione su una collezione

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



# Diagrammi di sequenza: annidamento di frame

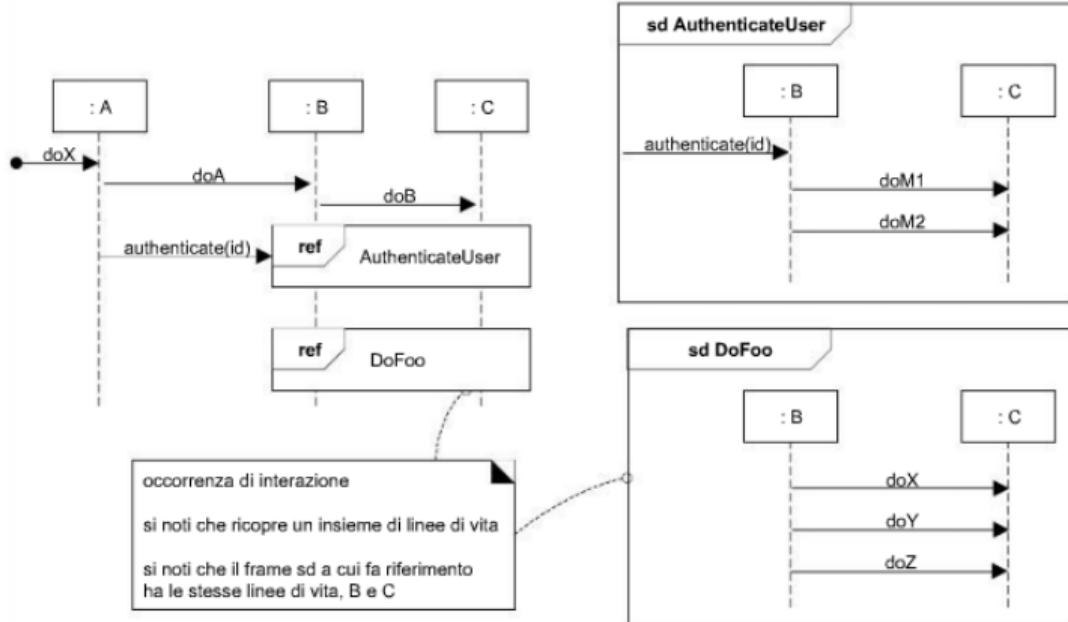
I frame possono essere annidati.



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

# Diagrammi di sequenza: correlare diagrammi di interazione

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



# Diagrammi di sequenza: invocare metodi statici

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

messaggio alla classe,  
o chiamata di metodo statico



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

# Diagrammi di sequenza: chiamate sincrone e asincrone

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

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

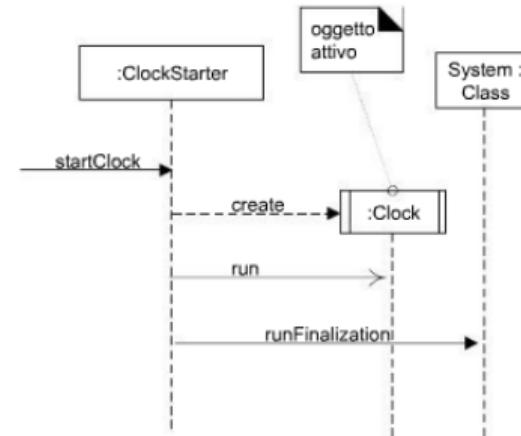
una freccia non piena indica una **chiamata asincrona** (diversamente da una freccia piena, che invece rappresenta una più comune chiamata sincrona)

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*



## Diagrammi delle classi

---

# Diagrammi delle classi e modellazione statica

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

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

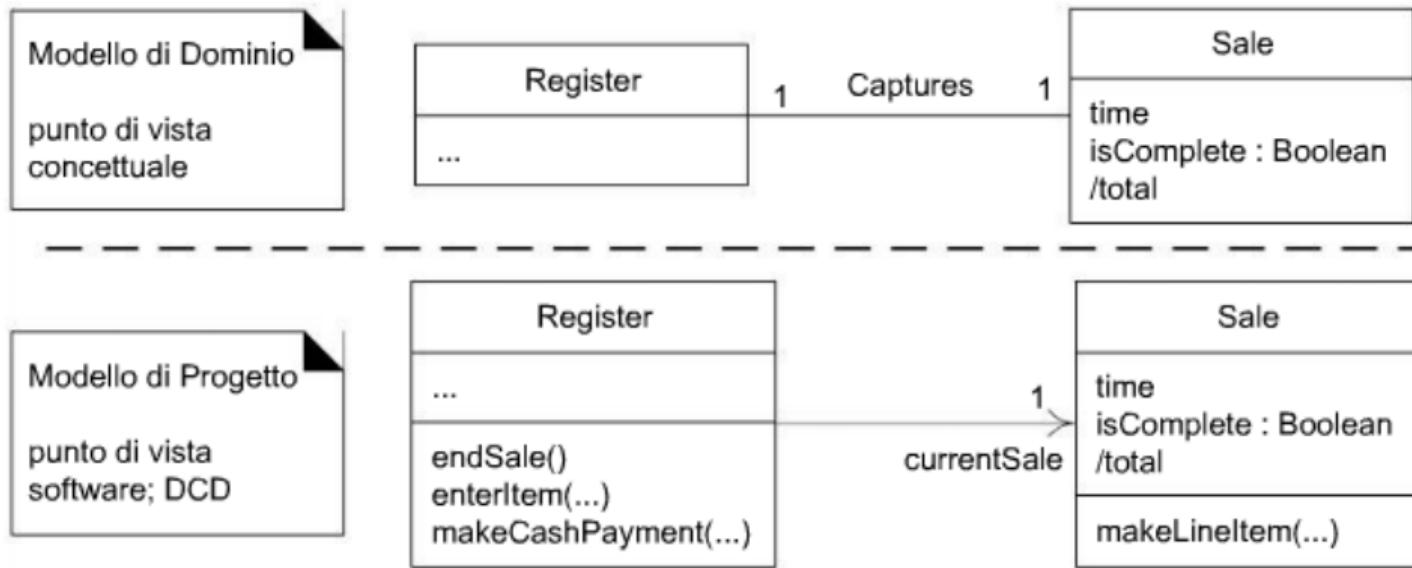
I diagrammi delle classi sono stati utilizzati, da un punto di vista concettuale, per visualizzare un modello di dominio.

## Design Class Diagram (DCD)

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

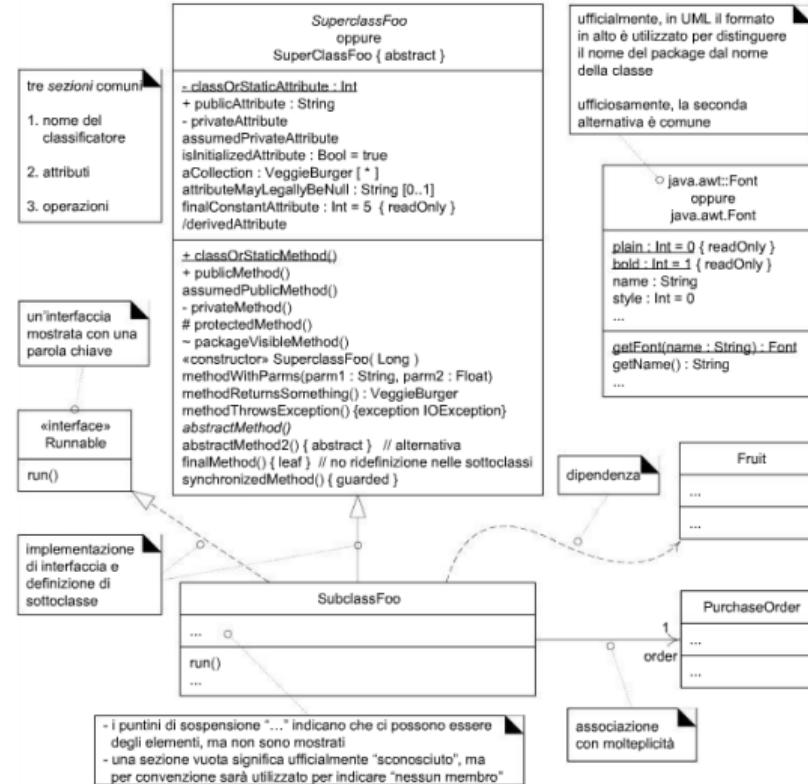
In UP, l'insieme di tutti i DCD fa parte del **Modello di Progetto** che comprende anche i diagrammi di interazione.

# Diagrammi delle classi di UML dai due punti di vista



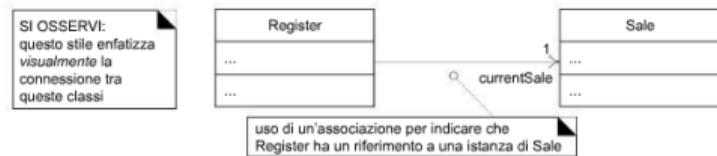
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Notazione comune dei diagrammi delle classi di UML



# Diagrammi delle classi: notazioni per attributi come associazioni

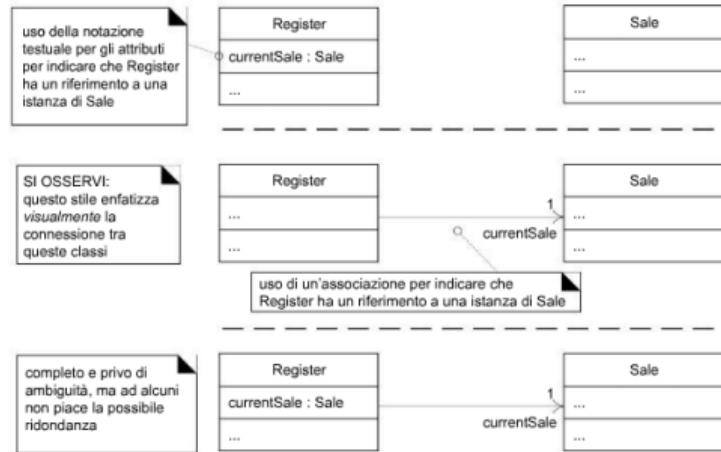
- Notazione testuale per un attributo (in alto)
- Notazione con una linea di associazione (in mezzo)
- Entrambe le notazioni, insieme (in basso): *non consigliata!*



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

# Diagrammi delle classi: notazioni per attributi come associazioni

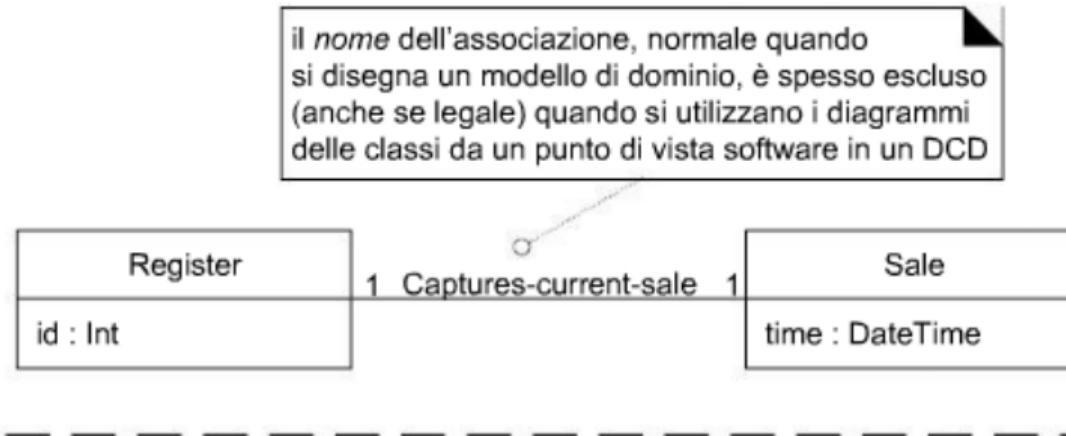
- Se non viene indicata alcuna visibilità, solitamente si ipotizza che gli attributi siano privati
- Una freccia di navigabilità rivolta dalla classe sorgente alla classe destinazione dell'associazione, che indica che un oggetto della classe sorgenti ha un attributo di tipo della classe destinazione
- Una molteplicità all'estremità vicina alla destinazione, ma non all'estremità vicina alla sorgente
- Un nome di ruolo solo all'estremità vicina alla destinazione, per indicare il nome dell'attributo
- Nessun nome per l'associazione



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

# Diagrammi delle classi: modello di dominio VS DCD

Punto di vista  
concettuale  
del Modello  
di Dominio di UP

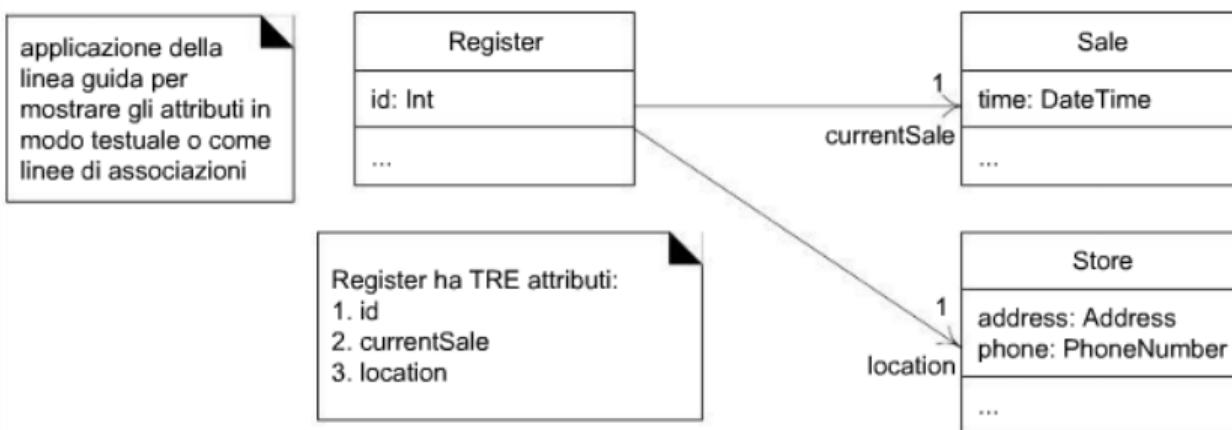


Punto di vista  
software del  
DCD del Modello  
di Progetto di UP



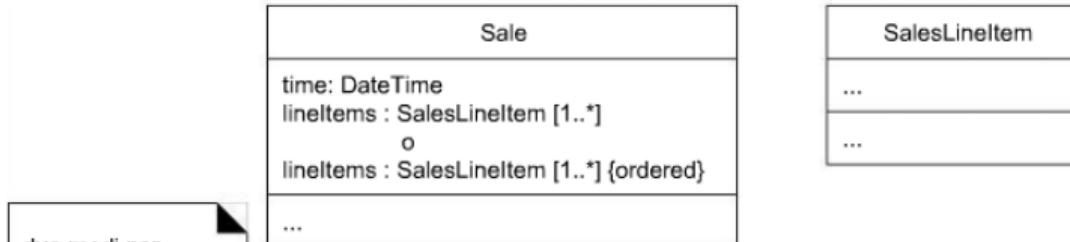
# Diagrammi delle classi: associazioni e attributi

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

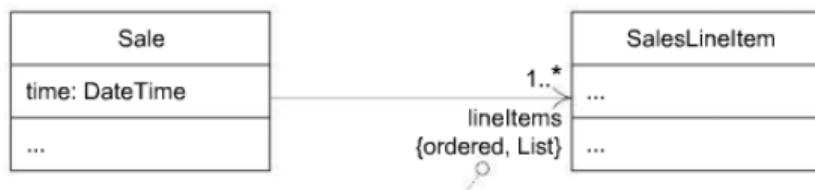


# Diagrammi delle classi: attributi collezione e note

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



due modi per  
mostrare un attributo  
collezione



si noti che un'estremità di associazione può anche  
avere una stringa di proprietà come {ordered, List}

# Diagrammi delle classi: operazioni e metodi

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

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

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

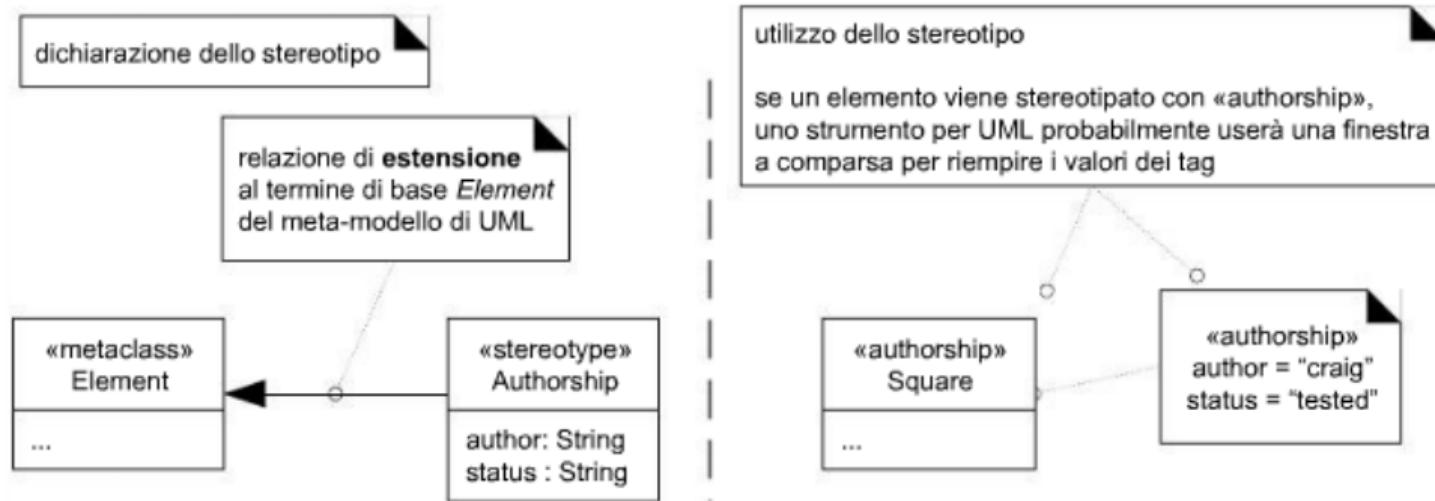
## Diagrammi delle classi: parole chiave

- Decoratori testuali per classificare un elemento di un modello

<b>Parola chiave</b>	<b>Significato</b>	<b>Esempio di uso</b>
«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

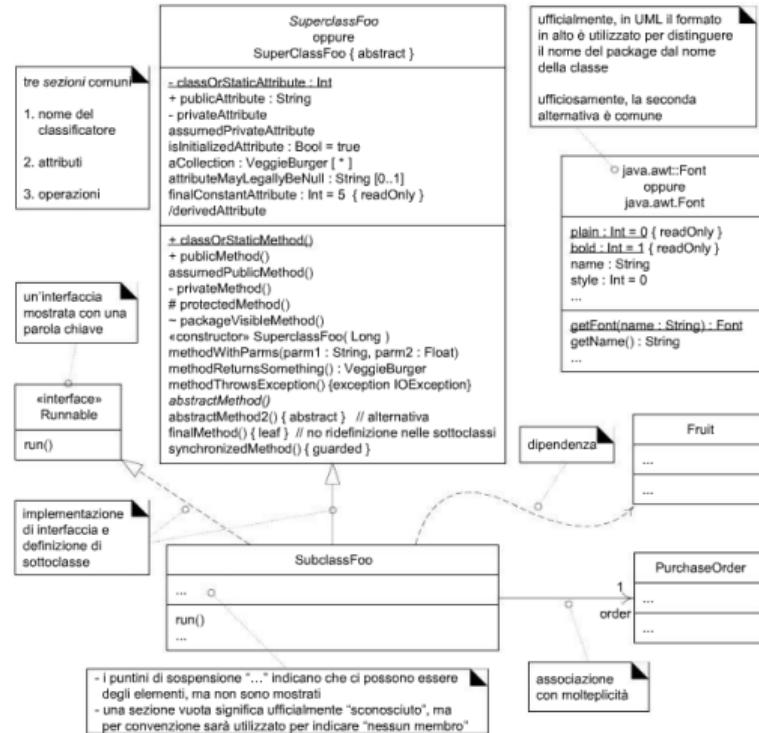
# Diagrammi delle classi: stereotipi

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



# Diagrammi delle classi: generalizzazione

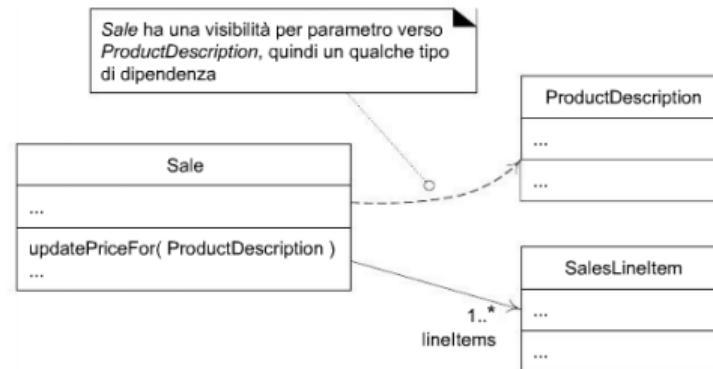
- È una **relazione tassonomica** tra un classificatore più generale e un classificatore più specifico. Ogni istanza del classificatore più specifico è anche un'istanza indiretta del classificatore più generale. Pertanto il classificatore più specifico possiede indirettamente le caratteristiche del classificatore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



# Diagrammi delle classi: dipendenze

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

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

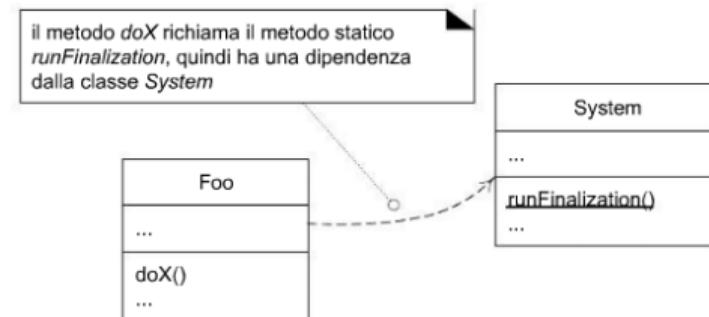


# Diagrammi delle classi: dipendenze

Esistono molte tipologie di dipendenze:

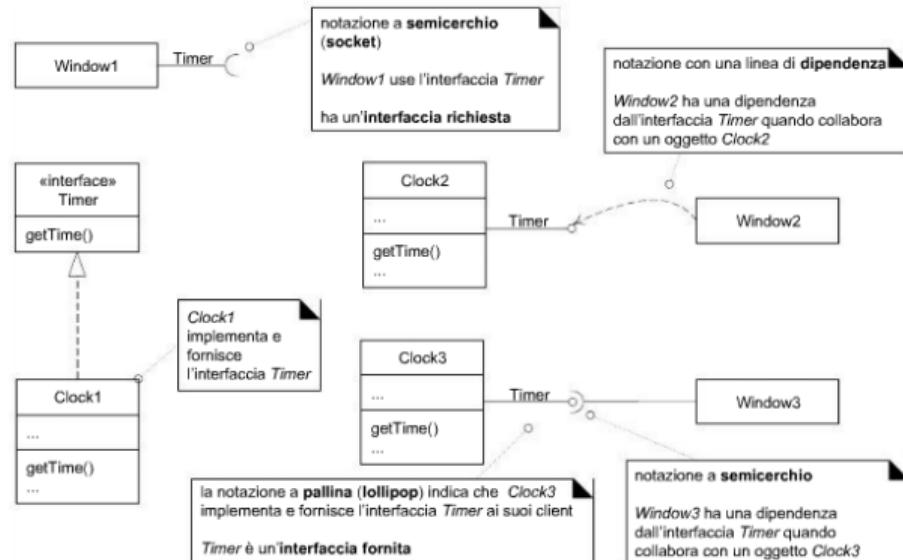
- Avere un attributo del tipo del fornitore
- Inviare un messaggio a un fornitore; la visibilità verso il fornitore potrebbe essere data da:
  - un attributo, una variabile parametro, una variabile locale, una variabile globale, o una visibilità di classe (chiamata di metodi statici o di classe)
- Ricevere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

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



# Diagrammi delle classi: interfacce

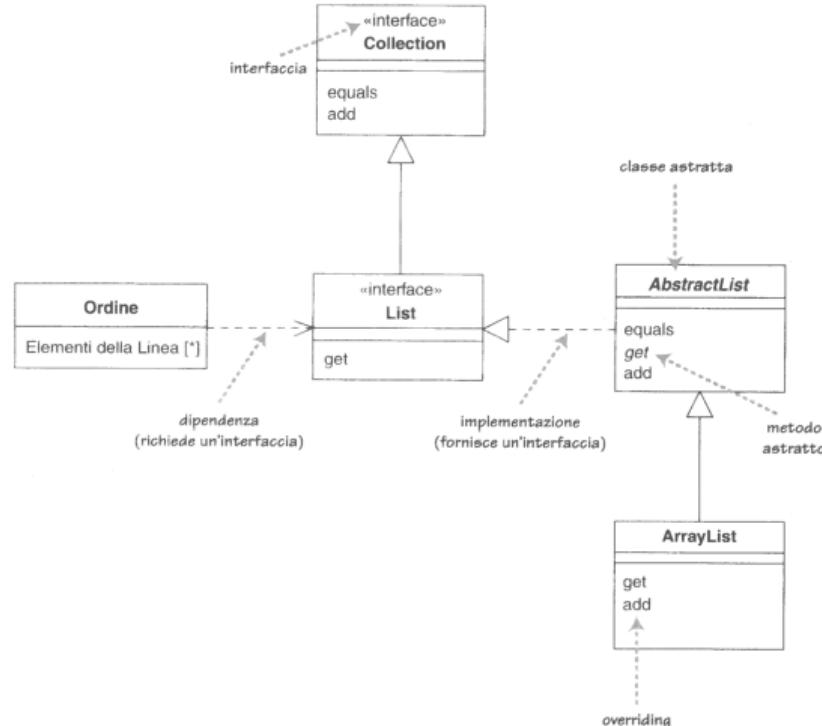
- L'implementazione di un'interfaccia viene chiamata una **realizzazione di interfaccia**
- La *notazione a pallina (lollipop)* indica che una classe X implementa (fornisce) un'interfaccia Y, senza disegnare il rettangolo per l'interfaccia Y
- La *notazione a semicerchio (socket)* (socket) indica che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



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

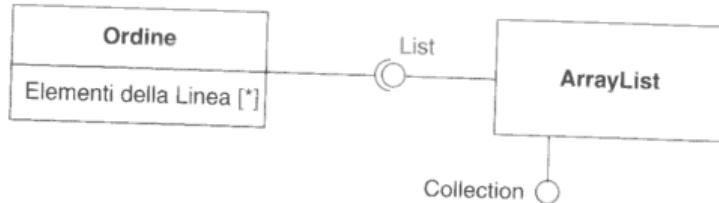
# Diagrammi delle classi: interfacce

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

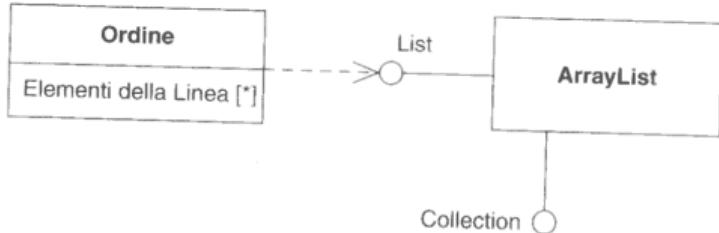


# Diagrammi delle classi: interfacce

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



©M. Fowler. UML Distilled. Pearson, 2018.



©M. Fowler. UML Distilled. Pearson, 2018.

# Diagrammi delle classi: composizione

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

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



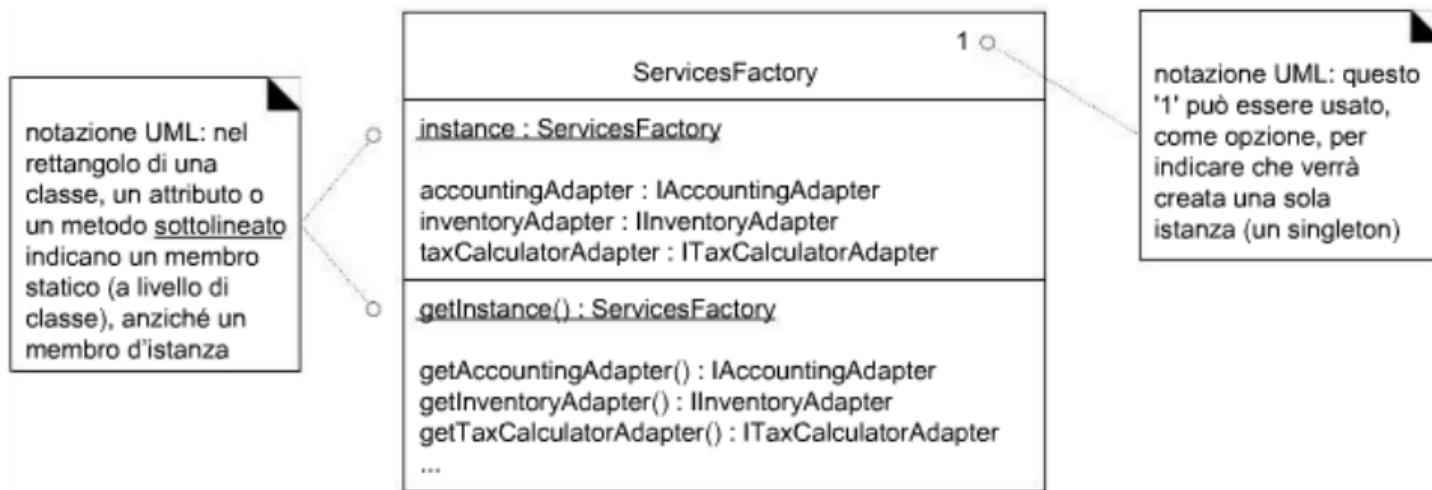
composition



la composizione indica che:  
- un'istanza della parte (*Square*) può far parte di un unico composto (*Board*) alla volta  
- il composto ha la responsabilità esclusiva della gestione delle proprie parti, soprattutto la creazione e l'eliminazione

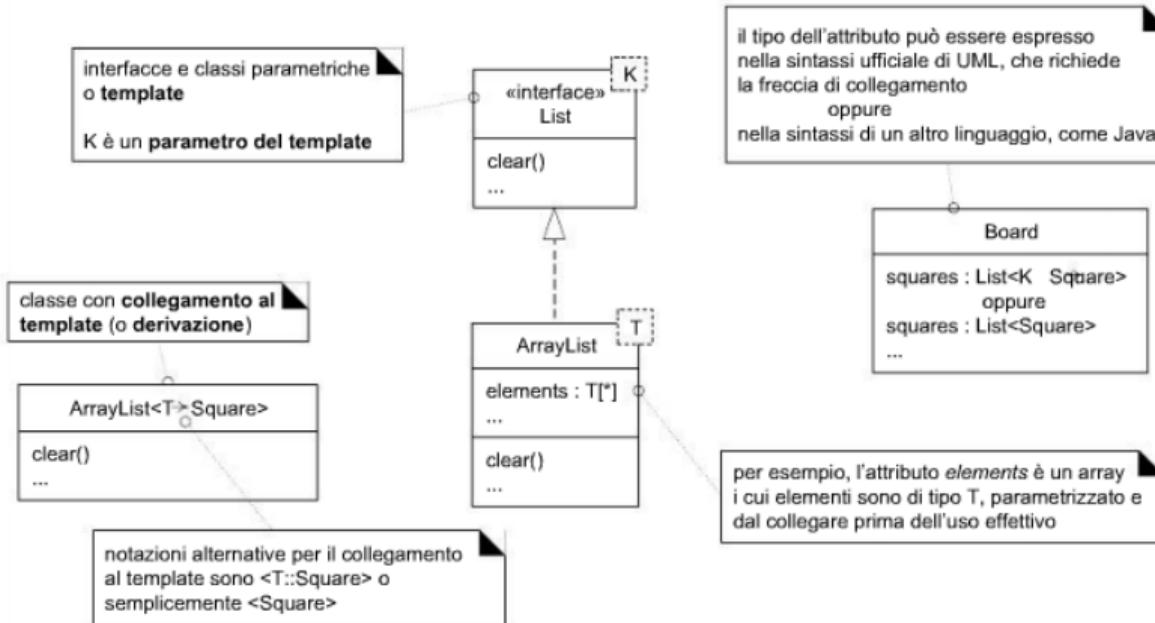
# Diagrammi delle classi: singleton

Esiste **una sola istanza** di una classe, mai due.



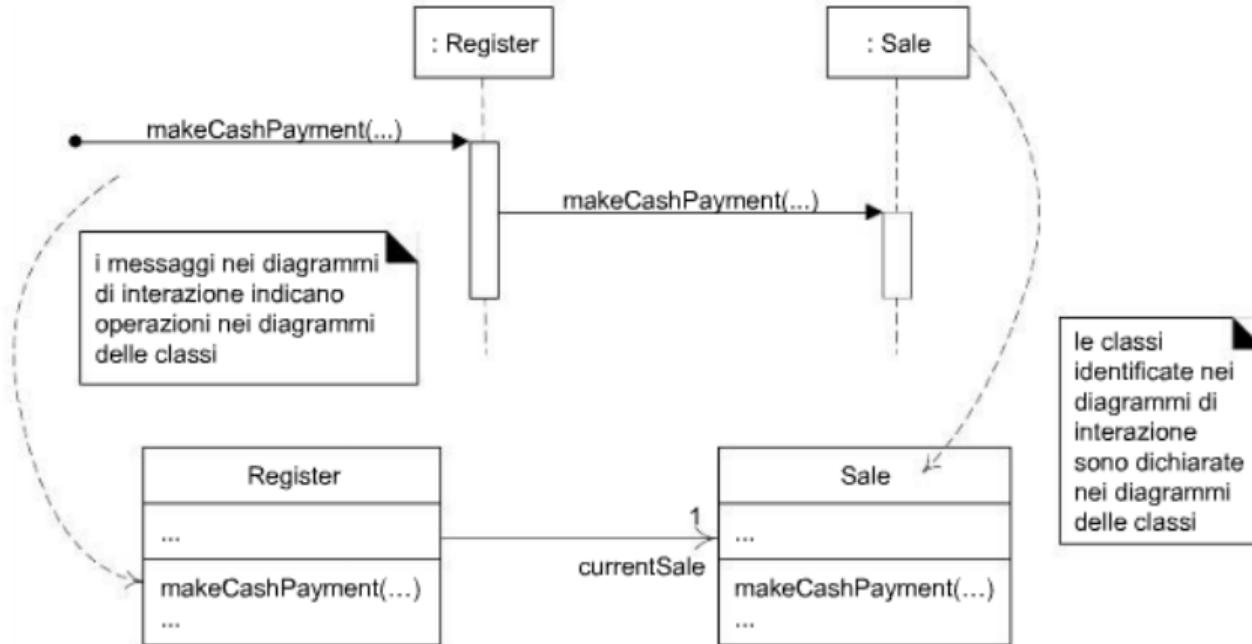
# Diagrammi delle classi: template

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



# Diagrammi delle classi e diagrammi di interazione/sequenza

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



# **10 . General Responsibility Assignment Software Patterns (GRASP)**

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

### **Si noti che**

questi lucidi sono basati sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016” e sul materiale fornito dai docenti Viviana Bono, Claudia Picardi e Gianluca Torta dell’Università degli Studi di Torino che hanno tenuto il corso negli anni accademici precedenti.

# Table of contents

1. Responsabilità
2. Responsibility-Driven Development
3. General Responsibility Assignment Software Pattern
4. I nove pattern GRASP

# **Responsabilità**

---

# Progettazione Object-Oriented (OODesign) e responsabilità

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

# Progettazione Object-Oriented (OODesign) e responsabilità

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

# Progettazione Object-Oriented (OODesign) e responsabilità

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

# Progettazione Object-Oriented (OODesign) e responsabilità

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

# Progettazione Object-Oriented (OODesign) e responsabilità

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

# Progettazione Object-Oriented (OODesign) e responsabilità

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

## Martin Fowler

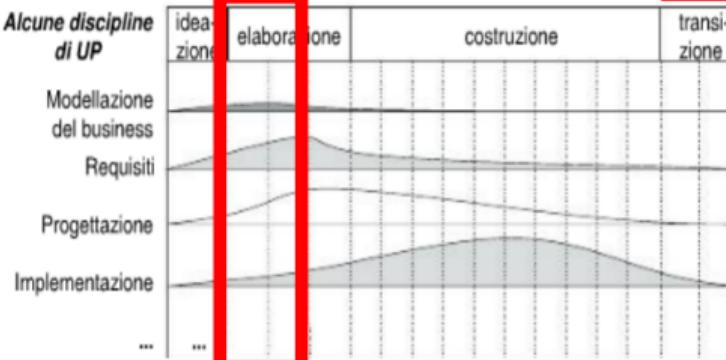
Capire le responsabilità è fondamentale per una buona programmazione ad oggetti.

# UP maps

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

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

## Alcune discipline di UP



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

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

# Dove siamo? Sempre alla prima iterazione...

Il primo **workshop dei requisiti della durata di due giorni** è finito.

Il chief architect e il responsabile dell'organizzazione concordano di implementare e testare alcuni **scenari del caso d'uso Elabora Vendita nella prima iterazione timeboxed di tre settimane**.

**Tre dei venti casi d'uso**, quelli che sono i più significativi dal punto di vista dell'architettura e di elevato valore di business, sono stati analizzati nel dettaglio, compreso naturalmente il caso d'uso *Elabora Vendita* (UP raccomanda la prassi tipica dei metodi iterativi, cioè analizzare solo il **10%-20% dei requisiti** in modo dettagliato prima di iniziare a programmare).

**Esperimenti di programmazione** hanno risolto le domande tecniche più eclatanti, per esempio se le UI Java Swing funzionano con i monitor touch screen.

**Altri elaborati** sono stati iniziati: Specifiche Supplementari, Glossario e Modello di Dominio.

Il chief architect ha disegnato **alcune idee per l'architettura logica su larga scala**, utilizzando i diagrammi dei package di UML. Ciò fa parte del Modello di Progetto di UP.

# Input della progettazione a oggetti

<p>Il <b>testo dei casi d'uso</b> definisce il comportamento visibile che gli oggetti software devono in definitiva supportare; gli oggetti vengono progettati per "realizzare" (implementare) i casi d'uso. In UP, questa progettazione OO è chiamata proprio <b>realizzazione dei casi d'uso</b>.</p>	<p>Le <b>Specifiche Supplementari</b> definiscono gli obiettivi non funzionali, che gli oggetti devono soddisfare, per esempio l'internazionalizzazione.</p>
<p>I <b>diagrammi di sequenza di sistema</b> identificano i messaggi per le operazioni di sistema, che sono i messaggi iniziali per i diagrammi di interazione di oggetti che collaborano da progettare.</p>	<p>Il <b>Glossario</b> chiarisce i dettagli dei parametri o dei dati che vengono dallo strato UI, i dati che vengono passati alla base di dati, e la logica specifica per ciascun elemento e i requisiti di validazione, come i formati legali e la validazione per i codici UPC dei prodotti.</p>
<p>I <b>contratti delle operazioni</b> possono essere complementari al testo dei casi d'uso per chiarire che cosa devono compiere gli oggetti software in un'operazione di sistema. Le post-condizioni definiscono in modo dettagliato i risultati da ottenere.</p>	<p>Il <b>Modello di dominio</b> suggerisce alcuni nomi e attributi degli oggetti software di dominio nello strato del dominio dell'architettura software.</p>

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

Non tutti questi elaborati sono necessari: in UP tutti gli elementi sono opzionali e vengono creati possibilmente per ridurre i rischi.

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

## Principi e pattern

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

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

# Progettazione Object-Oriented (OODesign) e principi/pattern

La cosa più importante è che, durante le attività di disegno e codifica, vengano applicati vari principi di progettazione OO, come i **pattern GRASP<sup>1</sup>** (**General Responsibility Assignment Software Patterns**, o **Schemi Generali per l'Assegnazione di Responsabilità nel Software**) e i **design pattern Gang-of-Four (GoF)**.

## Responsability-Driven Development (RDD)

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

---

<sup>1</sup>Nota, si dovrebbe dire Pattern GRAS ma si preferisce Pattern GRASP, in inglese *grasp* significa *afferrare*.

# Progettazione Object-Oriented (OODesign) e principi/pattern

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

## Siamo in UP!

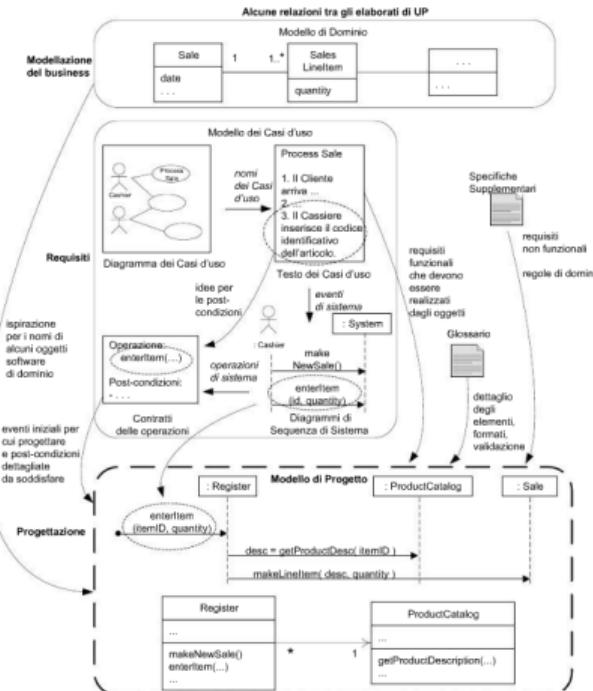
Presto si deve interrompere la modellazione e si deve passare a programmare, per evitare una mentalità a cascata che porterebbe a un'eccessiva modellazione prima della programmazione (iterazioni timeboxed).

# Output della progettazione a oggetti

Output: diagrammi di interazione e diagrammi delle classi UML.

In particolare, dopo la modellazione nella prima iterazione:

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



# Responsabilità e progettazione guidata dalle responsabilità

Un modo comune di pensare alla progettazione di oggetti software è in termini di **responsabilità, ruoli e collaborazioni: progettazione guidata dalle responsabilità o RDD** (*Responsibility-Driven Development*).

## RDD

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

# **Responsibility-Driven Development**

---

## Responsabilità e progettazione guidata dalle responsabilità

- In UML la responsabilità è un **contratto** o un **obbligo** di un classificatore
- Le responsabilità sono correlate agli obblighi o al **comportamento** di un oggetto in relazione al suo ruolo
- Le responsabilità sono fondamentalmente di due tipi:
  - **di fare**
  - **di conoscere**

## Responsabilità di fare

---



Le responsabilità **di fare** di un oggetto comprendono:

- fare qualcosa esso stesso, come per esempio creare un oggetto o eseguire un calcolo
- chiedere ad altri oggetti di eseguire azioni
- controllare e coordinare le attività di altri oggetti

# Responsabilità di conoscere



Le responsabilità **di conoscere** di un oggetto comprendono:

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

# Responsabilità e progettazione guidata dalle responsabilità

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

## Esempio:

Si può dichiarare che “una *Sale* è **responsabile** della creazione di oggetti *SaleLineItems*“ (una responsabilità di fare), o che “una *Sale* è **responsabile** di conoscere il suo totale“ (una responsabilità di conoscere).

La traduzione delle responsabilità in classi e metodi è influenzata dalla *granularità* delle responsabilità. Le responsabilità più grandi coinvolgono centinaia di classi e metodi, mentre le responsabilità minori possono coinvolgere un solo metodo.

# Responsabilità e collaborazione

Nel software non c'è niente che corrisponde direttamente a una responsabilità.

Tuttavia, nel software vengono definite classi, metodi e variabili con lo scopo di **soddisfare responsabilità**.

Le responsabilità sono implementate per mezzo degli oggetti e metodi che agiscono da soli oppure che **collaborano** con altri oggetti e metodi.

## Esempio:

La classe *Sale* potrebbe definire uno o più metodi per conoscere il tuo totale. Per soddisfare questa responsabilità, la *Sale* può **collaborare** con altri oggetti, per esempio inviando un messaggio *getSubtotal* a ciascun oggetto *SaleLineItem* per chiedere il rispettivo totale parziale.

## Responsabilità come metafora

La RDD (Responsibility Driven Development) è una **metafora** generale per pensare alla progettazione del software OO.

Si pensi agli oggetti software come simili a persone che hanno delle responsabilità e che collaborano con altre persone per svolgere un lavoro.

La RDD porta a considerare un progetto OO come una **comunità di oggetti con responsabilità che collaborano**.

La progettazione guidata dalle responsabilità viene fatta, iterativamente, come segue:

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

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

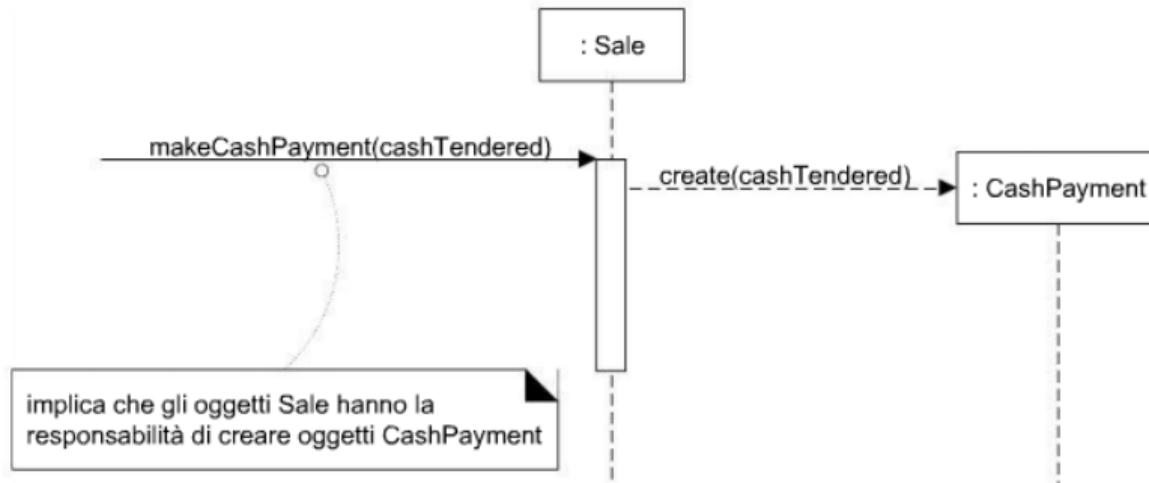
# **General Responsibility Assignment Software Pattern**

---

## GRASP come aiuto alla comprensione

- I pattern **GRASP**, per l'assegnamento di responsabilità, rappresentano solo un aiuto per apprendere la struttura e dare un nome ai principi
- In altre parole, i *principi o pattern GRASP* sono un aiuto per l'apprendimento degli aspetti essenziali della progettazione ad oggetti e per l'applicazione dei ragionamenti di progettazione in un modo metodico, razionale e spiegabile
- Uno strumento per aiutare ad acquisire la padronanza delle basi dell'OOD e a comprendere l'assegnazione di responsabilità nella progettazione a oggetti

- Le decisioni sull'assegnazione delle responsabilità agli oggetti possono essere prese **mentre si esegue la codifica oppure durante la modellazione**
- Il disegnare i diagrammi di interazione diventa l'occasione per considerare tali responsabilità (realizzate come metodi): **le responsabilità e i metodi sono correlati!**



# Cosa sono i pattern?

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

## Pattern

I principi e gli idiomi se codificati in un formato strutturato che descrive il problema e la soluzione e a cui è assegnato un nome, possono essere chiamati **pattern**.

Un pattern è una descrizione, con nome, di un problema di progettazione ricorrente e di una sua soluzione ben provata e che può essere applicata a nuovi contesti.

# Cosa sono i pattern?

Molti pattern, data una categoria di problemi, **guidano l'assegnazione di responsabilità** agli oggetti.

## Pattern

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

## I pattern GRASP e i pattern GoF

La nozione di pattern ebbe origine con i *pattern architettonici* (di costruzione) di *Christopher Alexander*.

I pattern per il software ebbero origine negli anni ottanta con *Ken Beck* (famoso anche per *Extreme Programming*) che riconobbe il lavoro svolto da Alexander nell'architettura, e furono sviluppati da Beck con *Ward Cunningham*.

Nel 1994 viene pubblicato il libro *Design Pattern* di *Gamma, Helm, Johnson e Vlissides* che descrive 23 pattern per la programmazione OO. Questi sono diventati noti come **design pattern GoF (Gang of Four)**.

I GoF sono più degli “schemi di progettazione avanzata” che “principi” (come nel caso di GRASP).

# Riepilogando

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

## Low Representational Gap (LRG)

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

LRG: *va sempre guardato il modello di dominio per trarre ispirazione.*

## I move pattern GRASP

---

# Obiettivi generali dei pattern GRASP

## Obiettivi di GRASP (e dei principi della progettazione del software)

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

# Obiettivi generali dei pattern GRASP

## Obiettivi di GRASP (e dei principi della progettazione del software)

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

## Obiettivi di GRASP e UP

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

## Progettazione modulare

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

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

## Progettazione modulare

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

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

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

# I nove pattern GRASP

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

# I nove pattern GRASP

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

# Creator

---

## Pattern Creator

**Nome:** Creator (Creatore)

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

**Soluzione:** Assegna alla classe B la responsabilità di creare un'istanza della classe A se una delle seguenti condizioni è vera (più sono vere meglio è):

- B “contiene” o aggrega con una composizione oggetti di tipo A
- B registra A<sup>2</sup>
- B utilizza strettamente A
- B possiede i dati per l'inizializzazione di A, che saranno passati ad A al momento della sua creazione (pertanto B è un Expert rispetto alla creazione di A)

---

<sup>2</sup>Registering is saving a object reference of one class to another.

## Esempio per Creator

### Creator

Nell'applicazione POS NextGen, chi deve essere responsabile della creazione di un'istanza di *SalesLineItem*?

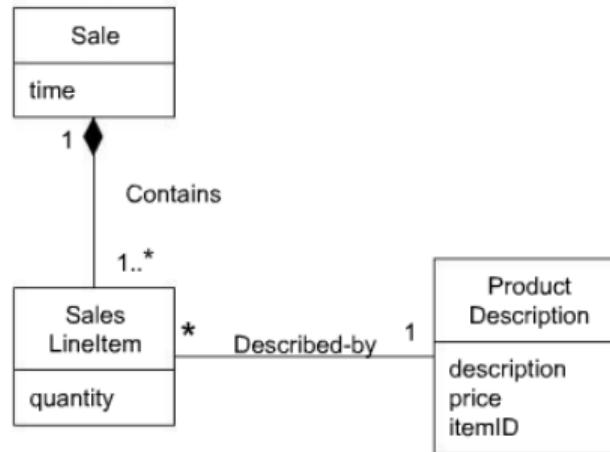
Secondo Creator, si deve cercare una classe che aggrega, contiene, istanza di SalesLineItem.

# Esempio per Creator

## Creator

Nell'applicazione POS NextGen, chi deve essere responsabile della creazione di un'istanza di *SalesLineItem*?

Secondo Creator, si deve cercare una classe che aggrega, contiene, istanza di *SalesLineItem*.



## Esempio per Creator

Secondo Creator, si deve cercare una classe che aggrega, contiene, istanza di *SalesLineItem*.

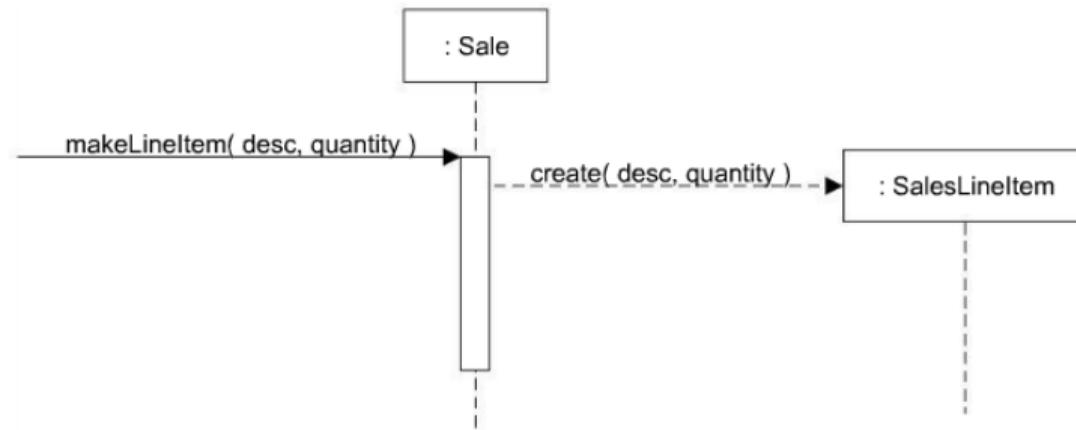
Un oggetto composto è un ottimo candidato per creare le sue parti. Quindi è *Sale* un buon candidato ad avere la responsabilità di creare istanze di *SalesLineItem*.

## Esempio per Creator

Secondo Creator, si deve cercare una classe che aggrega, contiene, istanza di *SalesLineItem*.

Un oggetto composto è un ottimo candidato per creare le sue parti. Quindi è *Sale* un buon candidato ad avere la responsabilità di creare istanze di *SalesLineItem*.

Questa assegnazione di responsabilità richiede che nella classe *Sale* sia definito un metodo in cui avviene la creazione di un oggetto *SalesLineItem*, per esempio un metodo *makeLineItem*.



# Altro esempio per Creator

Si consideri l'esempio del gioco *Monopoly* (si veda libro di testo, Sez. 7.21).

## 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.

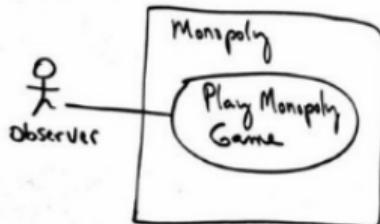


Figura 7.6 Diagramma dei casi d'uso ("diagramma di contesto") per il sistema Monopoly.

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).

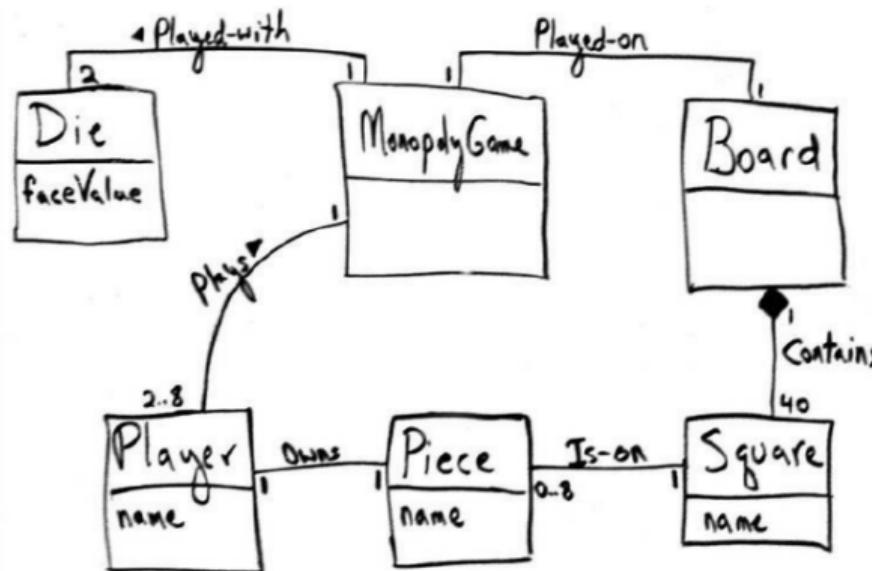
**Requisiti speciali:**

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

## Altro esempio per Creator

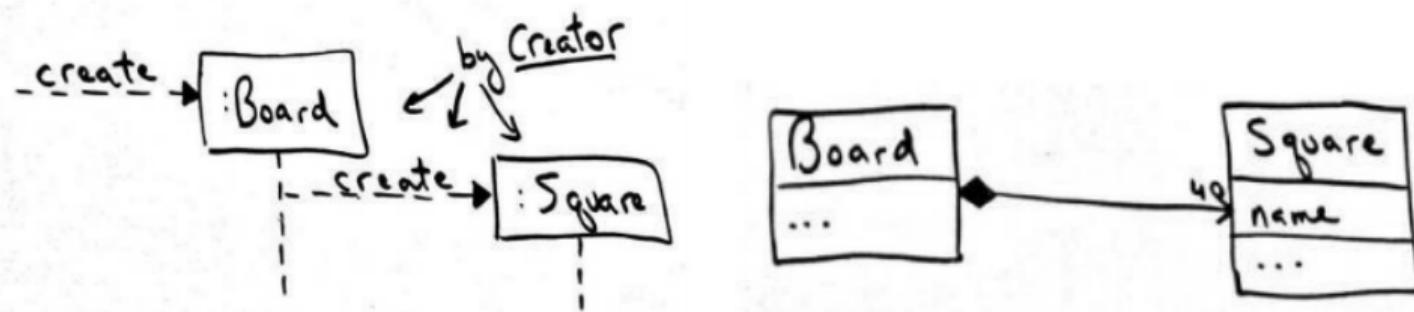
Si consideri l'esempio del gioco *Monopoly* (si veda libro di testo, Sez. 7.21).

Chi è responsabile di creare gli oggetti di tipo *Square*? È possibile notare dal Modello di Dominio che *Board* contiene oggetti *Square*.



## Altro esempio per Creator

Secondo Creator, *Board* creerà oggetti *Square*. Quindi si disegna un *diagramma di sequenza* e un *diagramma delle classi*, entrambi parziali, per riflettere questa decisione di progetto.



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

## Creator: osservazioni

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

# Information Expert

---

# Information Expert (o Expert)

## Pattern Expert

**Nome:** Information Expert (Esperto delle Informazioni)

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

**Soluzione:** Assegna una responsabilità alla classe che possiede le informazioni necessarie per soddisfarla, all'esperto delle informazioni, ovvero alla classe che possiede le informazioni necessarie per soddisfare la responsabilità

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

## Esempio per Expert

Nell'applicazione POS NextGen, qualche classe ha bisogno di conoscere il totale complessivo di una vendita

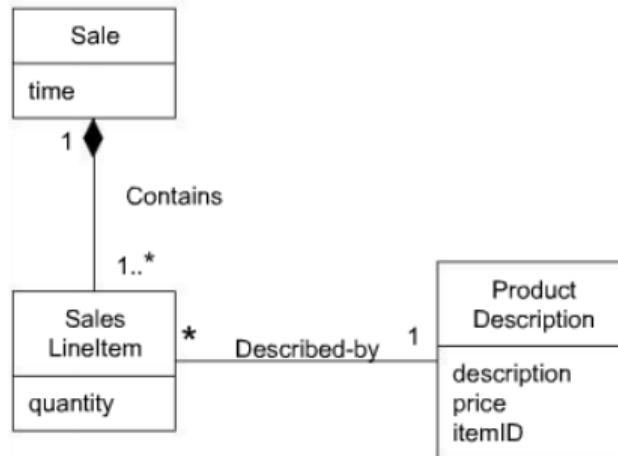
**Enunciare la responsabilità in modo chiaro!**

Chi deve essere responsabile di conoscere il totale complessivo di una vendita?

Quindi analizzare il *Modello di Dominio* (o il *Modello di Progetto* per primo se questo è già sufficientemente sviluppato) per cercare la classe degli oggetti che possiede le informazioni necessarie per il totale.

## Esempio per Expert

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



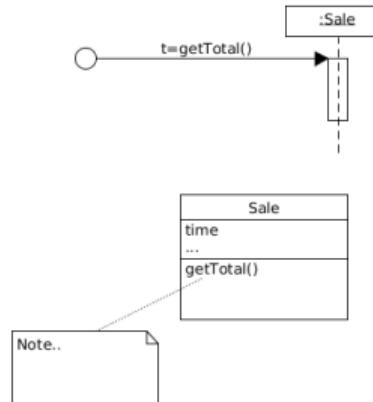
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Nota: quello riportato è il Modello di Dominio.

## Esempio per Expert

Si assegna la responsabilità a *Sale* di conoscere il suo totale, esprimendo questa responsabilità con un metodo chiamato *getTotal*.

Dal modello di dominio al modello di progetto: un diagramma delle classi con l'indicazione dei metodi (salto rappresentazionale basso).



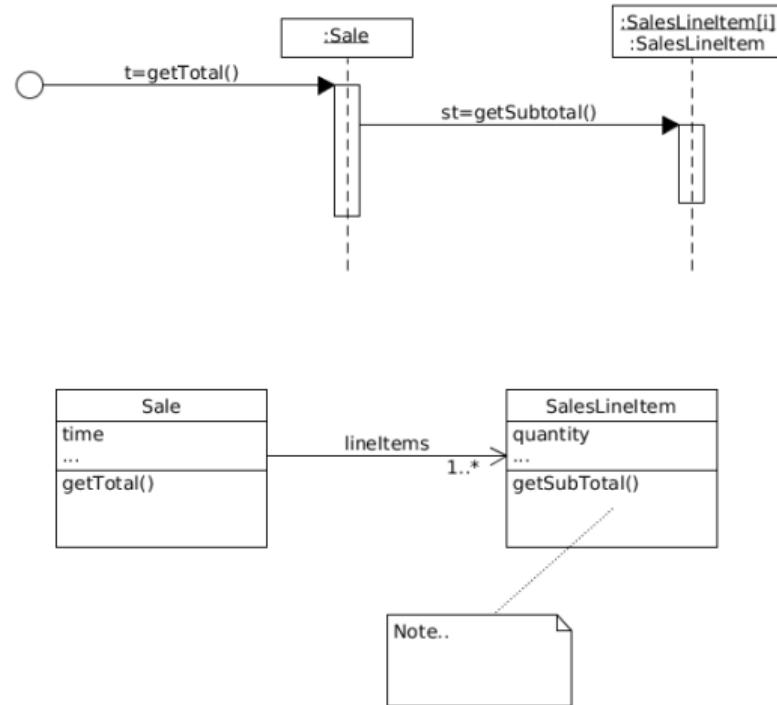
## Esempio per Expert

Quali informazioni sono necessarie per determinare il totale parziale per una riga di vendita per un articolo? *SalesLineItem.quantity* e *ProductDescription.price*, ovvero un oggetto *SalesLineItem* conosce la sua *quantità* e la *ProductDescription* ad esso associata, pertanto per Expert, *SalesLineItem* deve determinare il totale parziale, è l'*esperto delle informazioni*.

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

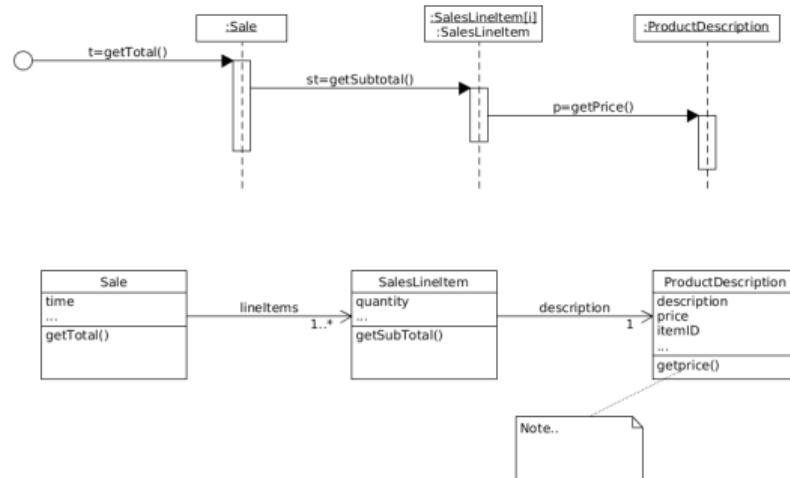
## Esempio per Expert

Quindi *Sale* deve inviare messaggi *getSubtotal* a ciascuna delle sue *SalesLineItem* e sommare i risultati.



# Esempio per Expert

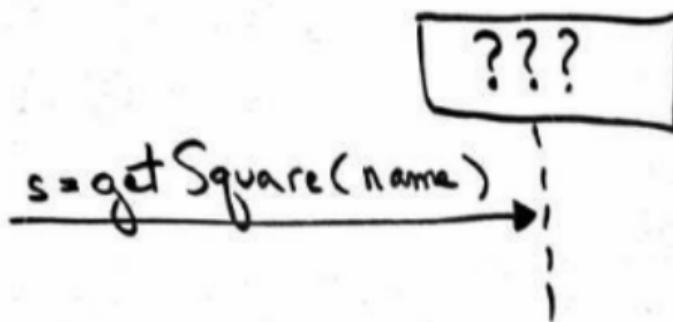
Quindi *Sale* deve inviare messaggi *getSubtotal* a ciascuna delle sue *SalesLineItem* e sommare i risultati. Per soddisfare la sua responsabilità, *SalesLineItem* deve conoscere il prezzo del prodotto a cui si riferisce. *SalesLineItem* invia a *ProductDescription* in messaggio *getPrice* chiedendogli il prezzo del prodotto.



## Altro esempio per Expert

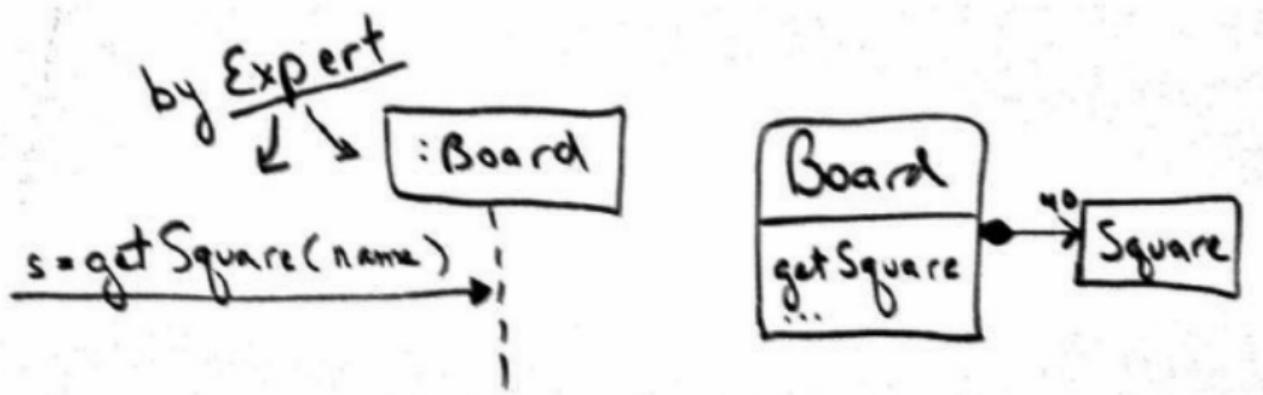
Si consideri l'esempio del gioco *Monopoly* (si veda libro di testo, Sez. 7.21).

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?



## Altro esempio per Expert

Un oggetto software Board aggregherà tutti gli oggetti Square. Pertanto, Board possiede tutte le informazioni necessarie per soddisfare questa responsabilità.



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

## Expert: osservazioni

---

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

## Expert: osservazioni

---

Chi salva un oggetto *Sale* in un DB?

*Sale* (e così tutte le classi) salva i propri oggetti?

## Expert: osservazioni

Chi salva un oggetto *Sale* in un DB?

*Sale* (e così tutte le classi) salva i propri oggetti?

In generale la risposta è **NO!**

- *Sale* avrebbe problemi di (1) coesione, (2) accoppiamento e (3) duplicazione: (1) non si occupa più solo della logica applicativa (meno coesa); (2) accoppiata a classi di sistema e non solo a classi del modello di dominio; (3) applicando la stessa idea a più classi duplico la logica del DB
- principio architetturale di base: separare le diverse logiche in sottosistemi separati, Expert non va sempre bene!

## Low Coupling

---

## Pattern Low Coupling

**Nome:** Low Coupling (Accoppiamento Basso)

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

**Soluzione:** Assegna le responsabilità in modo tale che l'accoppiamento (non necessario) rimanga basso. Usa questo principio per valutare le alternative.

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

## Low Coupling

---



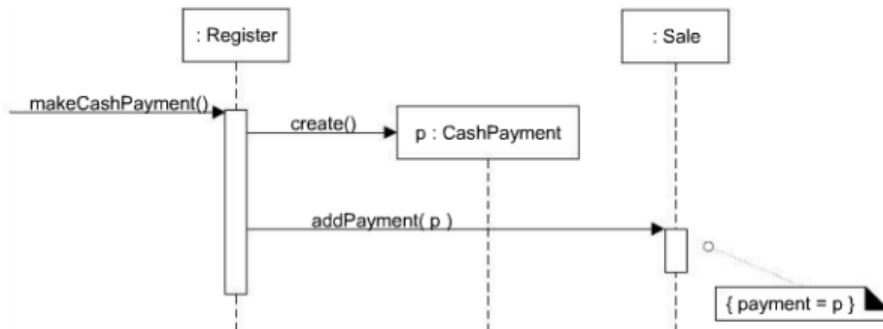
Una classe con un accoppiamento alto (o forte) dipende da molte altre classi. Tali classi fortemente accoppiate possono essere inopportune, e alcune di esse presentano i seguenti problemi:

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

# Esempio per Low Coupling

Siano *CashPayment*, *Register* e *Sale* tre classi nella progettazione dell'applicazione POS NextGen:

- con il pattern Creator si sceglie *Register* come creatore di *Payment*, suggerito dalle responsabilità nel “mondo reale” (registra i pagamenti)
- dunque uso metodo *addPayment(p)* per comunicare con *Sale*



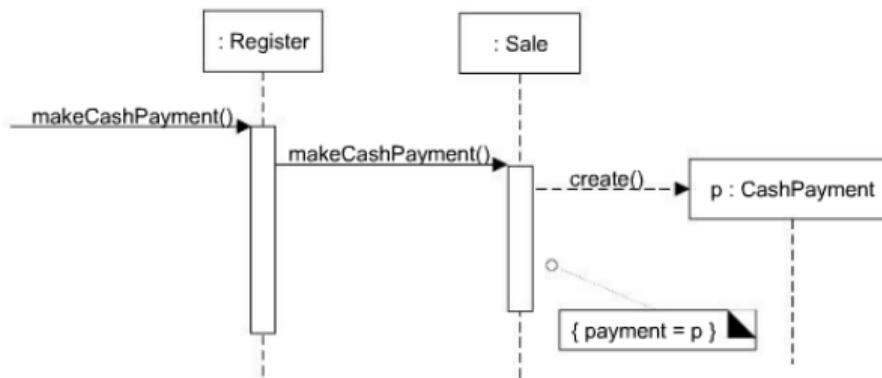
Per determinare gli accoppiamenti si contano le **dipendenze** che sono **direzionali**:

1. *Resister* --> *CashPayment*
2. *Register* --> *Sale*
3. *Sale* --> *CashPayment*

# Esempio per Low Coupling

Soluzione alternativa: *Sale* deve comunque conoscere *Payment*, per cui l'accoppiamento tra *Payment* e *Register* è inutile.

Il seguente progetto non aumenta l'accoppiamento, va preferito poiché mantiene un accoppiamento complessivo più basso.



Per determinare gli accoppiamenti si contano le **dipendenze** che sono **direzionali**:

1. *Register* --> *Sale*
2. *Sale* --> *CashPayment*

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

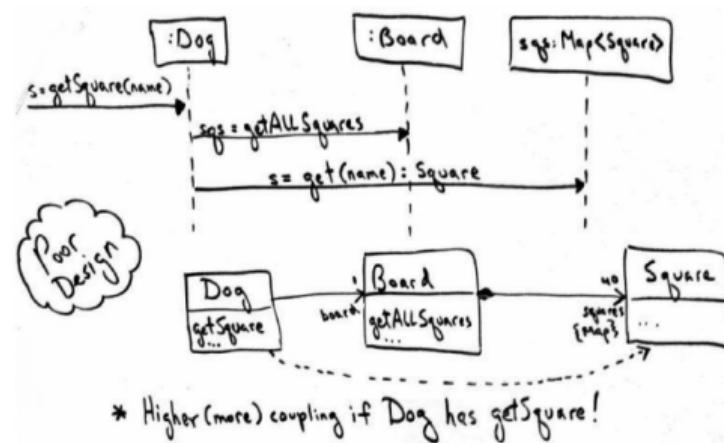
Low Coupling è un **principio di valutazione** da utilizzare in parallelo ad altri pattern.

## Altro esempio per Low Coupling

Si consideri l'esempio del gioco *Monopoly* (si veda libro di testo, Sez. 7.21).

Perché non assegnare l'operazione *getSquare* a *Dog* (ovvero a un'altra classe arbitraria)?

Perché l'accoppiamento complessivo (nel caso di *Board*) è più basso nel progetto con *Board*, e poiché tutto il resto è uguale, risulta migliore del progetto con *Dog*, in termini di sostegno dell'obiettivo di Low Coupling.



## Low Coupling: osservazioni

---

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

- la classe X **ha un attributo** (una variabile d'istanza o un dato membro) di tipo Y o **referenzia** un'istanza di tipo Y o una collezione di oggetti Y

## Low Coupling: osservazioni

---

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

- la classe X **ha un attributo** (una variabile d'istanza o un dato membro) di tipo Y o **referenzia** un'istanza di tipo Y o una collezione di oggetti Y
- un oggetto di tipo X **richiama operazioni o servizi** di un oggetto di tipo Y

## Low Coupling: osservazioni

---

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

- la classe X **ha un attributo** (una variabile d'istanza o un dato membro) di tipo Y o **referenzia** un'istanza di tipo Y o una collezione di oggetti Y
- un oggetto di tipo X **richiama operazioni o servizi** di un oggetto di tipo Y
- un oggetto di tipo X **crea** un oggetto di tipo Y

## Low Coupling: osservazioni

---

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

- la classe X **ha un attributo** (una variabile d'istanza o un dato membro) di tipo Y o **referenzia** un'istanza di tipo Y o una collezione di oggetti Y
- un oggetto di tipo X **richiama operazioni** o **servizi** di un oggetto di tipo Y
- un oggetto di tipo X **crea** un oggetto di tipo Y
- il tipo X ha un metodo che **contiene** un elemento (parametro, variabile locale oppure tipo di ritorno) di tipo Y o che **referenzia** un'istanza di tipo Y

## Low Coupling: osservazioni

---

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

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

## Low Coupling: osservazioni

---

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

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

## Low Coupling: osservazioni

---

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

## Low Coupling: osservazioni

---

- Le classi che sono per natura **generiche** e che hanno un'alta probabilità di **riuso** devono avere un **accoppiamento particolarmente basso**
- Un certo grado moderato di accoppiamento tra le classi è **normale**, anzi è **necessario** per la creazione di un sistema orientato agli oggetti in cui i compiti vengono svolti grazie a una **collaborazione** tra oggetti connessi

## Low Coupling: osservazioni

---

- Le classi che sono per natura **generiche** e che hanno un'alta probabilità di **riuso** devono avere un **accoppiamento particolarmente basso**
- Un certo grado moderato di accoppiamento tra le classi è **normale**, anzi è **necessario** per la creazione di un sistema orientato agli oggetti in cui i compiti vengono svolti grazie a una **collaborazione** tra oggetti connessi
- Una sottoclasse è fortemente accoppiata alla sua superclasse. Si consideri attentamente ogni decisione di **estendere** una superclasse, poiché è una forma di **accoppiamento forte**

## Low Coupling: osservazioni

- Le classi che sono per natura **generiche** e che hanno un'alta probabilità di **riuso** devono avere un **accoppiamento particolarmente basso**
- Un certo grado moderato di accoppiamento tra le classi è **normale**, anzi è **necessario** per la creazione di un sistema orientato agli oggetti in cui i compiti vengono svolti grazie a una **collaborazione** tra oggetti connessi
- Una sottoclasse è fortemente accoppiata alla sua superclasse. Si consideri attentamente ogni decisione di **estendere** una superclasse, poiché è una forma di **accoppiamento forte**
- Porzioni di **codice duplicato** sono **fortemente accoppiate** tra di loro; infatti, la modifica di una copia spesso implica la necessità di modificare anche le altre copie

## Low Coupling: osservazioni

Un accoppiamento alto con elementi *stabili* o con elementi *pervasivi* costituisce raramente un problema.

## Low Coupling: osservazioni

Un accoppiamento alto con elementi *stabili* o con elementi *pervasivi* costituisce raramente un problema.

### Low coupling

Il problema infatti non è l'accoppiamento alto di per sé, ma **l'accoppiamento alto con elementi per certi aspetti instabili**, per esempio nell'interfaccia, nell'implementazione o per loro pura e semplice presenza

# Low Coupling: osservazioni

Un accoppiamento alto con elementi *stabili* o con elementi *pervasivi* costituisce raramente un problema.

## Low coupling

Il problema infatti non è l'accoppiamento alto di per sé, ma **l'accoppiamento alto con elementi per certi aspetti instabili**, per esempio nell'interfaccia, nell'implementazione o per loro pura e semplice presenza

## Vantaggi

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

## High Cohesion

---

## Pattern High Cohesion

**Nome:** High Cohesion (Coesione Alta)

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

**Soluzione:** Assegna le responsabilità in modo tale che la coesione rimanga alta. Usa questo principio per valutare alternative.

La coesione (funzionale) è una misura di quanto fortemente siano **correlate** e **concentrate** le responsabilità di un elemento.

## Pattern High Cohesion

**Nome:** High Cohesion (Coesione Alta)

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

**Soluzione:** Assegna le responsabilità in modo tale che la coesione rimanga alta. Usa questo principio per valutare alternative.

La coesione (funzionale) è una misura di quanto fortemente siano **correlate** e **concentrate** le responsabilità di un elemento.

Un elemento con responsabilità **altamente correlate** che non esegue una **quantità di lavoro eccessiva** ha una *coesione alta*.

## High Cohesion

---



Una classe con una *coesione bassa* fa molte cose non correlate tra loro o svolge troppo lavoro.  
Tali classi presentano i seguenti problemi:

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

## High Cohesion

---



Una classe con una *coesione bassa* fa molte cose non correlate tra loro o svolge troppo lavoro.  
Tali classi presentano i seguenti problemi:

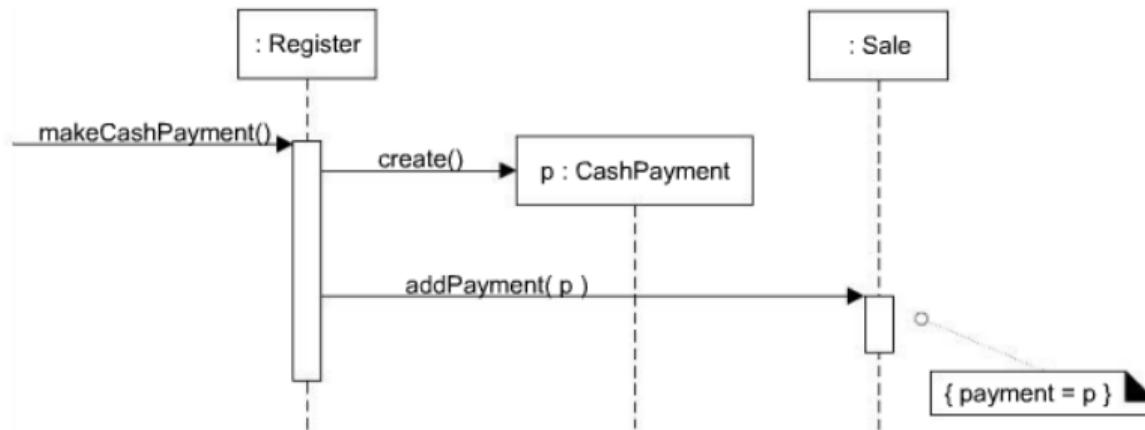
- sono difficili da comprendere
- sono difficili da mantenere
- sono difficili da riusare
- sono delicate; sono continuamente soggette a cambiamenti

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

## Esempio per High Cohesion

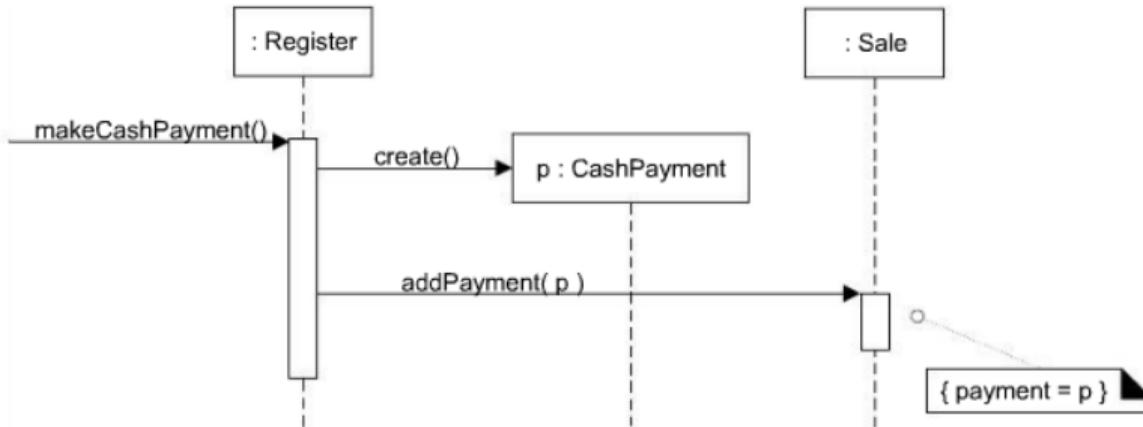
Siano *CashPayment*, *Register* e *Sale* tre classi di progetto dell'applicazione POS NextGen:

- con il pattern Creator si sceglie *Register* come creatore di *Payment*, suggerito dalle responsabilità nel “mondo reale” (registra i pagamenti)
- dunque uso metodo *addPayment(p)* per comunicare con *Sale*



## Esempio per High Cohesion

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

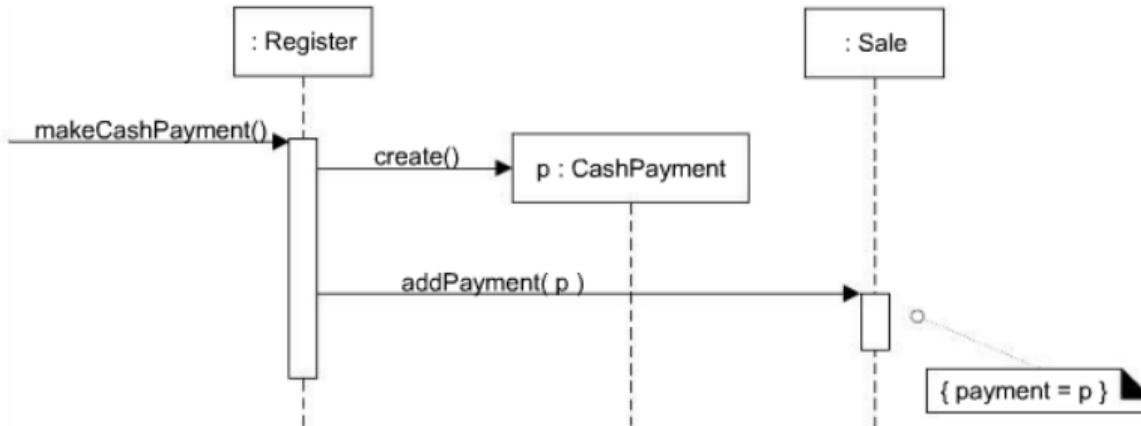


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Questa scelta significa che il *Register* si assume non solo la responsabilità di ricevere l'operazione di sistema *makeCashPayment*, ma anche parte della responsabilità di soddisfarla.

## Esempio per High Cohesion

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

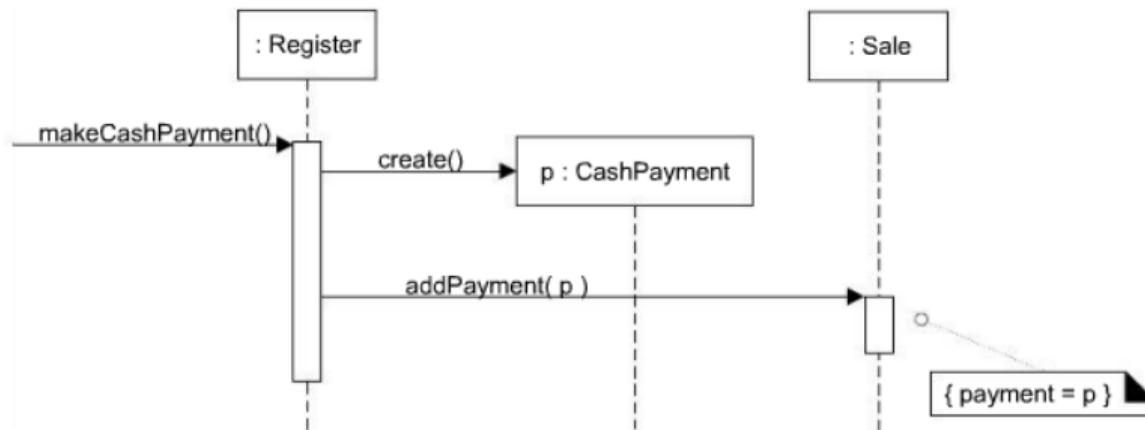


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Se si continua a rendere la classe **Register** responsabile di eseguire una parte del lavoro o l'intero lavoro relativo a sempre più operazioni di sistema, essa diventerà sempre più carica di compiti, e diventerà **non coesa**.

## Esempio per High Cohesion

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



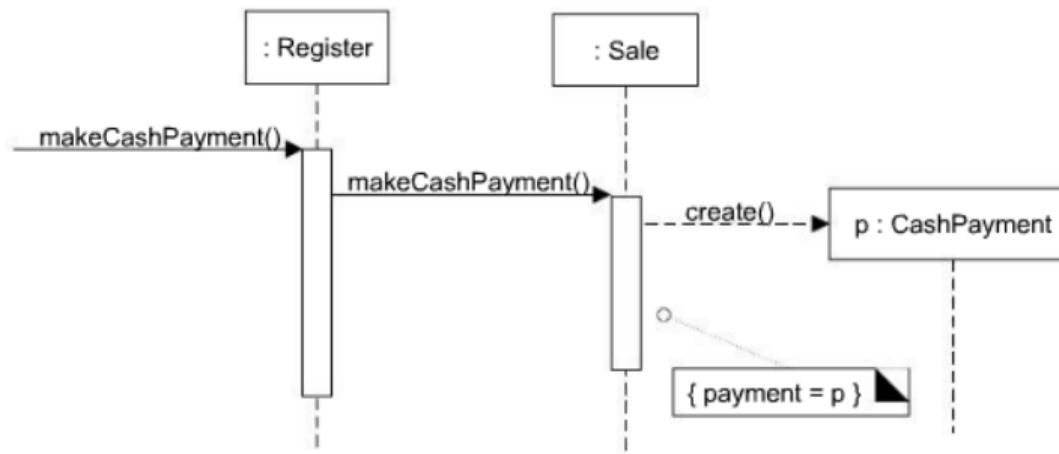
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Si suppongano molte operazioni di sistema ricevute da **Register**, se eseguisse il lavoro relativo a ciascuna operazione, diventerebbe un oggetto “gonfio” e *non coeso*.

## Esempio per High Cohesion

Soluzione alternativa: la responsabilità della creazione del pagamento è **delegata** alla *Sale* e sostiene una coesione più alta di *Register*.

Questo progetto che sostiene una coesione alta e un accoppiamento bassa, è da preferire.



## High Cohesion: osservazioni

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

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

- **Coesione di dati:** una classe implementa un tipo di dati (*molto buona*)

## High Cohesion: osservazioni

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

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

- **Coesione di dati:** una classe implementa un tipo di dati (*molto buona*)
- **Coesione funzionale:** gli elementi di una classe svolgono una singola funzione (*buona o molto buona*)

## High Cohesion: osservazioni

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

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

- **Coesione di dati:** una classe implementa un tipo di dati (*molto buona*)
- **Coesione funzionale:** gli elementi di una classe svolgono una singola funzione (*buona o molto buona*)
- **Coesione temporale:** gli elementi sono raggruppati perché usati circa nello stesso tempo (es. controller, *a volte buona a volte meno*)

## High Cohesion: osservazioni

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

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

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

## High Cohesion: osservazioni

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

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

## High Cohesion: osservazioni

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

- High Cohesion è la coesione funzionale, una misura di quanto sono correlate le responsabilità e le operazioni di un elemento software, una misura di quanto lavoro esegue un elemento software:
  - un elemento ha coesione alta se ha responsabilità (funzionali) altamente correlate e se “non svolge troppo lavoro”

## High Cohesion: osservazioni

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

- High Cohesion è la coesione funzionale, una misura di quanto sono correlate le responsabilità e le operazioni di un elemento software, una misura di quanto lavoro esegue un elemento software:
  - un elemento ha coesione alta se ha responsabilità (funzionali) altamente correlate e se “non svolge troppo lavoro”
  - un elemento ha coesione bassa se fa molte cose scorrelate o se svolge troppo lavoro

## High Cohesion: osservazioni

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

- High Cohesion è la coesione funzionale, una misura di quanto sono correlate le responsabilità e le operazioni di un elemento software, una misura di quanto lavoro esegue un elemento software:
  - un elemento ha coesione alta se ha responsabilità (funzionali) altamente correlate e se “non svolge troppo lavoro”
  - un elemento ha coesione bassa se fa molte cose scorrelate o se svolge troppo lavoro
- Grady Booch: c'è una coesione funzionale alta quando gli elementi di un componente (es. classe) “lavorano tutti insieme per fornire un comportamento ben circoscritto”

## High Cohesion: osservazioni

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

- High Cohesion è la coesione funzionale, una misura di quanto sono correlate le responsabilità e le operazioni di un elemento software, una misura di quanto lavoro esegue un elemento software:
  - un elemento ha coesione alta se ha responsabilità (funzionali) altamente correlate e se “non svolge troppo lavoro”
  - un elemento ha coesione bassa se fa molte cose scorrelate o se svolge troppo lavoro
- Grady Booch: c’è una coesione funzionale alta quando gli elementi di un componente (es. classe) “lavorano tutti insieme per fornire un comportamento ben circoscritto”

Nota: non è possibile dare una misura *assoluta* di quando la coesione di un progetto sia troppo bassa, è una misura *relativa*.

## High Cohesion: osservazioni

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

- **Coesione molto bassa:** una classe è la sola responsabile di molte cose in aree funzionali molto diverse

*Esempio:* una classe responsabile dell'interazione con le basi di dati e della gestione delle chiamate remote.

## High Cohesion: osservazioni

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

- **Coesione molto bassa:** una classe è la sola responsabile di molte cose in aree funzionali molto diverse
- **Coesione bassa:** una classe ha da sola la responsabilità di un compito complesso in una sola area funzionale

*Esempio:* una classe responsabile di tutte le interazioni con le basi di dati, i metodi sono correlati ma sono troppo numerosi e la quantità di codice di supporto è molto consistente, meglio suddividerla in una famiglia di classi leggere che condividono il compito di fornire l'accesso alle basi di dati.

## High Cohesion: osservazioni

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

- **Coesione molto bassa:** una classe è la sola responsabile di molte cose in aree funzionali molto diverse
- **Coesione bassa:** una classe ha da sola la responsabilità di un compito complesso in una sola area funzionale
- **Coesione alta:** una classe ha responsabilità moderate in un'area funzionale e collabora con altre classi per svolgere i suoi compiti  
*Esempio:* una classe che è responsabile solo in parte dell'interazione con le basi di dati.

## High Cohesion: osservazioni

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

- **Coesione molto bassa:** una classe è la sola responsabile di molte cose in aree funzionali molto diverse
- **Coesione bassa:** una classe ha da sola la responsabilità di un compito complesso in una sola area funzionale
- **Coesione alta:** una classe ha responsabilità moderate in un'area funzionale e collabora con altre classi per svolgere i suoi compiti
- **Coesione moderata:** una classe ha, da sola, responsabilità in poche aree diverse, che sono logicamente correlate al concetto rappresentato dalla classe, ma non l'una all'altra  
*Esempio:* una classe *Company* che è completamente responsabile di conoscere i suoi dipendenti e conoscere le proprie informazioni finanziarie, queste non sono aree correlate anche se entrambe sono logicamente legate al concetto di una *Company*.

## Regola pratica

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

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

L'elevato grado di correlazione delle funzionalità, combinato a un numero ridotto di operazioni, semplifica la manutenzione e l'**evoluzione** (siamo in UP!). Inoltre sostiene un maggiore potenziale di riuso.

## High Cohesion: osservazioni

---

Tuttavia, in alcune situazioni è accettabile una coesione bassa:

- Se per un determinato compito di programmazione/manutenzione ci vuole un esperto (es. raggruppare tutte le istruzioni SQL di un sistema in una sola classe)
- per gli oggetti distribuiti lato server (in caso di RMI o middleware affini): per avere migliori prestazioni

## Vantaggi

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

# **Controller**

---

## Pattern Controller

**Nome:** Controller (Controllore)

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

**Soluzione:** Assegna la responsabilità a un oggetto che rappresenta una delle seguenti scelte:

1. rappresenta il “*sistema*” complessivo, un “oggetto radice”, un dispositivo all’interno del quale viene eseguito il software, un punto di accesso al software o un sottosistema principale (variante del *facade controller*)
2. rappresenta uno scenario di un *caso d’uso* all’interno del quale si verifica l’operazione di sistema (un *controller di caso d’uso* o *controller di sessione*)

## Controller

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

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

# Controller

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

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

Nota: le classi dello strato UI (le classi “finestre”, “viste” e “documenti”) non sono Controller, non devono svolgere i compiti associati agli eventi di sistema, ma normalmente ricevono questi eventi e li delegano a un controller.

# Controller

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

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

Nota: le classi dello strato UI (le classi “finestre”, “viste” e “documenti”) non sono Controller, non devono svolgere i compiti associati agli eventi di sistema, ma normalmente ricevono questi eventi e li delegano a un controller.

Altra nota: i Controller sono classi che ricevono/gestiscono i messaggi legati alle operazioni di sistema (coordinano), ma non fanno molto di più, **non** sono classi di dominio.

# Controller

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

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

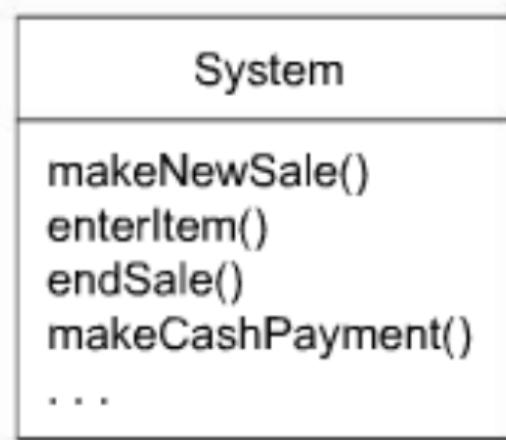
Nota: le classi dello strato UI (le classi “finestre”, “viste” e “documenti”) non sono Controller, non devono svolgere i compiti associati agli eventi di sistema, ma normalmente ricevono questi eventi e li delegano a un controller.

Altra nota: i Controller sono classi che ricevono/gestiscono i messaggi legati alle operazioni di sistema (coordinano), ma non fanno molto di più, **non** sono classi di dominio.

Ancora una nota: posso controllare che gli eventi avvengano in un ordine prestabilito (es. *makePayment* dopo *endSale*).

## Un esempio di Controller

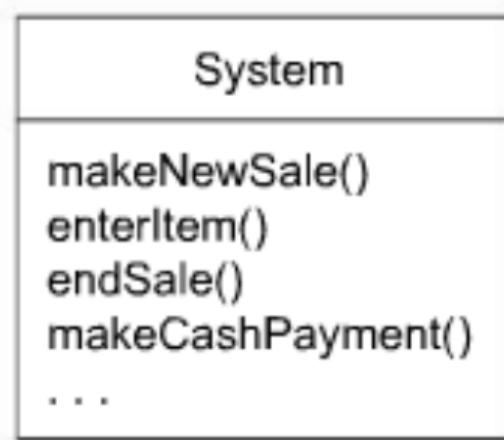
Durante l'analisi, le operazioni di sistema possono essere assegnate alla classe System in un modello di analisi, per indicare che si tratta di operazioni di sistema.



## Un esempio di Controller

Durante l'analisi, le operazioni di sistema possono essere assegnate alla classe System in un modello di analisi, per indicare che si tratta di operazioni di sistema.

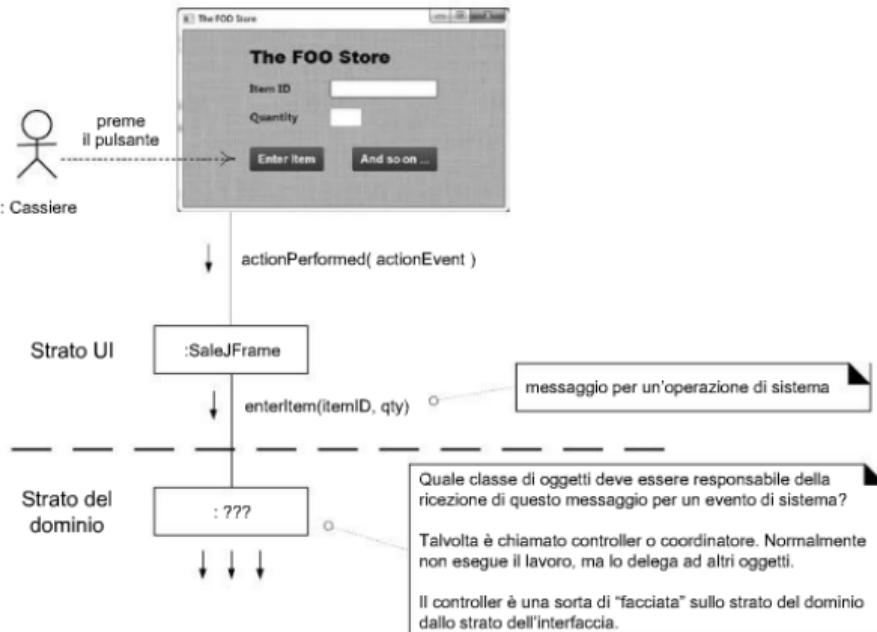
Nota: ciò non significa che nel progetto c'è una classe software chiamata System che le gestisce; al contrario, durante la progettazione, la responsabilità delle operazioni di sistema è assegnata a una classe controller.



# Un esempio di Controller

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

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

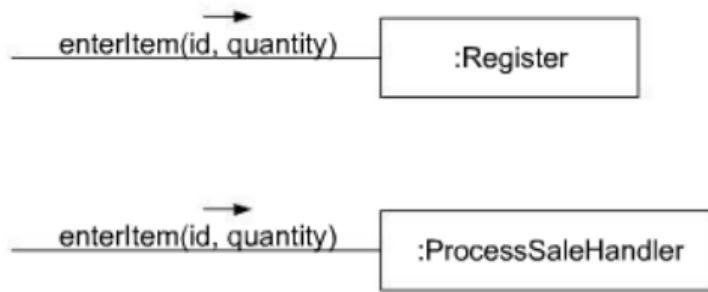


# Un esempio di Controller

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

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

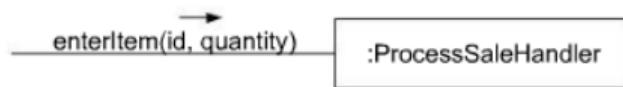
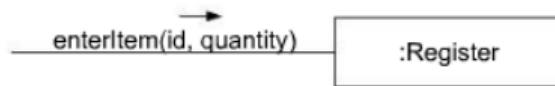
**Due possibili soluzioni.**



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

# Un esempio di Controller

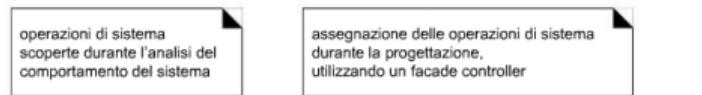
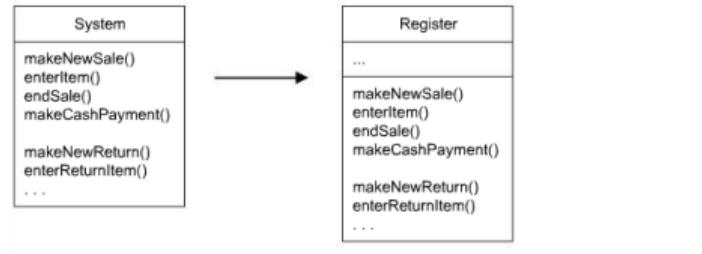
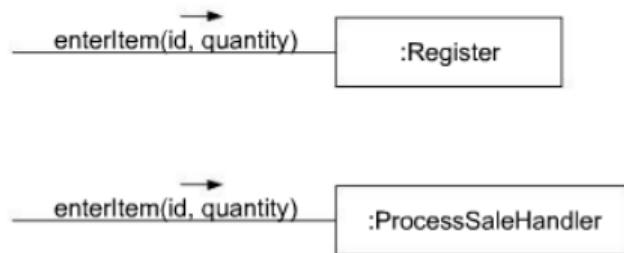
Due possibili soluzioni:



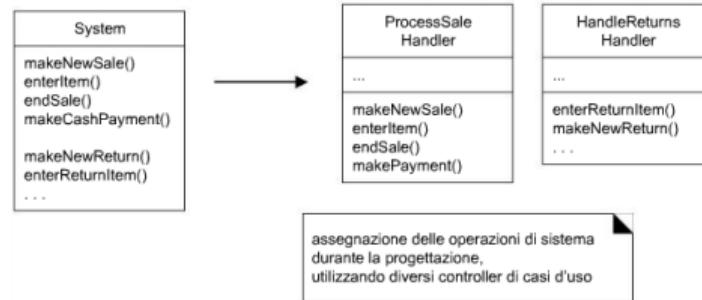
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Un esempio di Controller

Due possibili soluzioni:



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



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

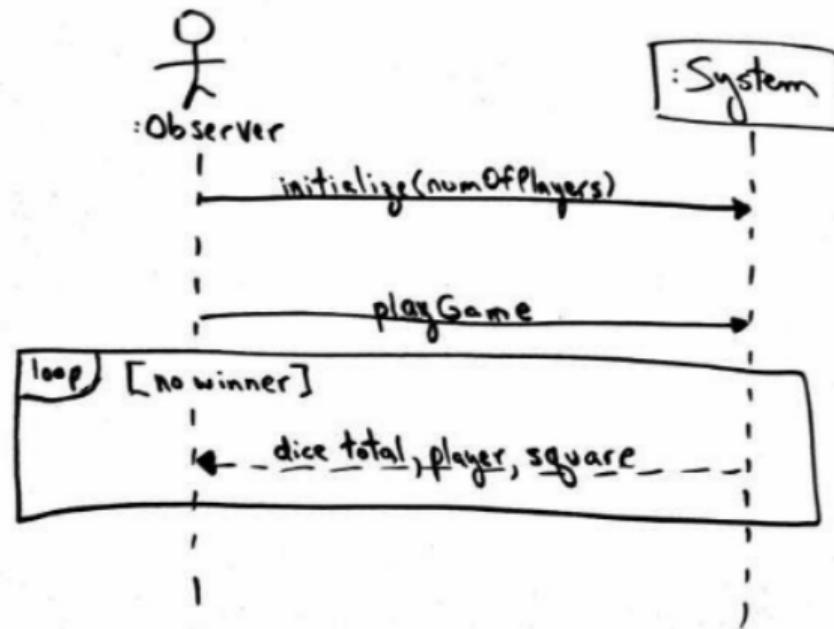
# Un altro esempio di Controller

Il pattern Controller risponde alla domanda: *qual è il primo oggetto, dopo o oltre lo strato dell'interfaccia utente, che deve ricevere il messaggio dallo strato UI?*

In base al *Principio di Separazione Modello-Vista* (si veda 08).

**Architettura logica e organizzazione in layer**), gli oggetti della UI **non** devono contenere logica applicativa, devono **delegare** (inoltrare il compito a un altro oggetto) la richiesta agli oggetti di dominio nello strato del dominio.

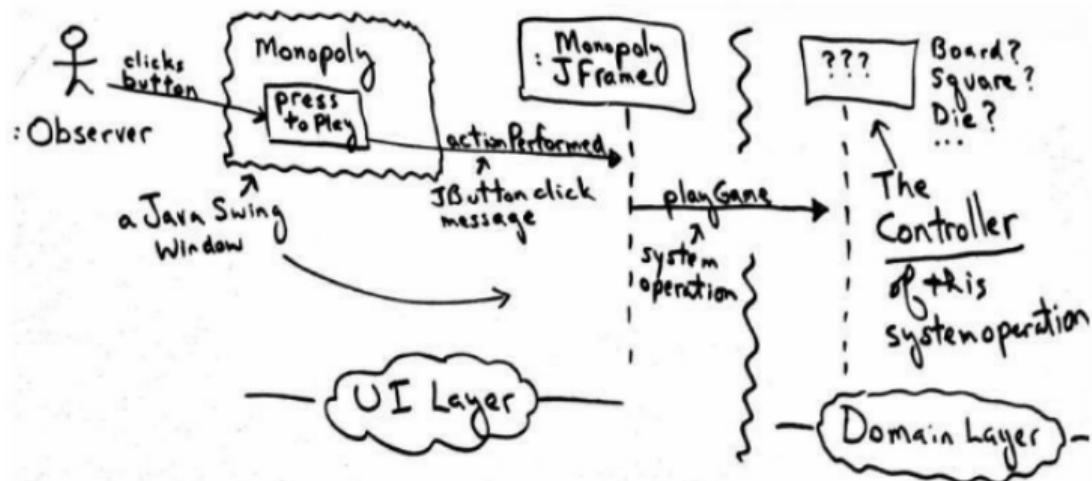
Nell'esempio, chi gestisce *playGame*?



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

# Un altro esempio di Controller

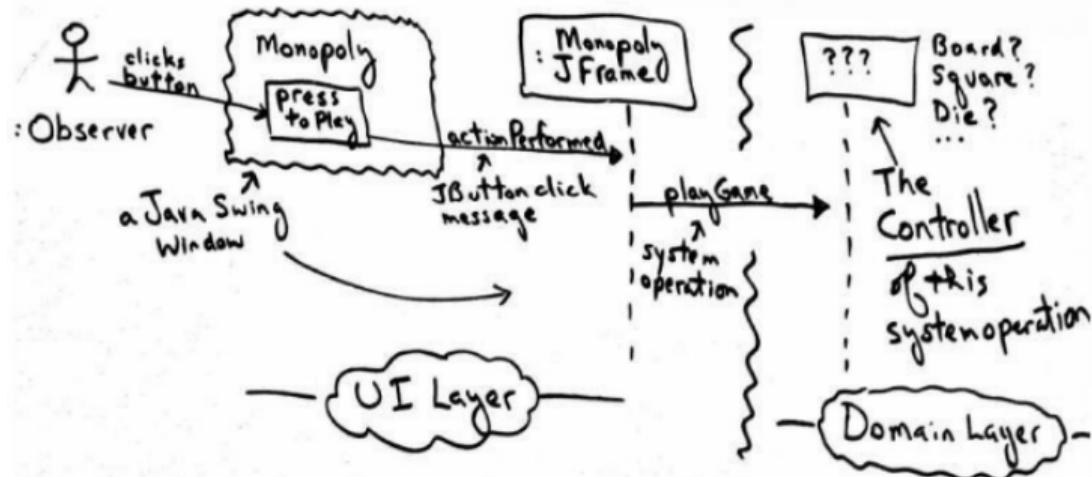
La finestra *JFrame* deve adattare quel messaggio *actionPerformed* a qualcosa di semanticamente più significativo, come un messaggio *playGame* (per corrispondere all'analisi del SSD), e **delegare** il messaggio *playGame* a un oggetto di dominio nello strato del dominio.



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

# Un altro esempio di Controller

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



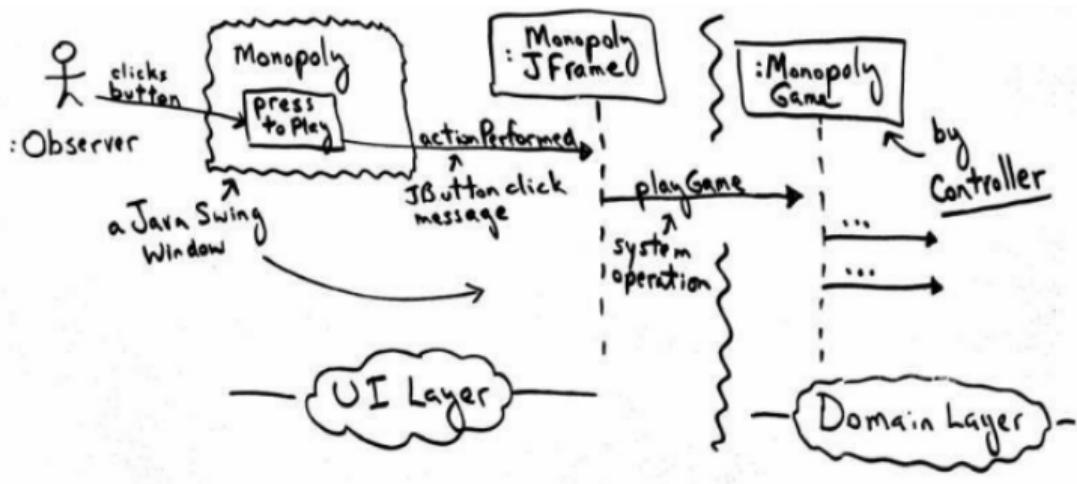
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Un altro esempio di Controller

## Opzione 1:

un oggetto che rappresenta il “sistema” complessivo o un “oggetto radice”, per esempio un oggetto chiamato MonopolyGame.

*L'opzione 1 è ragionevole in questo caso, poiché ci sono solo poche operazioni di sistema possibili.*

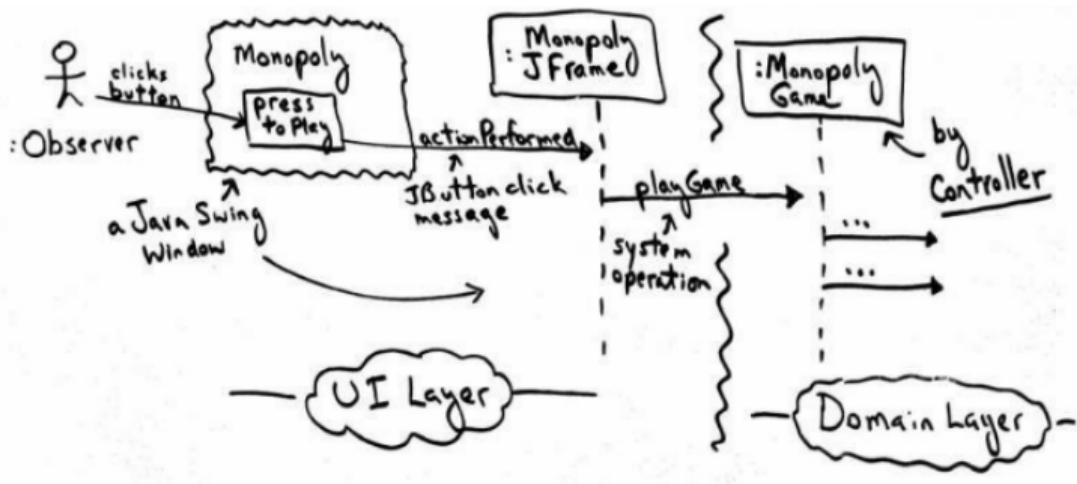


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Un altro esempio di Controller

## Opzione 2:

un oggetto che rappresenta un dispositivo all'interno del quale è eseguito il software (per dispositivi hardware specializzati).

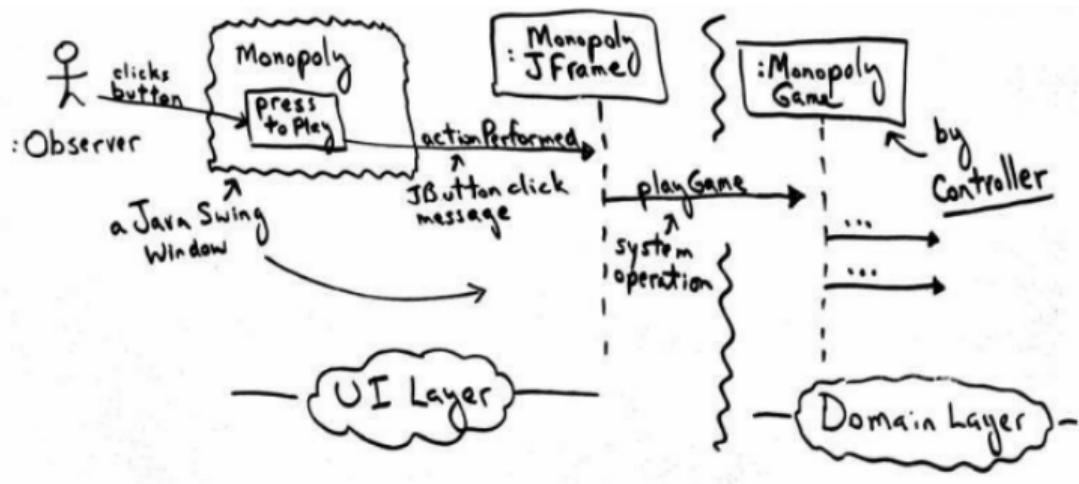


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# Un altro esempio di Controller

## Opzione 3:

un oggetto che rappresenta il caso d'uso o la sessione.



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

# Controller: osservazioni

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

## Controller come delega

Gli oggetti dello strato UI catturano gli eventi di sistema generati dagli attori.

Gli oggetti dello strato UI devono delegare le richieste di lavoro ad oggetti di un altro strato.

# Controller: osservazioni

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

## Controller come delega

Gli oggetti dello strato UI catturano gli eventi di sistema generati dagli attori.

Gli oggetti dello strato UI devono delegare le richieste di lavoro ad oggetti di un altro strato.

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

# Controller: osservazioni

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

## Controller come delega

Gli oggetti dello strato UI catturano gli eventi di sistema generati dagli attori.

Gli oggetti dello strato UI devono delegare le richieste di lavoro ad oggetti di un altro strato.

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

Il controller è una sorta di “facciata” dello strato del dominio dallo strato UI.

# Controller: osservazioni

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

## Controller come delega

Gli oggetti dello strato UI catturano gli eventi di sistema generati dagli attori.

Gli oggetti dello strato UI devono delegare le richieste di lavoro ad oggetti di un altro strato.

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

Il controller è una sorta di “facciata” dello strato del dominio dallo strato UI.

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

### Importante!

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

## Controller: osservazioni

### Importante!

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

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

## Controller: osservazioni

---

Utilizzare la stessa classe controller per tutti gli eventi di sistema di un unico caso d'uso, in questo modo il controller può **conservare le informazioni sullo stato del caso d'uso**.

È utile, ad esempio, per identificare degli eventi di sistema “fuori sequenza” (rispetto all'SSD).

È utile, ad esempio, per ricordare lo stato della sessione o della conversazione (es. la vendita in corso).

Queste sono tipiche responsabilità assegnate all'oggetto controller.

## Controller: osservazioni

---

Utilizzare la stessa classe controller per tutti gli eventi di sistema di un unico caso d'uso, in questo modo il controller può **conservare le informazioni sullo stato del caso d'uso**.

È utile, ad esempio, per identificare degli eventi di sistema “fuori sequenza” (rispetto all'SSD).

È utile, ad esempio, per ricordare lo stato della sessione o della conversazione (es. la vendita in corso).

Queste sono tipiche responsabilità assegnate all'oggetto controller.

Gli oggetti controller appartengono spesso allo strato del dominio ma, nei sistemi più complessi, possono appartenere a uno strato Application separato, collocato tra lo strato UI e lo strato del dominio.

## Corollario

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

# Controller: osservazioni

Le nozioni di controller MVC e controller GRASP sono distinte!

## Controller MVC

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

## Controller GRASP

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

# Controller: osservazioni

Le nozioni di controller MVC e controller GRASP sono distinte!

## Controller MVC

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

## Controller GRASP

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

Entrambe si occupano di gestire le richieste provenienti dall'utente ma a livelli di astrazione differenti.

In generale, il controller MVC delega le richieste di lavoro dell'utente al controller GRASP del dominio.

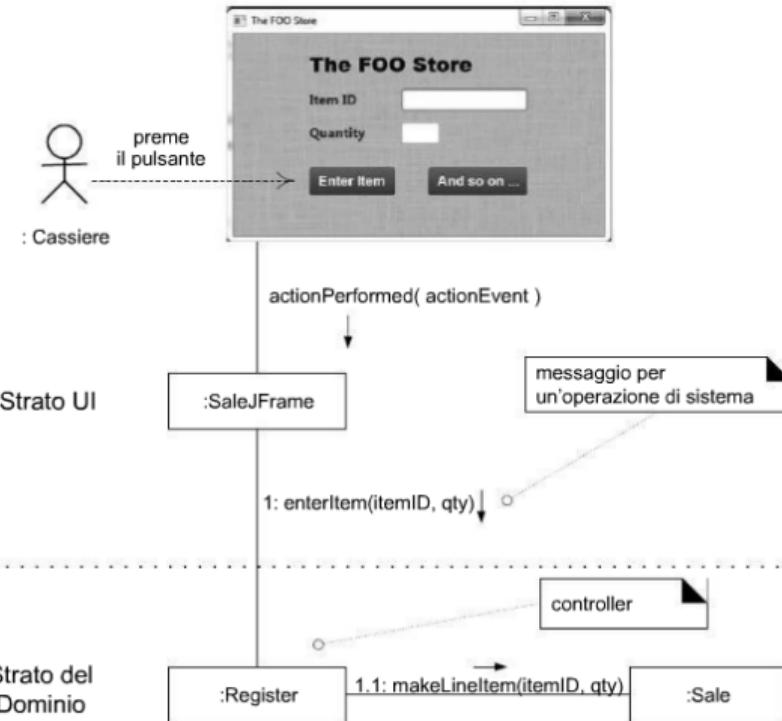
# Controller: osservazioni

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

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

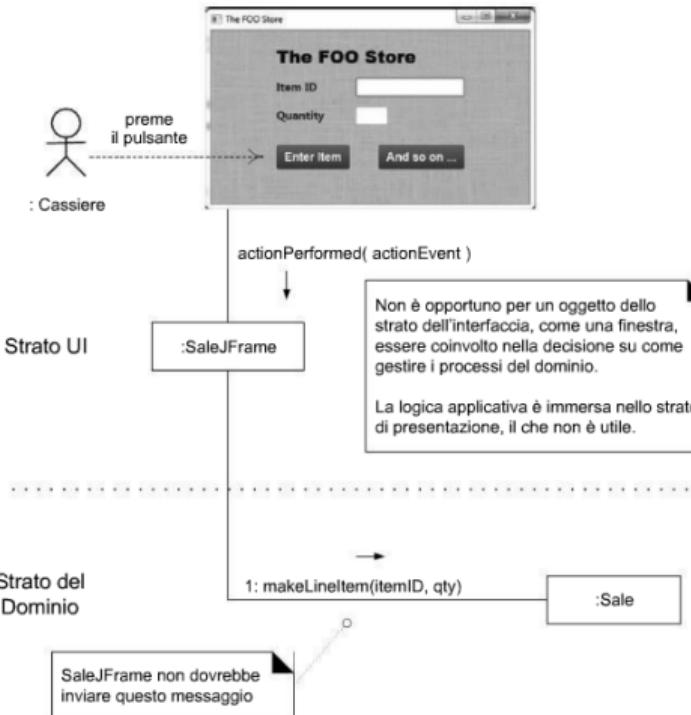
# Controller: osservazioni

Accoppiamento accettabile dallo strato UI allo strato del dominio.



# Controller: osservazioni

Accoppiamento meno opportuno dello strato dell'interfaccia allo strato del dominio.



## Controller: osservazioni

Il pattern Controller suggerisce due categorie di controller: i **facade controller** o i **controller di caso d'uso**.

## Controller: osservazioni

Il pattern Controller suggerisce due categorie di controller: i **facade controller** o i **controller di caso d'uso**.

### Facade controller

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

Sono adatti quando non ci sono “troppi” eventi di sistema o quando l’interfaccia utente UI non può reindirizzare i messaggi per gli eventi di sistema a più controller alternativi.

# Controller: osservazioni

Il pattern Controller suggerisce due categorie di controller: i **facade controller** o i **controller di caso d'uso**.

## Controller di caso d'uso

Un controller diverso per ogni caso d'uso.

Non è un oggetto di dominio, ma un costrutto artificiale per supportare il sistema (*Pure Fabrication*).

Lo si sceglie quando la collocazione delle responsabilità in un facade controller porta a progetti con coesione bassa o accoppiamento alto, ovvero quando il facade controller diventa “gonfio” di eccessive responsabilità.

## Controller: vantaggi

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

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

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

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

# 11 . Esempio di Progettazione con i pattern GRASP

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

# Table of contents

1. Esempi di progettazione con GRASP
2. Collegare lo strato UI allo strato del dominio e inizializzazione

## **Esempi di progettazione con GRASP**

---

## Importante

L'assegnazione delle responsabilità e la progettazione delle collaborazioni sono passi molto importanti e creativi della progettazione, sia durante la creazione dei diagrammi che durante la codifica.

Però, l'assegnazione delle responsabilità e la scelta delle collaborazioni può e deve essere spiegata in modo razionale.

# Realizzazione di un caso d'uso

## Realizzazione di un caso d'uso (o di uno scenario)

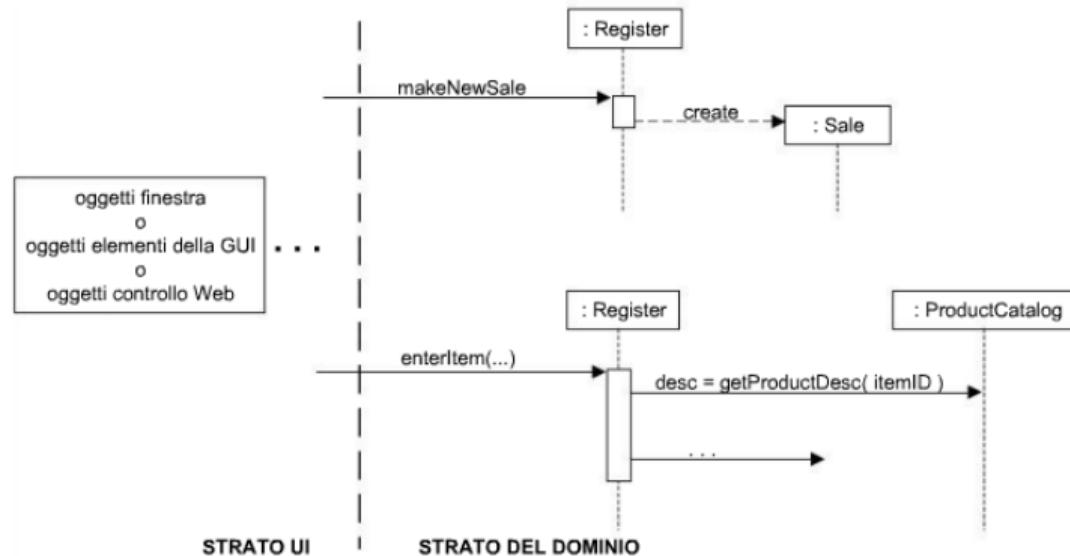
Una realizzazione di un caso d'uso descrive come viene realizzato un caso d'uso all'interno del Modello di Progetto, in termini di oggetti che collaborano.

- Il caso d'uso suggerisce le operazioni di sistema, mostrate negli SSD
- Le operazioni di sistema diventano i messaggi iniziali entranti nei controller per i diagrammi di interazione per lo strato del dominio
- I diagrammi di interazione per lo strato del dominio illustrano come gli oggetti interagiscono per soddisfare i compiti richiesti, ovvero la realizzazione di caso d'uso

# Gestione delle operazioni di sistema

Le operazioni di sistema negli SSD sono usate come messaggi iniziali per gli oggetti controller dello strato del dominio.

Gli eventi di sistema sono catturati dallo strato UI che li “trasforma” in operazioni di sistema.



## Esempio: makeNewSale

L'operazione di sistema *makeNewSale* avviene quando un cassiere inizia una richiesta per avviare una nuova vendita, dopo che un cliente è giunto alla cassa con degli articoli da acquistare.

### Contratto CO1: makeNewSale

---

**Operazione:** makeNewSale()

**Riferimenti:** Casi d'Uso: Elabora Vendita

**Pre-condizioni:** nessuna.

**Post-condizioni:**

- è stata creata un'istanza s di Sale.
- s è stata associata con il Register.
- gli attributi di s sono stati inizializzati.

## Esempio: makeNewSale, controller

La prima scelta di progetto riguarda la scelta del controller per il messaggio dell'operazione di sistema (*controller*).

- Un oggetto di sistema complessivo, l'oggetto radice, un dispositivo speciale, un punto d'accesso o un sottosistema principale: *Store*, *Register*, *POS-Terminal* sono possibili scelte
- Un oggetto che rappresenta un ricevitore o un gestore di tutti gli eventi di sistema di uno scenario di caso d'uso: *ProcessSaleHandler*, *ProcessSaleSession* sono possibili scelte

## Esempio: makeNewSale, controller

La prima scelta di progetto riguarda la scelta del controller per il messaggio dell'operazione di sistema (*controller*).

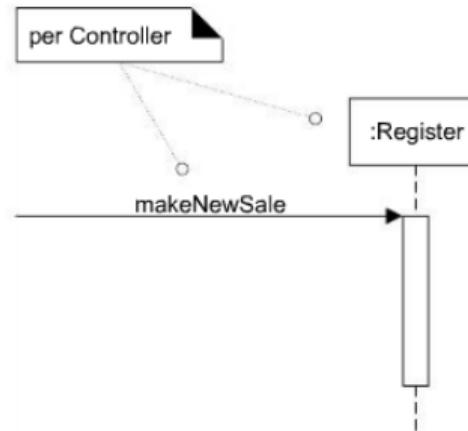
- Un oggetto di sistema complessivo, l'oggetto radice, un dispositivo speciale, un punto d'accesso o un sottosistema principale: *Store*, *Register*, *POS-Terminal* sono possibili scelte
- Un oggetto che rappresenta un ricevitore o un gestore di tutti gli eventi di sistema di uno scenario di caso d'uso: *ProcessSaleHandler*, *ProcessSaleSession* sono possibili scelte

La scelta di *facade controller* relativa a un oggetto-dispositivo come *Register* è soddisfacente se ci sono solo poche operazioni di sistema, e se il facade controller non assume troppe responsabilità (cioè non diventa poco coeso).

# Esempio: makeNewSale, controller

Si noti che

Register è un oggetto software nel *Modello di Progetto*, non un registratore di cassa fisico.



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

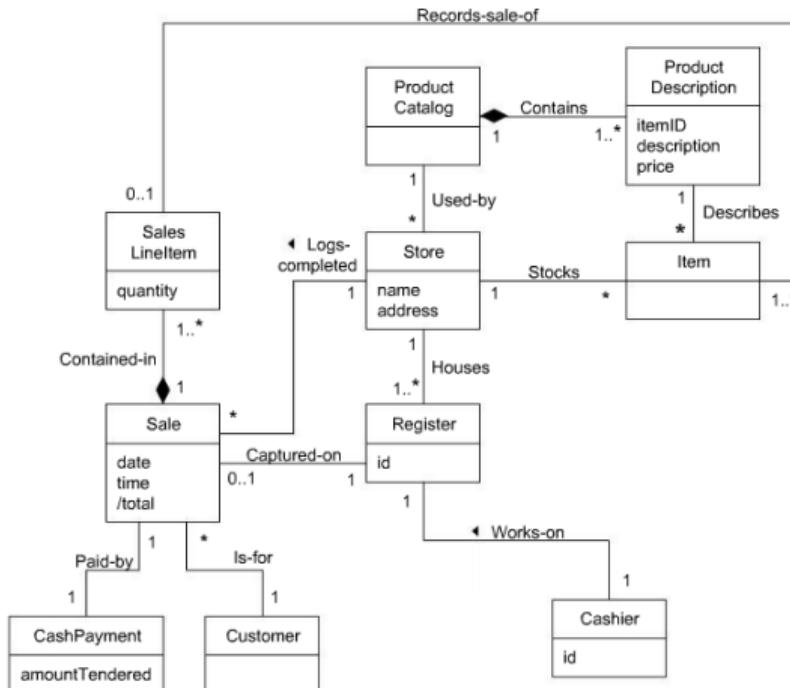
# Esempio: makeNewSale, prima post-condizione “è stata creata un’istanza s di Sale”

Una post-condizione del contratto indica la creazione di un oggetto *concettuale Sale*.

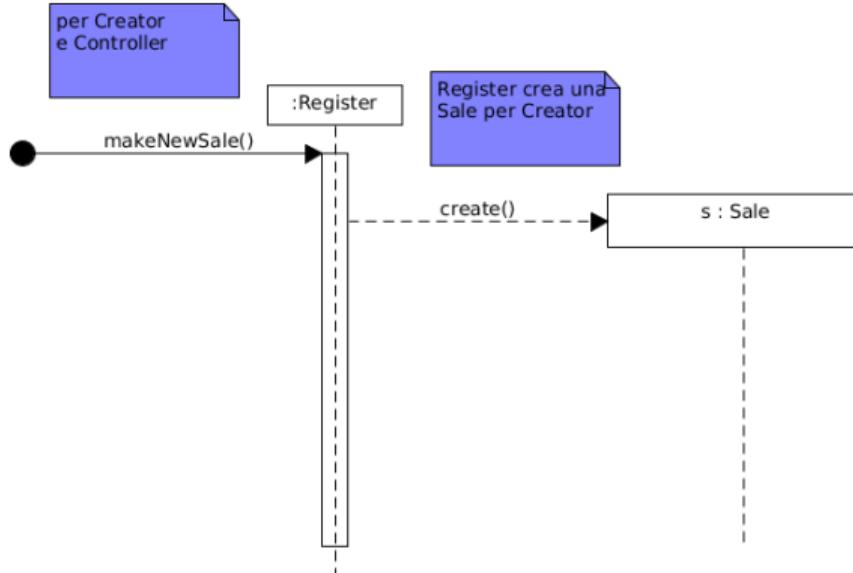
Il pattern GRASP Creator, suggerisce di assegnare la responsabilità per la creazione a una classe che aggrega, contiene o registra l’oggetto da creare.

- *Register* può essere considerato come il registratore di una *Sale*
- *Store* registra vendite, e pertanto va considerato un altro candidato creatore di una *Sale*

*Store* registra vendite completate, la *Sale* da creare va considerata, al momento, solo un tentativo di vendita. Pertanto *Register* è un candidato ragionevole per la creazione di una *Sale*.



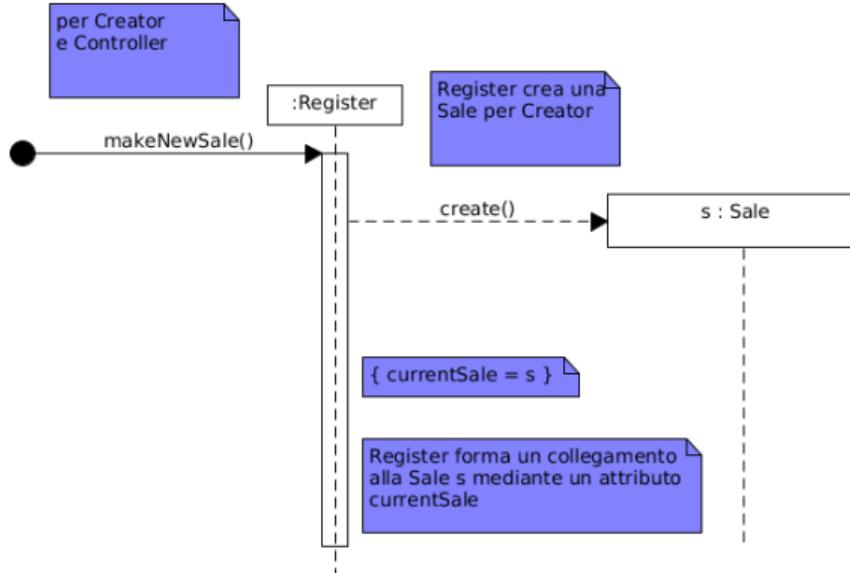
## Esempio: makeNewSale, diagramma di interazione



Esempio: `makeNewSale`, seconda post-condizione “*s* è stata associata con il *Register*”

Dopo che il *Register* ha creato la *Sale*, è possibile associare la *Sale* al *Register* per tutta l'esecuzione del caso d'uso, in modo che durante le operazioni successive all'interno della sessione il *Register* mantenga un riferimento all'istanza corrente nell'attributo *currentSale*.

## Esempio: makeNewSale, diagramma di interazione



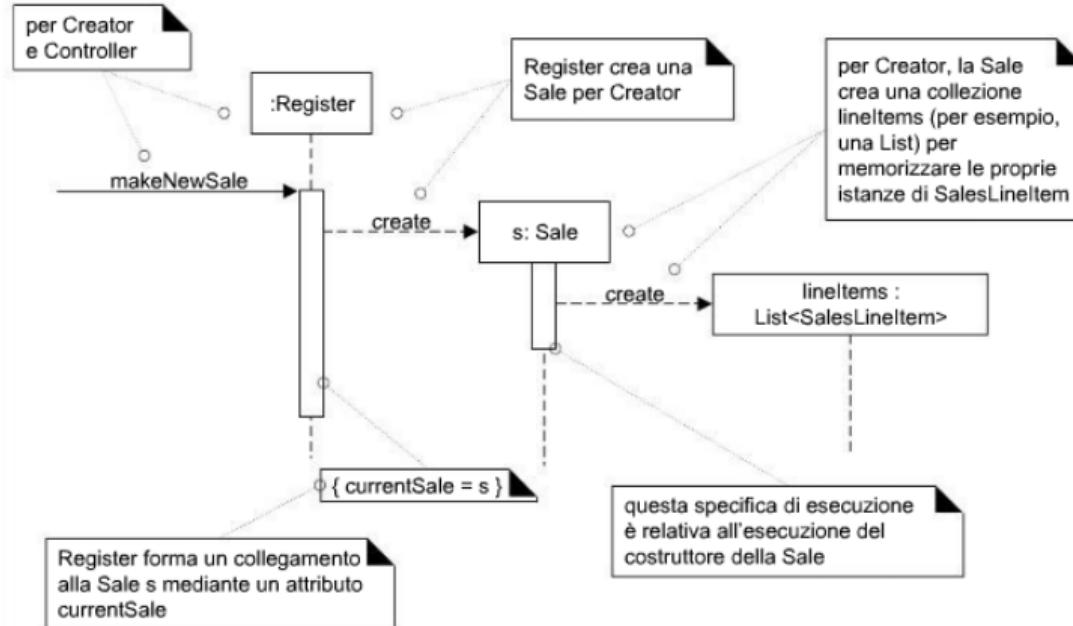
Esempio: `makeNewSale`, terza post-condizione “gli attributi di `s` sono stati inizializzati”

La `Sale` deve creare una collezione vuota per memorizzare tutte le future istanze di `SalesLineItem` che verranno aggiunte alla vendita.

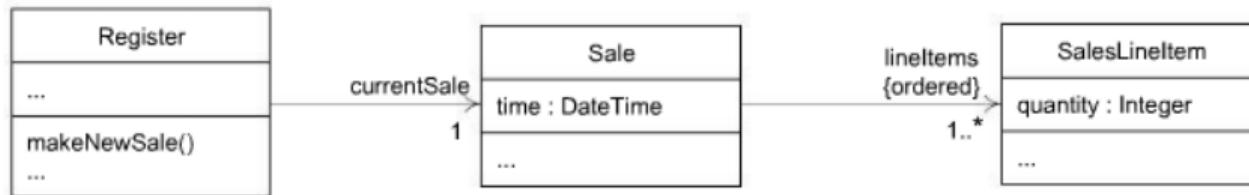
Questa collezione sarà associata all’istanza `Sale` e mantenuta dalla stessa. Quindi `Sale` è un buon candidato anche per la sua creazione

Pertanto il `Register` crea la `Sale`, la `Sale` crea una collezione vuota di `SalesLineItem`, e infine il `Register` memorizza la `Sale` corrente.

# Esempio: makeNewSale, diagramma di interazione



## Esempio: makeNewSale, diagramma delle classi



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

## Esempio: enterItem

L'operazione di sistema *enterItem* avviene quando un cassiere inserisce l'*itemID* e la quantità di un articolo da acquistare.

### 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.

– sli è stata associata con la Sale corrente s.

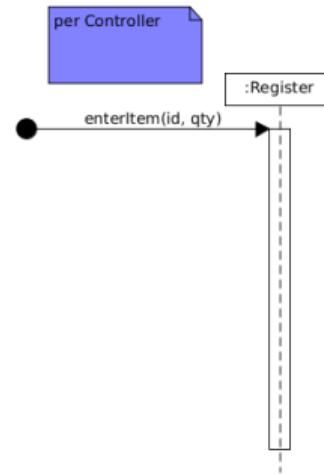
– sli è stata associata con una ProductDescription, in base alla corrispondenza con itemID.

– sli.quantity è diventata quantity.

## Esempio: `enterItem`, controller

Il pattern *Controller* suggerisce di utilizzare la stessa classe controller per tutte le operazioni di sistema di un caso d'uso; pertanto si continuerà a utilizzare *Register* come controller per l'operazione *enterItem* e per le ulteriori operazioni di sistema del caso d'uso *Elabora Vendita*.

## Esempio: enterItem, diagramma di interazione

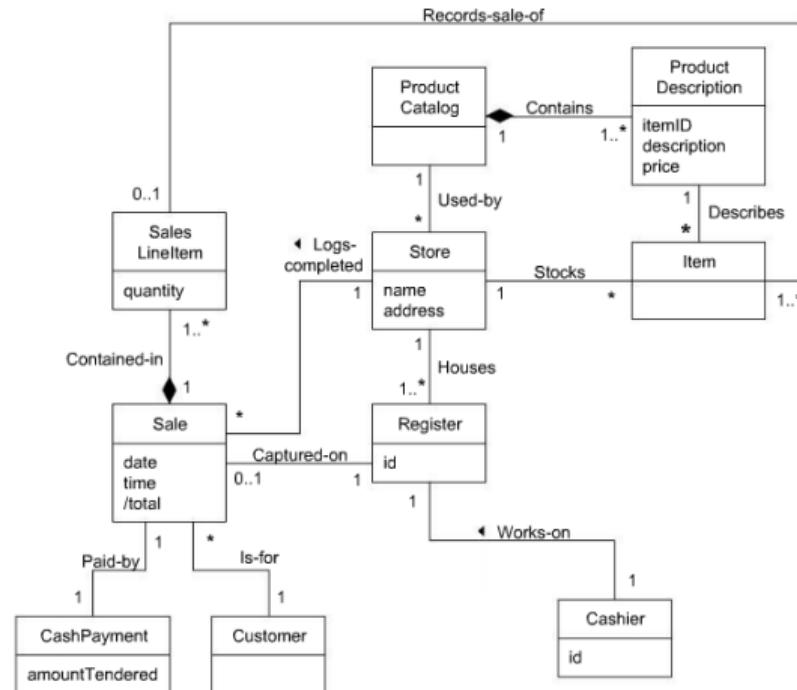


# Esempio: enterItem, prima post-condizione “è stata creata un’istanza sli di SalesLineItem”

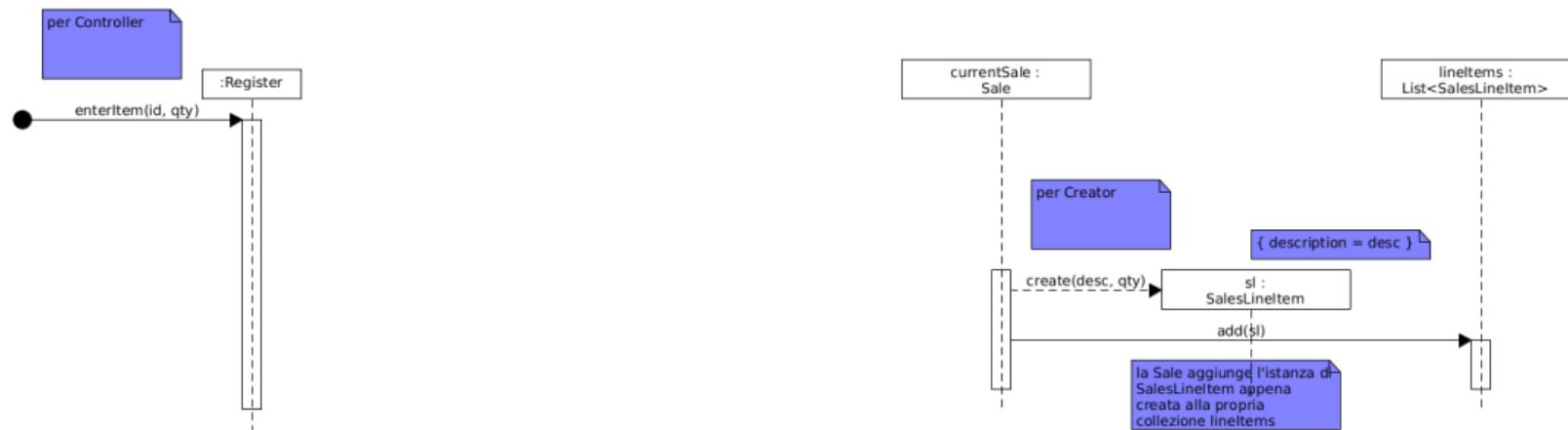
È necessario creare, inizializzare e associare una *SalesLineItem*.

- Una *Sale* software (traendo ispirazione dal Modello di Dominio) contenere degli oggetti software *SalesLineItem*. Per *Creator*, una *Sale* software è un candidato appropriato per la creazione di una *SalesLineItem*
- Un altro candidato creatore di *SalesLineItem* è il controller *Register*, poiché possiede tutti i dati necessari per l'inizializzazione della nuova vendita

*Creator* suggerisce di preferire la *Sale*, in virtù della composizione tra *Sale* e *SalesLineItem*.



# Esempio: enterItem, diagramma di interazione



**Esempio: enterItem, seconda, quarta post-condizione “sli è stata associata con la Sale corrente s” e “sli.quantity è diventata quantity”**

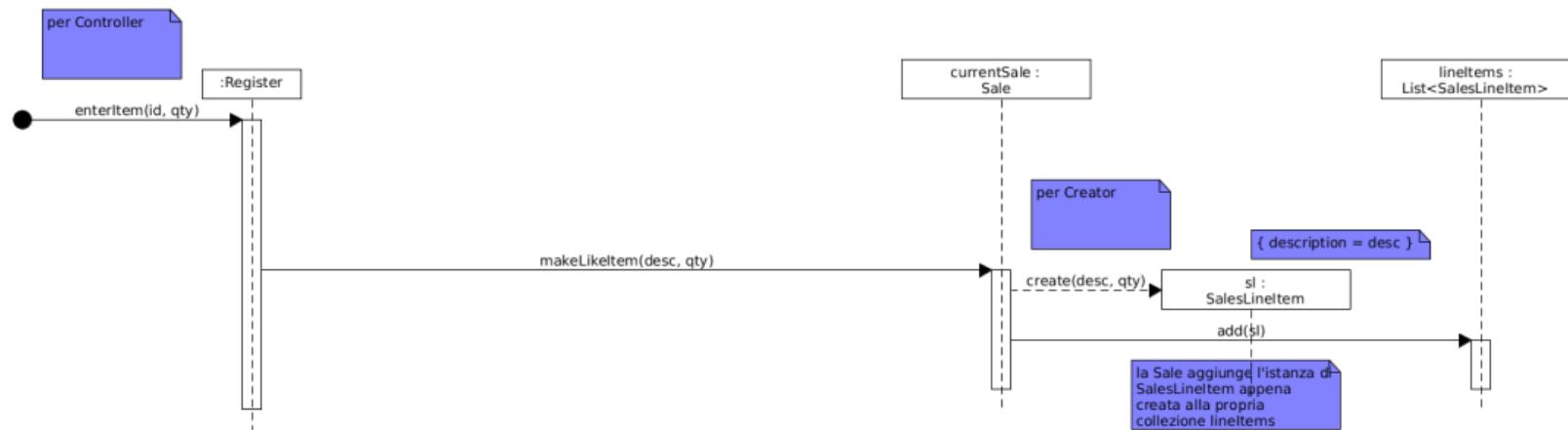
La *Sale* può essere collegata con la *SalesLineItem* appena creata memorizzando la nuova istanza nella sua collezione di righe di articoli.

Pertanto, per *Creator*, viene inviato un messaggio *makeLineItem* a una *Sale* affinché essa crei una *SalesLineItem*. I parametri per il messaggio *makeLineItem* comprendono:

- *ProductDescription* che corrisponde all'*itemID*
- *quantity*

*SalesLineItem* memorizza tali parametri.

# Esempio: enterItem, diagramma di interazione



**Esempio: `enterItem`, terza post-condizione “sli è stata associata con una `ProductDescription`, in base alla corrispondenza con `itemID`”**

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

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

### Responsabilità

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

# Esempio: enterItem, terza post-condizione “sli è stata associata con una ProductDescription, in base alla corrispondenza con itemId”

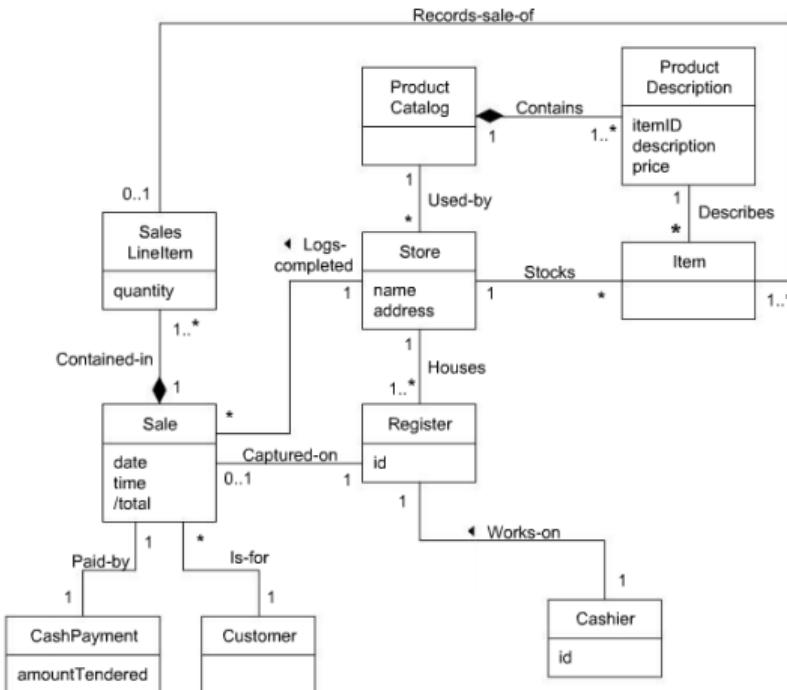
Dal Modello di Dominio, *ProductCatalog*

contiene logicamente tutte le

*ProductDescription*.

Per *Information Expert*, *ProductCatalog* è un buon candidato per questa responsabilità di ricerca, poiché conosce tutti gli oggetti *ProductDescription*.

La responsabilità può essere rappresentata con un metodo chiamato *getProductDescription*. Il *ProductCatalog* può soddisfare la responsabilità di conoscere gli oggetti *ProductDescription* utilizzando una mappa *descriptions*.



Esempio: `enterItem`, terza post-condizione “sli è stata associata con una `ProductDescription`, in base alla corrispondenza con `itemID`”

### Responsabilità

Chi deve inviare il messaggio `getProductDescription` al `ProductCatalog` per cercare una `ProductDescription`?

**Esempio: enterItem, terza post-condizione “sli è stata associata con una ProductDescription, in base alla corrispondenza con itemId”**

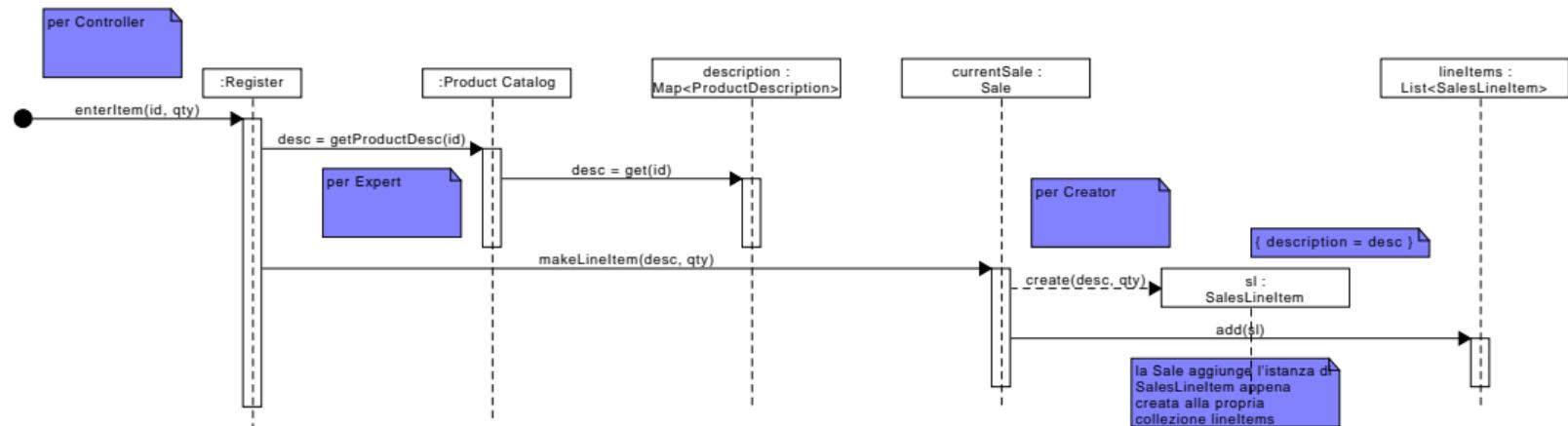
### Responsabilità

Chi deve inviare il messaggio *getProductDescription* al *ProductCatalog* per cercare una *ProductDescription*?

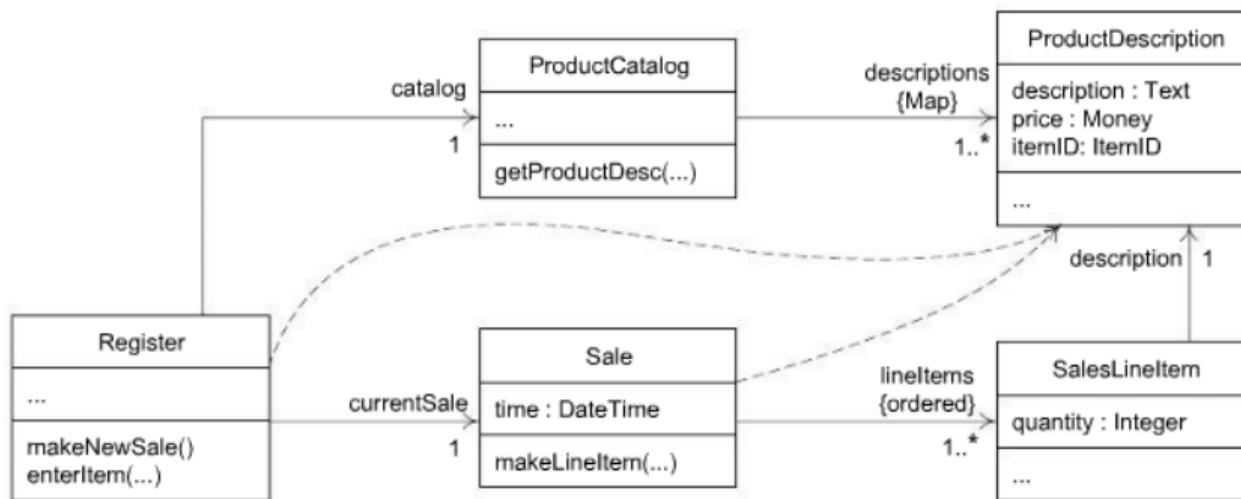
È ragionevole pensare che durante l'avviamento iniziale siano stati creati degli oggetti di lunga vita *Register* e *ProductCatalog* e che l'oggetto *Register* sia stato connesso in modo permanente all'oggetto *ProductCatalog*.

*Register* può inviare il messaggio *getProductDescription* al *ProductCatalog*.

# Esempio: enterItem, diagramma di interazione



## Esempio: enterItem, diagramma delle classi



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

## Esempio: endSale

L'operazione di sistema *endSale* avviene quando un cassiere preme un pulsante per indicare la fine dell'inserimento degli articoli in una vendita.

### Contratto CO3: endSale

---

**Operazione:** endSale()

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

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

**Post-condizioni:** – nessuna.

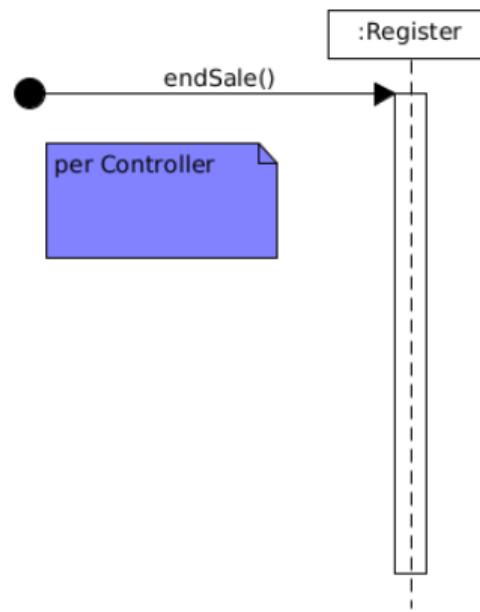
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

## Esempio: endSale, controller

---

Il pattern *Controller* suggerisce di utilizzare la stessa classe controller per tutte le operazioni di sistema di un caso d'uso; pertanto si continuerà a utilizzare *Register* come controller per l'operazione *endSale* e per le ulteriori operazioni di sistema del caso d'uso *Elabora Vendita*.

## Esempio: endSale, diagramma di interazione



## Esempio: endSale, post-condizioni

---

Durante l'analisi a oggetti, per l'operazione di sistema *endSale* non sono state identificate post-condizioni. Dunque questa operazione si configura principalmente come un'interrogazione, per calcolare e visualizzare il *totale di una vendita*, e non anche come una trasformazione.

## Esempio: `endSale`, gestione dello stato del caso d'uso

### Attenzione!

A volte si vuole ragionare sullo *stato del caso d'uso*.

Ad esempio, si potrebbe voler impedire l'esecuzione dell'operazione `makeCashPayment` fino a che non è stata eseguita l'operazione `endSale`.

In questo caso, l'operazione `endSale` dovrebbe implicare anche una **trasformazione**, per ricordare che è terminato l'inserimento degli articoli della vendita.

Ad esempio: un attributo booleano `itemEntryComplete` di `Sale`. In questo caso, per *Expert*, il controller Register potrebbe richiedere alla `currentSale` di impostare questo attributo a `true`.

Ad esempio: si può utilizzare un oggetto “sessione” per tenere traccia dello stato della sessione del caso d'uso.

Nota: questo viene per ora omesso.

## Esempio: endSale, mostrare il totale

### Scenario principale di successo:

---

1. Il Cliente arriva...
2. Il Cassiere inizia una nuova vendita.
3. Il Cassiere inserisce il codice identificativo di un articolo.
4. Il Sistema registra la riga di vendita per l'articolo e...  
*Il Cassiere ripete i passi 3-4 fino a che non indica che ha terminato.*
5. Il Sistema mostra il totale con le imposte calcolate.

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

Nel passo 5, viene mostrato un totale.

In virtù del principio di separazione Modello-Vista, nella progettazione dello strato del dominio non ci si deve preoccupare di progettare come sarà visualizzato il totale della vendita, ma si deve assicurare che il totale sia noto.

**Si noti che, al momento, nessuna classe di progetto conosce il totale della vendita, per cui è necessario creare un progetto di interazione di oggetti per soddisfare questo requisito.**

## Esempio: endSale, mostrare il totale

### Responsabilità

Chi deve essere responsabile di conoscere il totale della vendita?

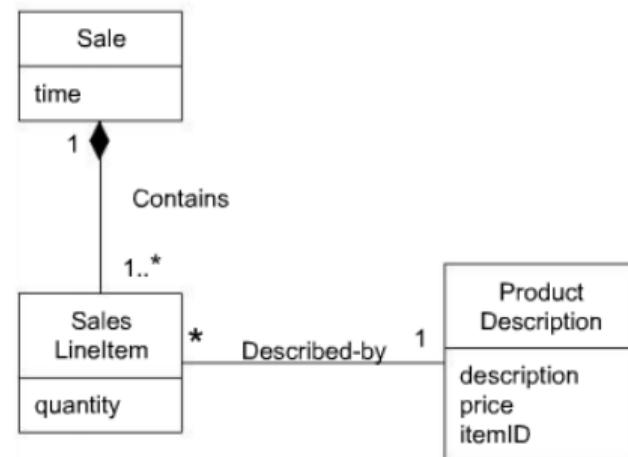
# Esempio: endSale, mostrare il totale

## Responsabilità

Chi deve essere responsabile di conoscere il totale della vendita?

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

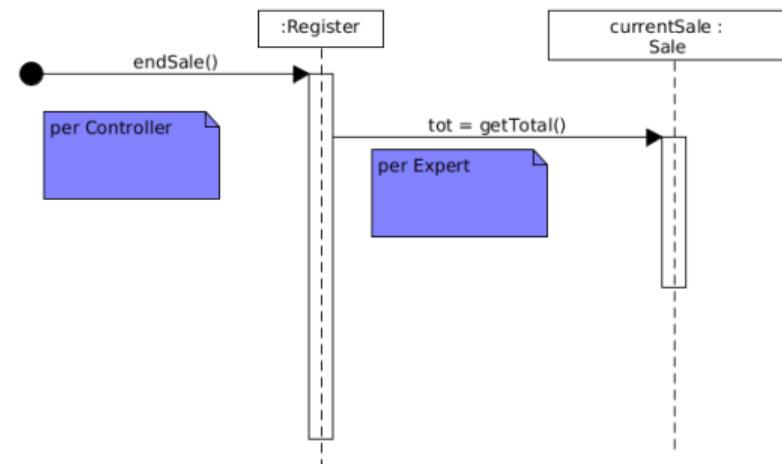
Nota: quello riportato è il Modello di Dominio.



## Esempio: endSale, mostrare il totale

Si assegna la responsabilità a *Sale* di conoscere il suo totale, esprimendo questa responsabilità con un metodo chiamato *getTotal*.

Dal modello di dominio al modello di progetto: un diagramma delle classi con l'indicazione dei metodi (salto rappresentazionale basso).



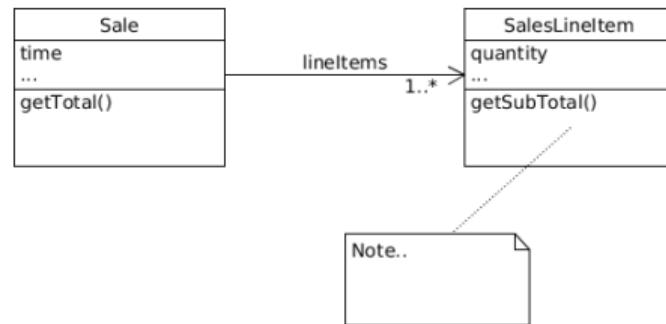
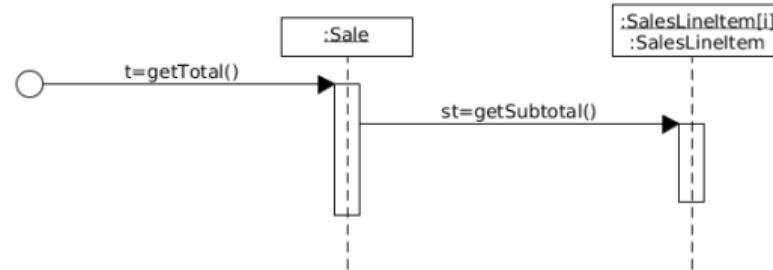
## Esempio: endSale, mostrare il totale

Quali informazioni sono necessarie per determinare il totale parziale per una riga di vendita per un articolo? *SalesLineItem.quantity* e *ProductDescription.price*, ovvero un oggetto *SalesLineItem* conosce la sua *quantità* e la *ProductDescription* ad esso associata, pertanto per Expert, *SalesLineItem* deve determinare il totale parziale, è l'*esperto delle informazioni*.

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

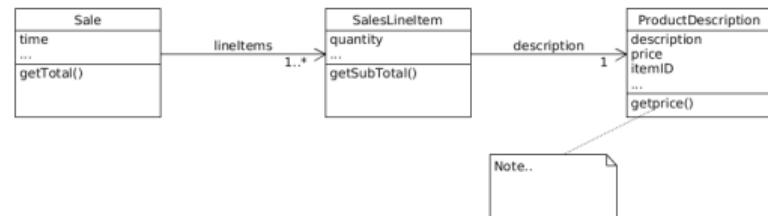
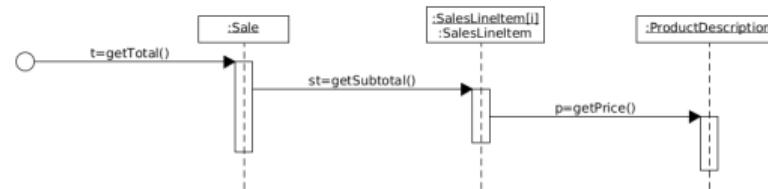
## Esempio: endSale, mostrare il totale

Quindi *Sale* deve inviare messaggi *getSubtotal* a ciascuna delle sue *SalesLineItem* e sommare i risultati.

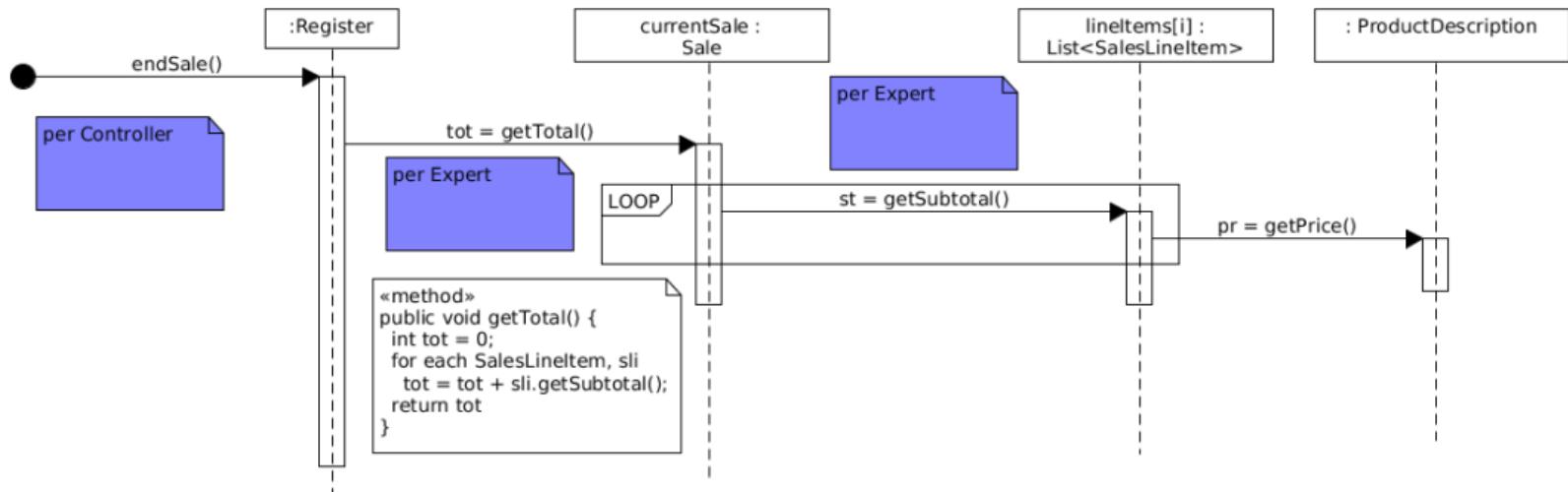


## Esempio: endSale, mostrare il totale

Quindi *Sale* deve inviare messaggi *getSubtotal* a ciascuna delle sue *SalesLineItem* e sommare i risultati. Per soddisfare la sua responsabilità, *SalesLineItem* deve conoscere il prezzo del prodotto a cui si riferisce. *SalesLineItem* invia a *ProductDescription* un messaggio *getPrice* chiedendogli il prezzo del prodotto.



## Esempio: endSale, diagramma di interazione



## Esempio: makeCashPayment

L'operazione di sistema *makeCashPayment* avviene quando un cassiere inserisce l'importo in contanti offerto dal cliente per il pagamento.

### Contratto CO4: makeCashPayment

---

**Operazione:** makeCashPayment( amount: Money )

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

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

**Post-condizioni:** – è stata creata un'istanza p di CashPayment.

– p è stata associata con la Sale corrente s.

– p.amountTendered è diventato amount.

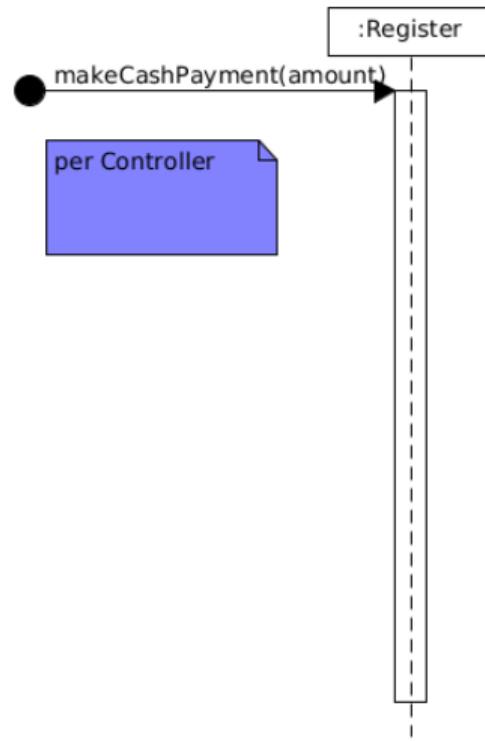
– la Sale corrente s è stata associata con lo Store (s è stata aggiunta al registro storico delle vendite completate).

## Esempio: makeCashPayment, controller

---

Il pattern *Controller* suggerisce di utilizzare la stessa classe controller per tutte le operazioni di sistema di un caso d'uso; pertanto si continuerà a utilizzare *Register* come controller per l'operazione *makeCashPayment* e per le ulteriori operazioni di sistema del caso d'uso *Elabora Vendita*.

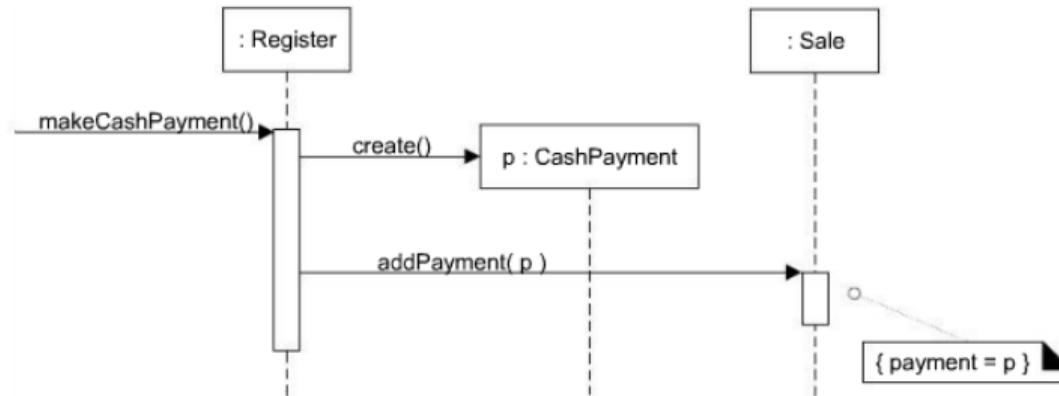
## Esempio: makeCashPayment, diagramma di interazione



Esempio: `makeCashPayment`, prima, seconda e terza post-condizione “è stata creata un’istanza p di `CashPayment`”, “p è stata associata con la Sale corrente s” e “`p.amountTendered` è diventato `amount`”

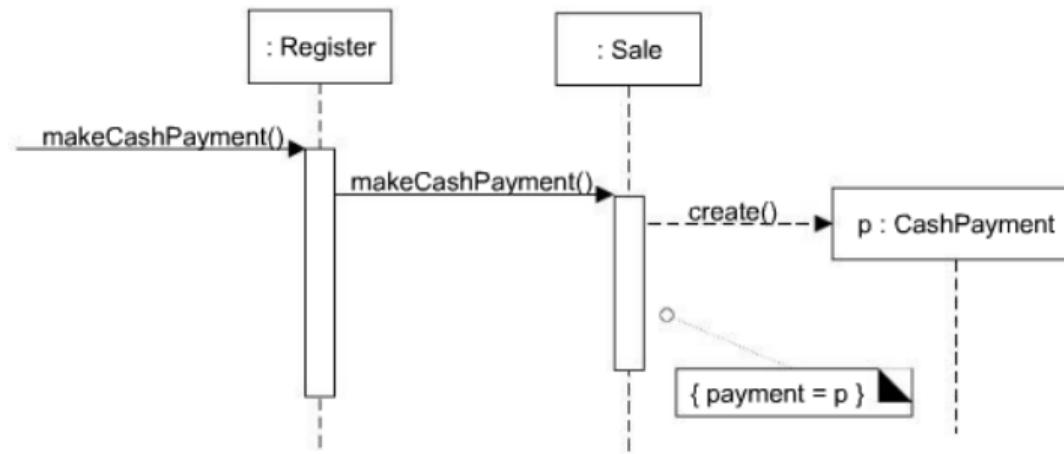
### Prima soluzione:

- con il pattern Creator si sceglie *Register* come creatore di *Payment*, suggerito dalle responsabilità nel “mondo reale” (registra i pagamenti)
- dunque uso metodo `addPayment(p)` per comunicare con *Sale*



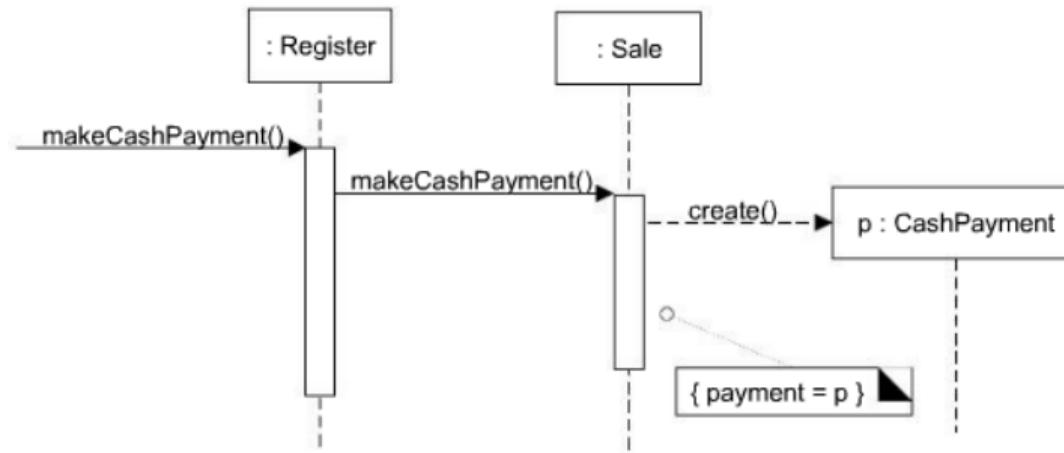
Esempio: `makeCashPayment`, prima, seconda e terza post-condizione “è stata creata un’istanza p di `CashPayment`”, “p è stata associata con la Sale corrente s” e “`p.amountTendered` è diventato `amount`”

**Seconda soluzione:** per Expert, *Sale* conosce *Payment*, la creazione di *CashPayment* la può fare la *Sale*.



Esempio: `makeCashPayment`, prima, seconda e terza post-condizione “è stata creata un’istanza p di `CashPayment`”, “p è stata associata con la Sale corrente s” e “`p.amountTendered` è diventato `amount`”

**Seconda soluzione:** per Expert, *Sale* conosce *Payment*, la creazione di *CashPayment* la può fare la *Sale*.



Esempio: `makeCashPayment`, prima, seconda e terza post-condizione “è stata creata un’istanza `p` di `CashPayment`”, “`p` è stata associata con la Sale corrente `s`” e “`p.amountTendered` è diventato `amount`”

---

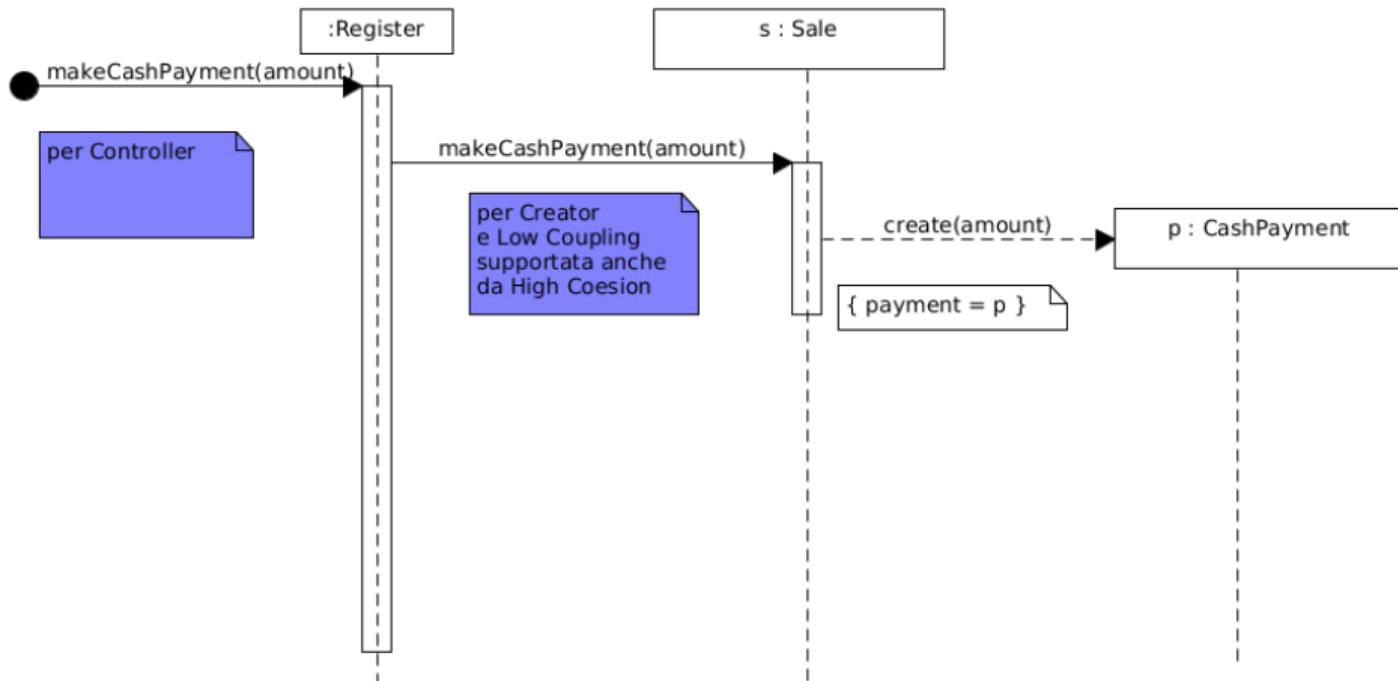
### Valutazione con Low Coupling:

- Per Low Coupling la seconda soluzione va preferita poiché mantiene un accoppiamento complessivo più basso (si veda discussione su *Low Coupling*)

### Valutazione con High Cohesion:

- La prima soluzione significa che il `Register` si assume non solo la responsabilità di ricevere l’operazione di sistema `makeCashPayment`, ma anche parte della responsabilità di soddisfarla
- Se si continua a rendere la classe **Register** responsabile di eseguire una parte del lavoro o l’intero lavoro relativo a sempre più operazione di sistema, essa diventerà sempre più carica di compiti, e diventerà **non coesa**
- Per High Cohesion la seconda soluzione va preferita

# Esempio: makeCashPayment, diagramma di interazione



**Esempio: makeCashPayment, quarta post-condizione “la Sale corrente s’è stata associata con lo Store (s’è stata aggiunta al registro storico delle vendite complete)”**

---

I requisiti affermano che la vendita deve essere inserita in un registro (log) storico delle vendite complete.

Per Expert e l’analisi del Modello di Dominio suggerisce che lo *Store* conosca e registri le *Sale* complete.

Un’alternativa è l’introduzione di un *SalesLedger*, un libro mastro (nuovo concetto).

**Esempio: makeCashPayment, quarta post-condizione “la Sale corrente s’è stata associata con lo Store (s’è stata aggiunta al registro storico delle vendite complete)”**

---

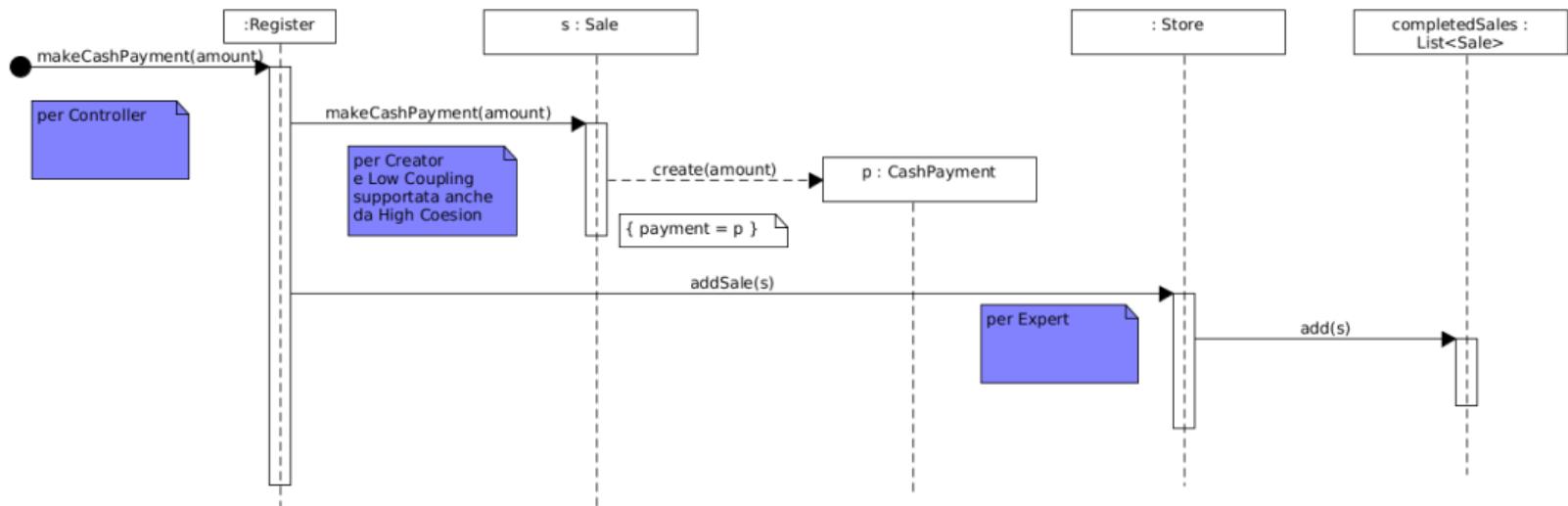
I requisiti affermano che la vendita deve essere inserita in un registro (log) storico delle vendite complete.

Per Expert e l’analisi del Modello di Dominio suggerisce che lo *Store* conosca e registri le *Sale* complete.

Un’alternativa è l’introduzione di un *SalesLedger*, un libro mastro (nuovo concetto).

**Valutazione con High Cohesion:** la scelta di *Store* per questa responsabilità è accettabile se *Store* ha poche responsabilità, mentre l’utilizzo di un oggetto *SalesLedger* ha senso man mano che il progetto cresce e *Store* diventa poco coeso.

# Esempio: makeCashPayment, diagramma di interazione



## Esempio: makeCashPayment, calcolo del resto

In virtù del principio di Separazione Modello-Vista, non ci si dovrebbe preoccupare del modo in cui il resto sia visualizzato o stampato, **ma ci si deve assicurare che sia conosciuto.**

Al momento nessuna classe conosce il resto, per cui è necessario creare un progetto di interazioni di oggetti che soddisfi questo requisito.

### Responsabilità

Chi è responsabile di conoscere il resto?

## Esempio: makeCashPayment, calcolo del resto

In virtù del principio di Separazione Modello-Vista, non ci si dovrebbe preoccupare del modo in cui il resto sia visualizzato o stampato, **ma ci si deve assicurare che sia conosciuto.**

Al momento nessuna classe conosce il resto, per cui è necessario creare un progetto di interazioni di oggetti che soddisfi questo requisito.

### Responsabilità

Chi è responsabile di conoscere il resto?

Per calcolare il resto è necessario conoscere il totale della vendita e l'importo in contanti offerto.

Per Expert, *Sale* e *CashPayment* sono esperti parziali (rispettivamente esperto del totale della vendita ed esperto dell'importo in contanti offerto).

Esempio: `makeCashPayment`, prima, seconda e terza post-condizione “è stata creata un’istanza `p` di `CashPayment`”, “`p` è stata associata con la `Sale` corrente `s`” e “`p.amountTendered` è diventato `amount`”

### Valutazione con Low Coupling:

- Se il `CashPayment` è il responsabile principale per conoscere il resto, necessita della visibilità nei confronti della `Sale`, per chiedere alla `Sale` il suo totale. Poiché al momento non conosce la `Sale`, aumenterebbe l’accoppiamento complessivo
- Se la `Sale` è il responsabile principale per conoscere il resto, necessita della visibilità nei confronti del `CashPayment`, per chiedergli l’importo in contanti consegnato. Dal momento che la `Sale` ha già la visibilità nei confronti del `CashPayment` (era il suo creatore, si vedano i lucidi precedenti), questo approccio non aumenta l’accoppiamento complessivo.<sup>1</sup>

---

<sup>1</sup>Per determinare gli accoppiamenti si contano le dipendenze che sono direzionali.

Esempio: `makeCashPayment`, prima, seconda e terza post-condizione “è stata creata un’istanza `p` di `CashPayment`”, “`p` è stata associata con la `Sale` corrente `s`” e “`p.amountTendered` è diventato `amount`”

### Valutazione con Low Coupling:

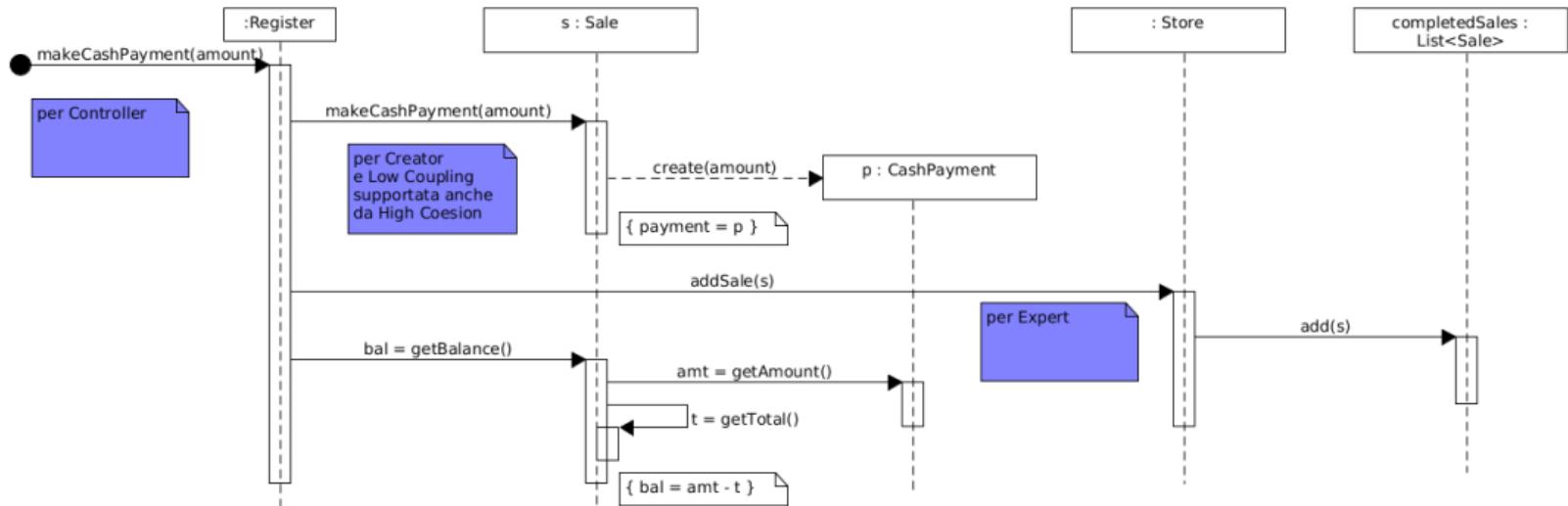
- Se il `CashPayment` è il responsabile principale per conoscere il resto, necessita della visibilità nei confronti della `Sale`, per chiedere alla `Sale` il suo totale. Poiché al momento non conosce la `Sale`, aumenterebbe l’accoppiamento complessivo
- Se la `Sale` è il responsabile principale per conoscere il resto, necessita della visibilità nei confronti del `CashPayment`, per chiedergli l’importo in contanti consegnato. Dal momento che la `Sale` ha già la visibilità nei confronti del `CashPayment` (era il suo creatore, si vedano i lucidi precedenti), questo approccio non aumenta l’accoppiamento complessivo.<sup>1</sup>

### Il responsabile è quindi `Sale`.

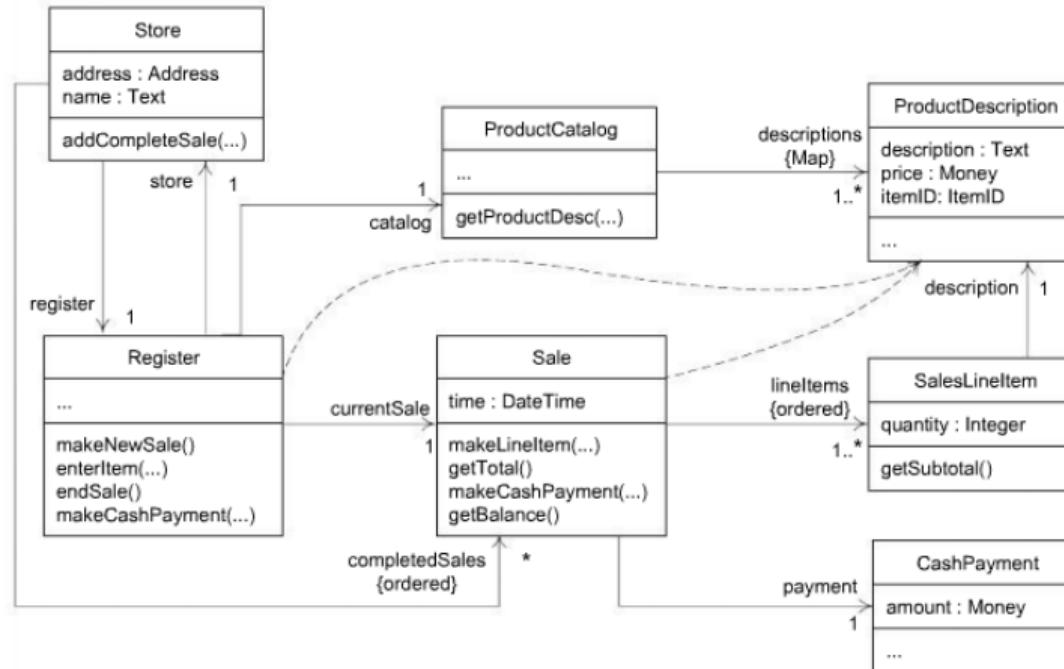
---

<sup>1</sup>Per determinare gli accoppiamenti si contano le dipendenze che sono direzionali.

# Esempio: makeCashPayment, diagramma di interazione



# Esempio: DCD finale di NextGen per l'iterazione 1, strato di dominio



## **Collegare lo strato UI allo strato del dominio e inizializzazione**

---

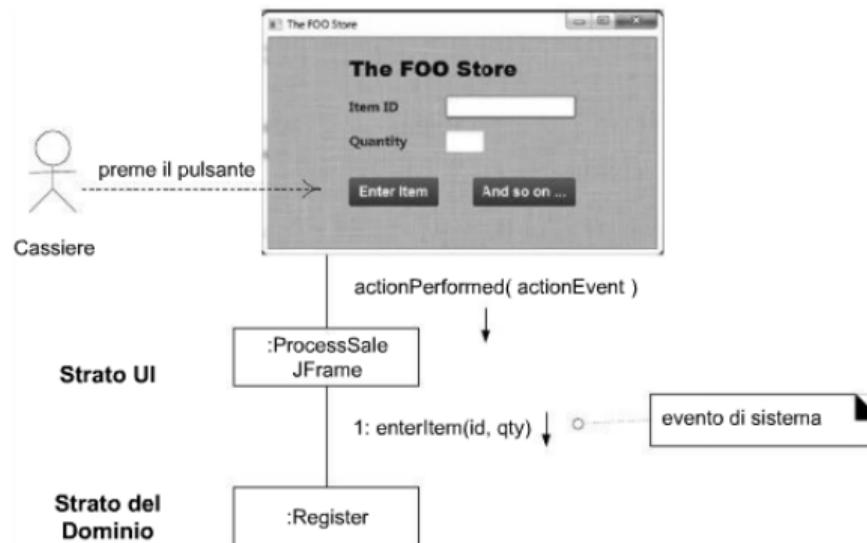
## Collegare lo strato UI allo strato di dominio

Scelte comuni per dare agli oggetti dello strato UI la visibilità nei confronti degli oggetti dello strato di dominio sono:

- Un oggetto inizializzatore chiamato dal metodo iniziale dell'applicazione che crea sia un oggetto UI che un oggetto di dominio e passa l'oggetto di dominio all'oggetto UI
- Un oggetto UI che recupera l'oggetto di dominio da una sorgente nota, come un oggetto factory responsabile della creazione di oggetti di dominio

# Collegare lo strato UI allo strato di dominio

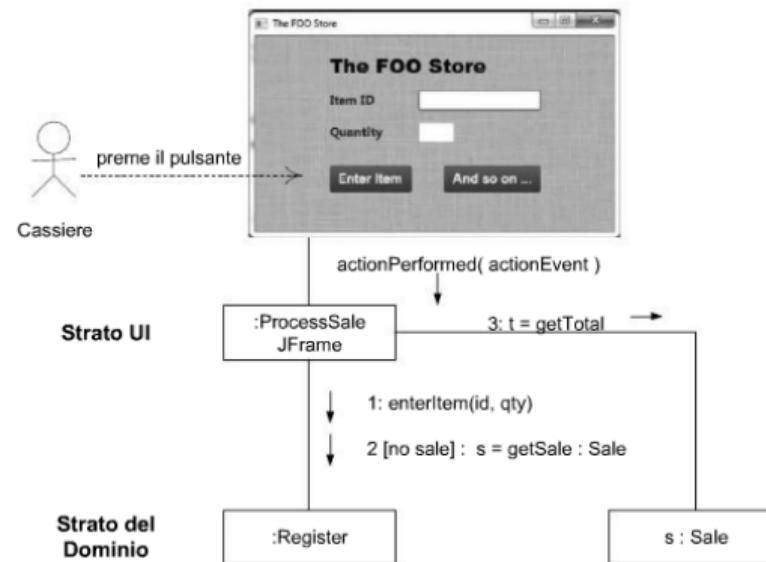
Nel nostro esempio, una volta che l'oggetto UI ha una connessione all'istanza di Register, può inoltrare ad essa i messaggi per gli eventi di sistema.



# Collegare lo strato UI allo strato di dominio

Per visualizzare il totale corrente (o altra informazione relativa alla vendita): un oggetto UI chiede il riferimento all'oggetto Sale corrente, e invia direttamente i messaggi alla Sale.

Alternativa, si passa da Register.



## Caso d'uso d'avviamento o inizializzazione

La maggior parte dei sistemi hanno un caso d'uso di Avviamento (Start Up),隐式的 oppure esplicito, e alcune operazioni di sistema iniziali relative all'avvio dell'applicazione.

### Importante

La progettazione dell'inizializzazione va eseguita per ultima

Un idioma di progettazione comune è quello di creare un **oggetto di dominio iniziale** o un insieme di oggetti di dominio iniziali di pari grado che sono i primi oggetti software di "dominio" creati.

## Caso d'uso d'avviamento o inizializzazione

Questa creazione può avvenire in modo esplicito nel metodo iniziale *main* o in un oggetto *Factory* chiamato dal metodo *main*.

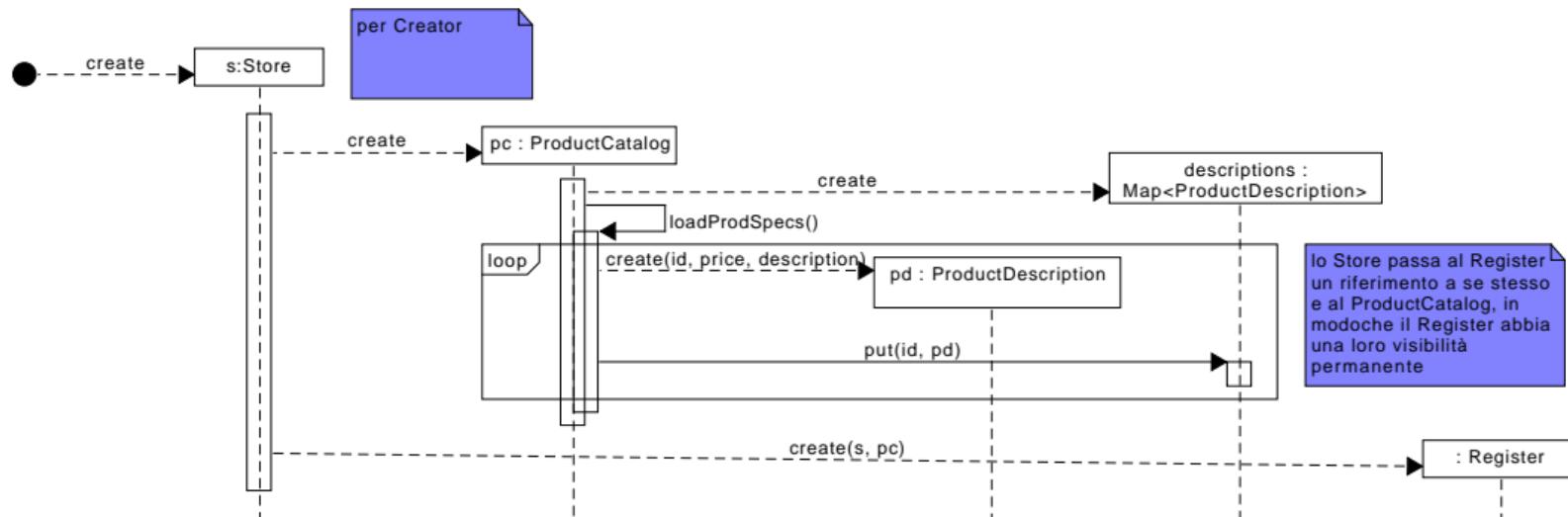
```
public class Main {  
    public static void main( String[] args ) {  
  
        /* Store è l'oggetto di dominio iniziale.  
         * Lo Store crea degli altri oggetti di dominio. */  
  
        Store store = new Store();  
  
        Register register = store.getRegister();  
  
        ProcessSaleJFrame frame = new ProcessSaleJFrame( register );  
        ...  
    }  
}
```

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

Nota: *register* è il controller GRASP.

# Caso d'uso d'avviamento o inizializzazione

Esempio per Elabora Vendita. Si è scelto Store come l'oggetto iniziale.



# 11 . Design Pattern GoF

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

### **Si noti che**

questi lucidi sono basati sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016” e sul documento “F. Guidi Polanco, *GoF's Design Patterns in Java*, Politecnico di Torino, 2002, disponibile all’indirizzo:

<http://eii.pucv.cl/pers/guidi/designpatterns.htm>.

# Table of contents

1. Creazionali: Abstract Factory e Singleton
2. Strutturali: Adapter, Composite e Decorator
3. Comportamentali: Observer, State, Strategy e Visitor

# I design pattern GoF

(Dai lucidi: *10 . General Responsibility Assignment Software Patterns, GRASP*)

La nozione di pattern ebbe origine con i *pattern architettonici* (di costruzione) di *Christopher Alexander*.

I pattern per il software ebbero origine negli anni ottanta con *Ken Beck* (famoso anche per *Extreme Programming*) che riconobbe il lavoro svolto da Alexander nell'architettura, e furono sviluppati da Beck con *Ward Cunningham*.

Nel 1994 viene pubblicato il libro *Design Pattern* di *Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides* che descrive 23 *pattern* per la *programmazione OO*. Questi sono diventati noti come design pattern **GoF** (**Gang-of-Four**, “*banda dei quattro*” ).

# I design pattern GoF

I GoF sono più degli “schemi di progettazione avanzata” che “principi” (come nel caso di GRASP).

- Ciascun design pattern descrive una soluzione progettuale comune a un problema di progettazione ricorrente
- I design pattern GoF sono classificati in base al loro scopo:
  - **creazionale**
  - **strutturale**
  - **comportamentale**

# I design pattern GoF creazionali

Risolvono problematiche inerenti l'istanziazione degli oggetti:

- **Abstract Factory**
- **Builder**
- **Factory Method**
- **Lazy Initialization**
- **Prototype Pattern**
- **Singleton**
- **Double-check Locking**

# I design pattern GoF creazionali

Risolvono problematiche inerenti l'istanziazione degli oggetti:

- **Abstract Factory**
- **Builder**
- **Factory Method**
- **Lazy Initialization**
- **Prototype Pattern**
- **Singleton**
- **Double-check Locking**

# I design pattern GoF strutturali

Risolvono problematiche inerenti la struttura delle classi e degli oggetti:

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

# I design pattern GoF strutturali

Risolvono problematiche inerenti la struttura delle classi e degli oggetti:

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

# I design pattern GoF comportamentali

Forniscono soluzione alle più comuni tipologie di interazione tra gli oggetti:

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

# I design pattern GoF comportamentali

Forniscono soluzione alle più comuni tipologie di interazione tra gli oggetti:

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

# Favorire composizione rispetto ereditarietà

**GoF: preferire la composizione rispetto all'ereditarietà tra classi!**

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

## Favorire composizione rispetto ereditarietà

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

- Ereditarietà di classi:
  - Definita staticamente, non è possibile cambiarla a tempo di esecuzione: se una classe estende un'altra, questa relazione è definita nel codice sorgente, non può cambiare a runtime
  - Una modifica alla sopraclasse potrebbe avere ripercussioni indesiderate sul funzionamento di una classe che la estende (**non rispetta l'incapsulamento**)<sup>1</sup>
- Composizione di oggetti:
  - Se una classe usa un'altra classe, questa potrebbe essere referenziata attraverso una interfaccia, a runtime potrebbe esserci una qualsiasi altra classe che implementa l'interfaccia
  - La composizione attraverso un'interfaccia **rispetta l'incapsulamento**, solo una modifica all'interfaccia comporterebbe ripercussioni

---

<sup>1</sup><http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.8552&rep=rep1&type=pdf>

## Favorire composizione rispetto ereditarietà

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

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

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

### In breve...

Il riuso è meglio ottenerlo attraverso il meccanismo di delega piuttosto che attraverso il meccanismo di ereditarietà, almeno per quanto riguarda l'utilizzo della specializzazione.

## **Creazionali: Abstract Factory e Singleton**

---

# Abstract Factory

## Abstract Factory

**Nome:** Abstract Factory

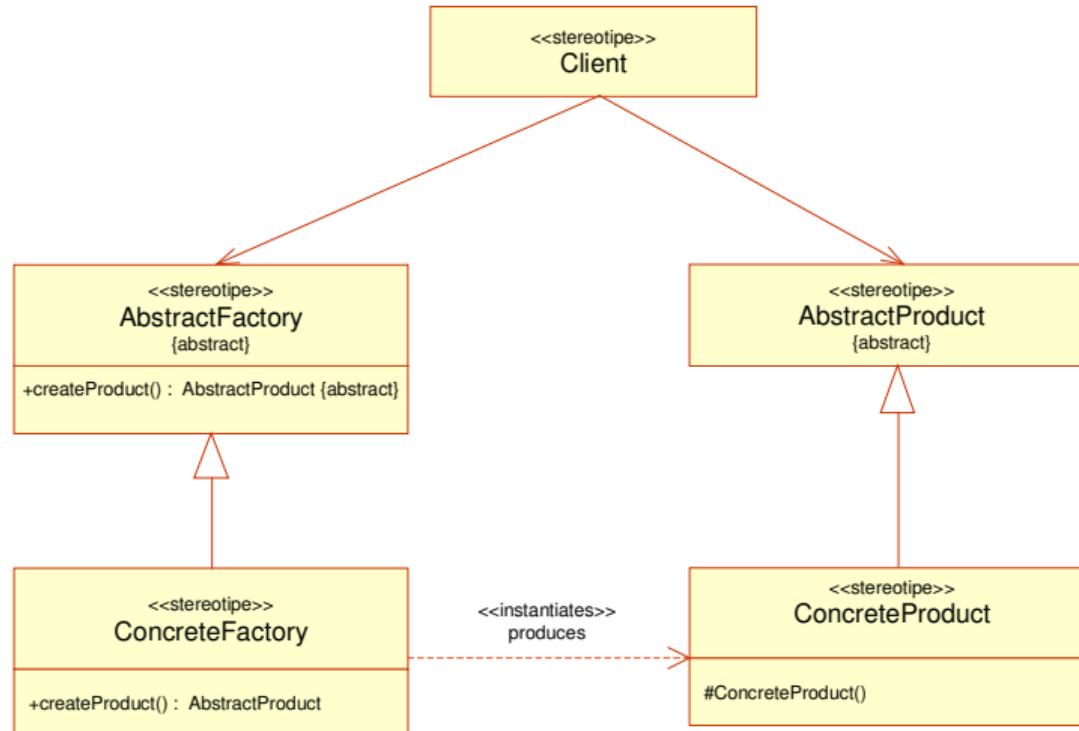
**Problema:** Come creare famiglie di classi correlate che implementano un'interfaccia comune?

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

# Abstract Factory

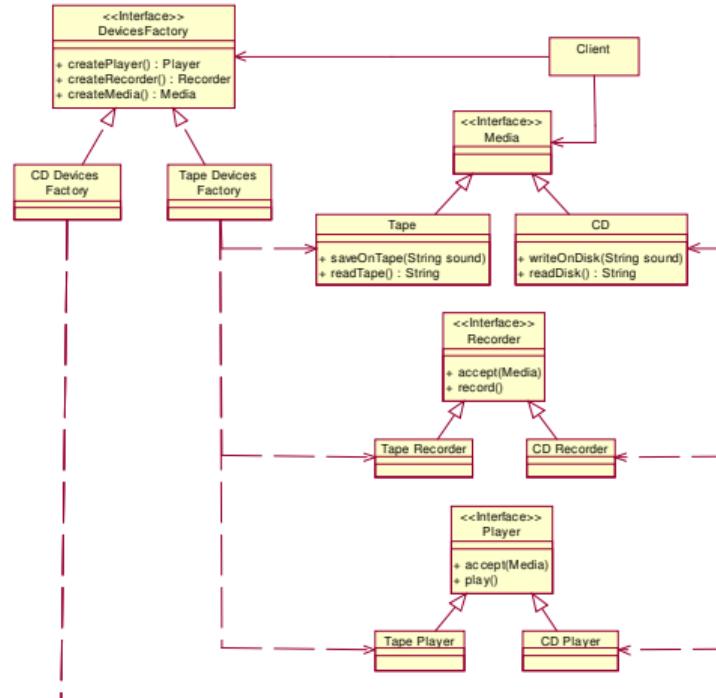
- Presenta un'interfaccia per la creazione di famiglie di prodotti, in modo tale che il cliente che li utilizza non abbia conoscenza delle loro concrete classi. Questo consente:
  - di assicurarsi che il cliente crei soltanto prodotti vincolati fra di loro
  - l'utilizzo di diverse famiglie di prodotti da parte dello stesso cliente
- Una variante comune di Abstract Factory consiste nel creare una classe astratta factory a cui si accede utilizzando il pattern Singleton
- È usata nelle librerie Java per la creazione di famiglie di elementi GUI per diversi sistemi operativi e sottosistemi GUI

# Struttura del pattern Abstract Factory



# Applicazione del pattern Abstract Factory

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 6–12.



## Singleton

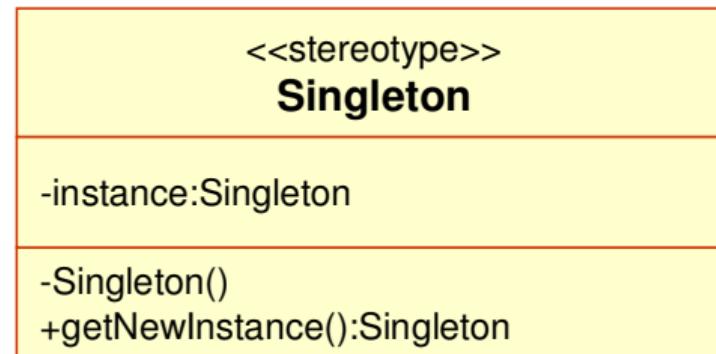
**Nome:** Singleton

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

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

- Il “Singleton” pattern definisce una classe della quale è possibile la istanziazione di un unico oggetto, tramite l’invocazione a un metodo della classe, incaricato della produzione degli oggetti
- Le diverse richieste di istanziazione comportano la restituzione di un riferimento allo stesso oggetto
- In UML un singleton viene illustrato con un “1” nella sezione del nome, in alto a destra

# Struttura del pattern Singleton



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

Tre diverse implementazioni in Java:

- Singleton come classe statica: non è un vero e proprio Singleton, si lavora con la classe statica, non un oggetto. Questa classe statica ha metodi statici che offrono i servizi richiesti
- Singleton creato da un metodo statico: una classe che ha un metodo statico che deve essere chiamato per restituire l'istanza del Singleton. L'oggetto verrà istanziato **solo la prima volta**. Le successive sarà restituito un riferimento allo stesso oggetto (inizializzazione pigra<sup>2</sup>)
- Singleton multi-thread: versione multi-thread della soluzione precedente

---

<sup>2</sup>Contrapposta all'inizializzazione golosa: l'oggetto verrà istanziato quando la classe è caricata. È preferibile quella pigra perché se non si accede mai all'istanza viene evitato il lavoro della creazione. L'inizializzazione potrebbe essere complessa.

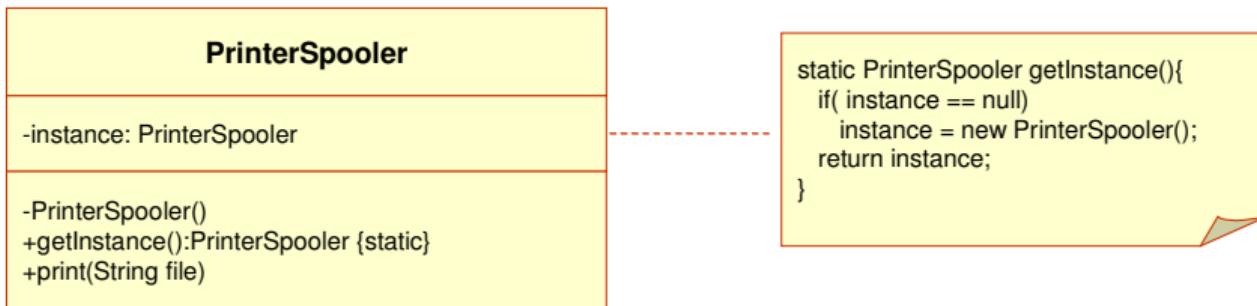


Il *Singleton creato da un metodo statico* è preferibile rispetto al *Singleton come classe statica* per tre motivi:

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

# Applicazione del pattern Singleton

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 38–41.



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

## Singleton: si ponga attenzione a

- Presenza di Singleton in virtual machine multiple
- Singleton caricati contemporaneamente da diversi class loader
- Singleton distrutti dal garbage collector e dopo caricati quando sono necessari
- Presenza di istanze multiple come sottoclassi di un Singleton
- Copia di Singleton come risultato di un doppio processo di deserializzazione

## **Strutturali: Adapter, Composite e Decorator**

---

## Adapter

**Nome:** Adapter

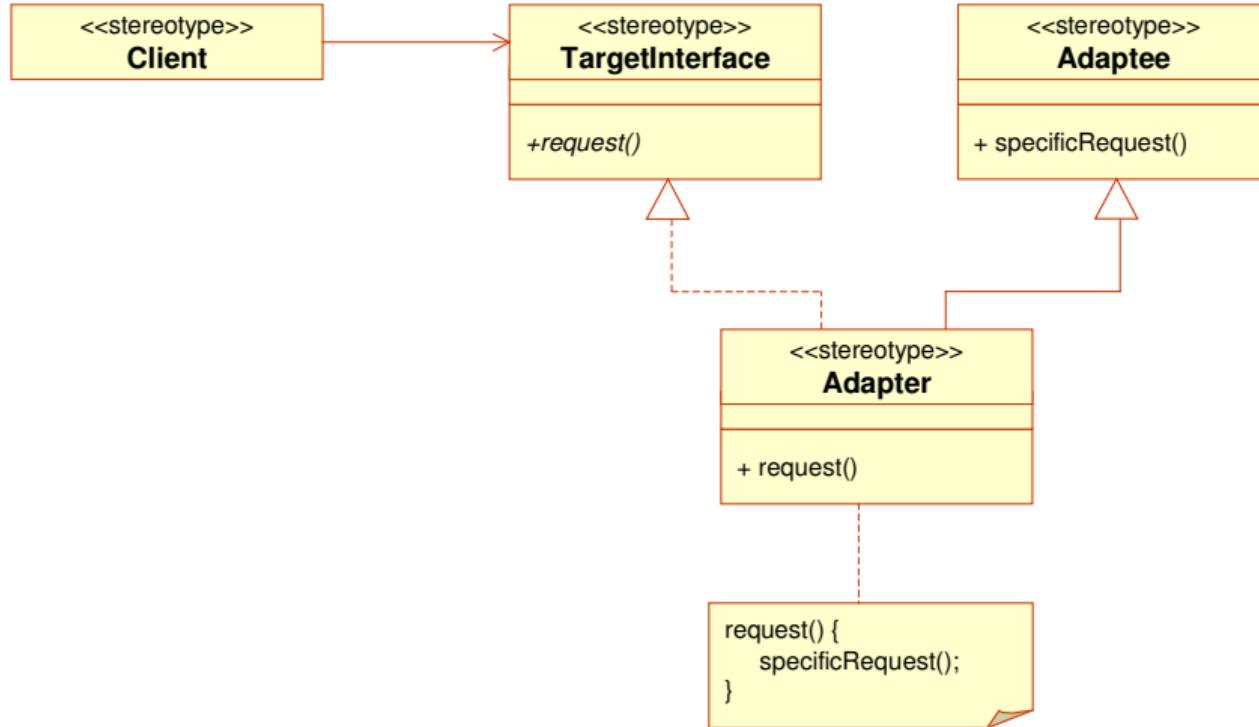
**Problema:** Come gestire interfacce incompatibili, o fornire un'interfaccia stabile a comportamenti simili ma con interfacce diverse?

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

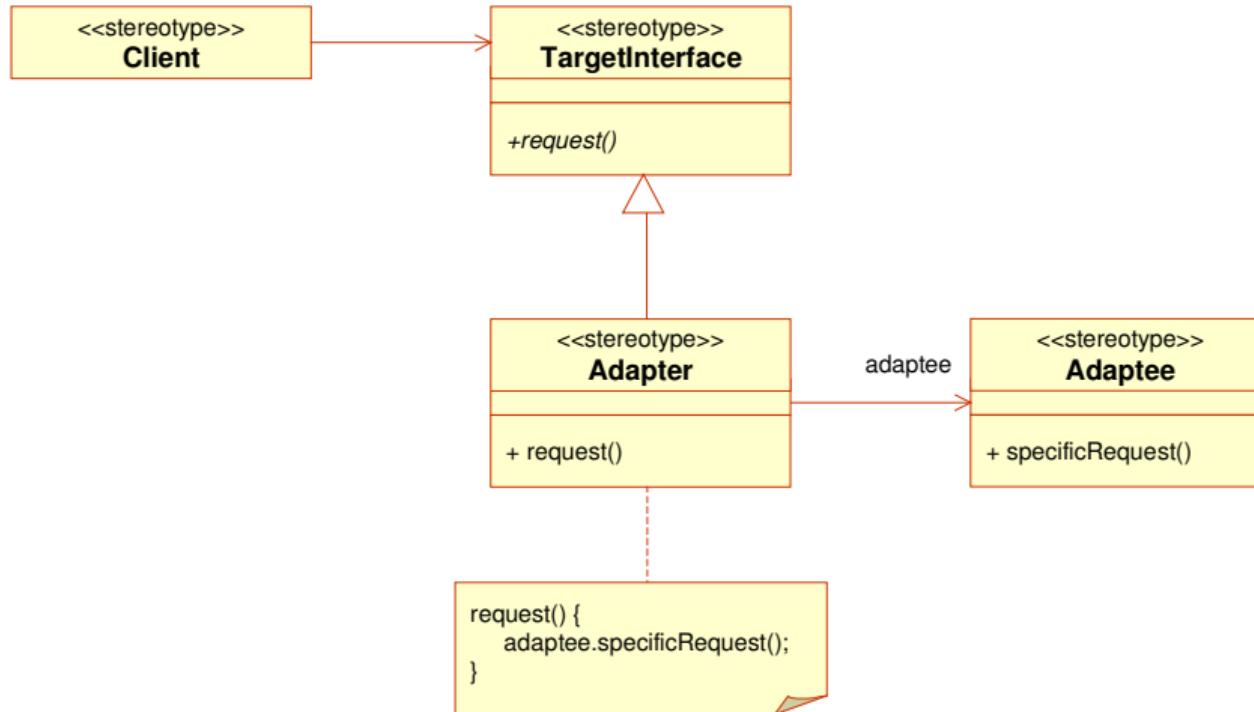
- Si consideri una coppia di oggetti software in una relazione client-server. Si parla di interfacce incompatibili quando l'oggetto server offre servizi di interesse per l'oggetto client ma l'oggetto client vuole fruire di questi servizi in una modalità diversa da quella prevista dall'oggetto server (**interfacce incompatibili**)
- Ci sono più oggetti server che offrono servizi simili; questi oggetti hanno interfacce simili ma diverse tra loro. Un oggetto client vuole fruire dei servizi offerti da uno tra questi oggetti server (**componenti simili con interfacce diverse**)

- In generale, un adattatore riceve richieste dai suoi client, per esempio da un oggetto dello strato di dominio, nel formato client dell'adattatore
- L'adattatore poi adatta, trasforma, una richiesta ricevuta in una richiesta nel formato del server
- L'adattatore invia la richiesta al server
- Se il server fornisce una risposta, lo fa nel formato del server
- L'adattatore adatta, trasforma, la risposta ricevuta dal server in una risposta nel formato del client e poi la restituisce al suo client

# Struttura del pattern Adapter (Class)

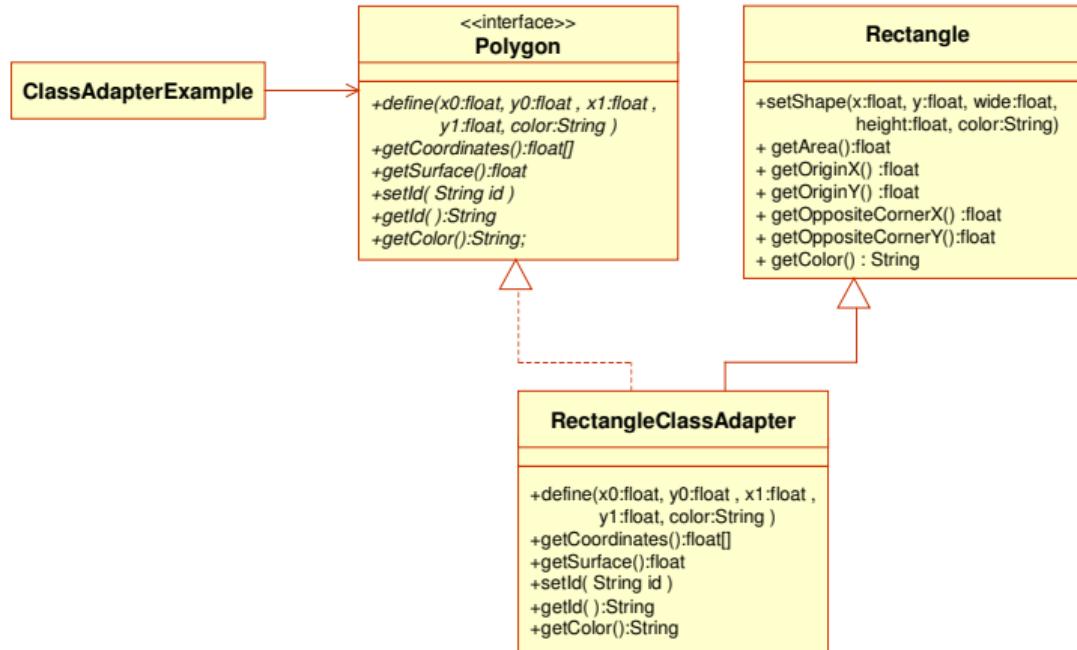


# Struttura del pattern Adapter (Object)



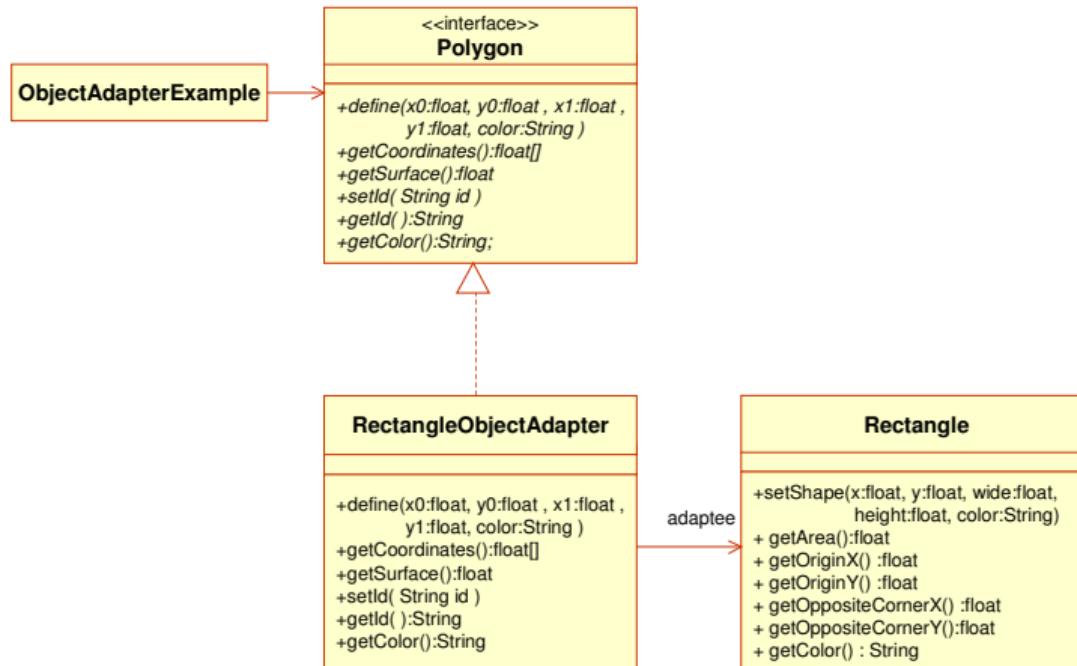
# Applicazione del pattern Adapter (Class)

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 42–49.



# Applicazione del pattern Adapter (Object)

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 42–49.



## Composite

**Nome:** Composite

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

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

# Composite

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

Questo pattern è utile nei casi in cui si vuole:

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

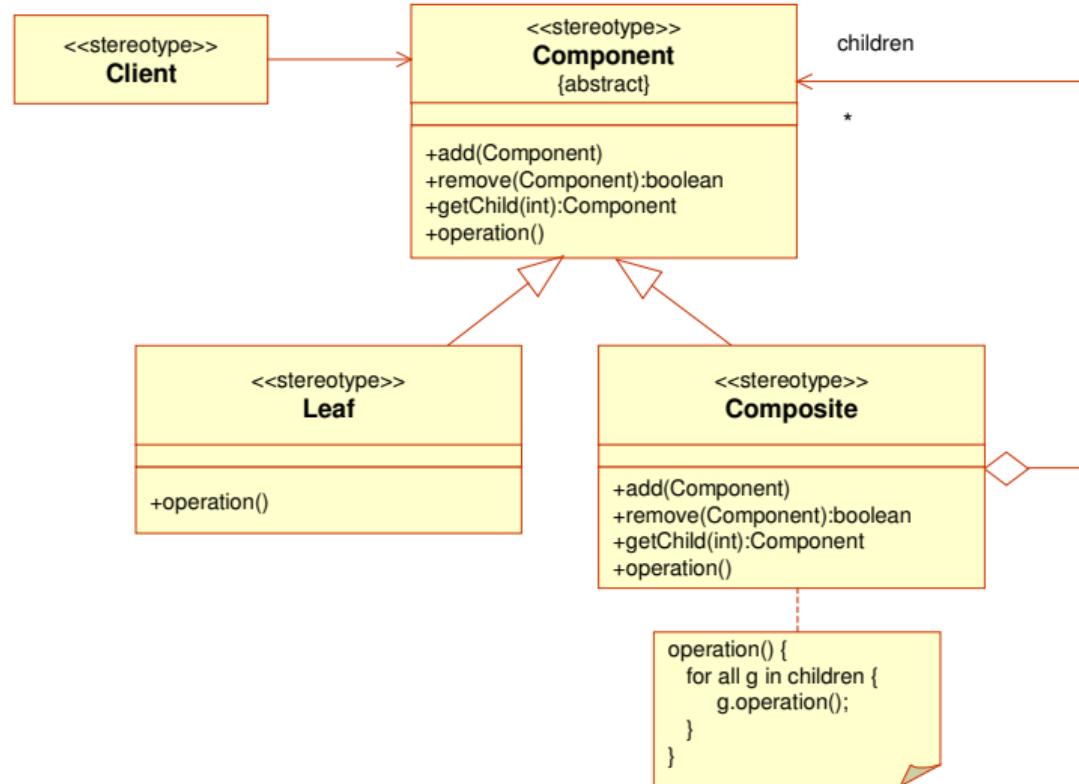
# Composite

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

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

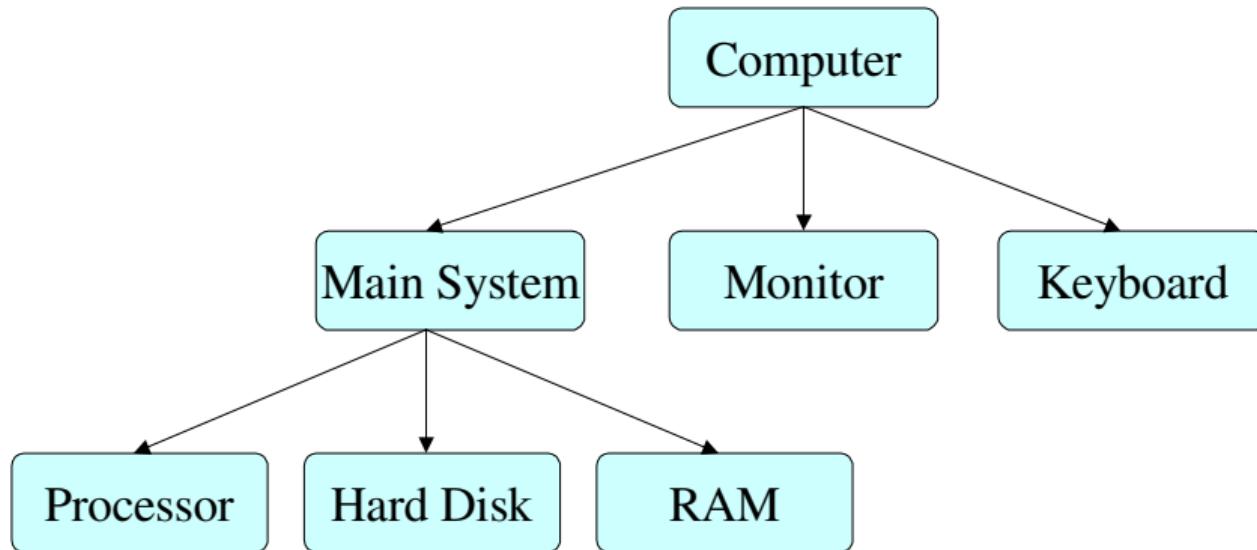
Nota: i componenti composti possono immagazzinare sia componenti singoli sia altri contenitori.

# Struttura del pattern Composite



# Applicazione del pattern Composite

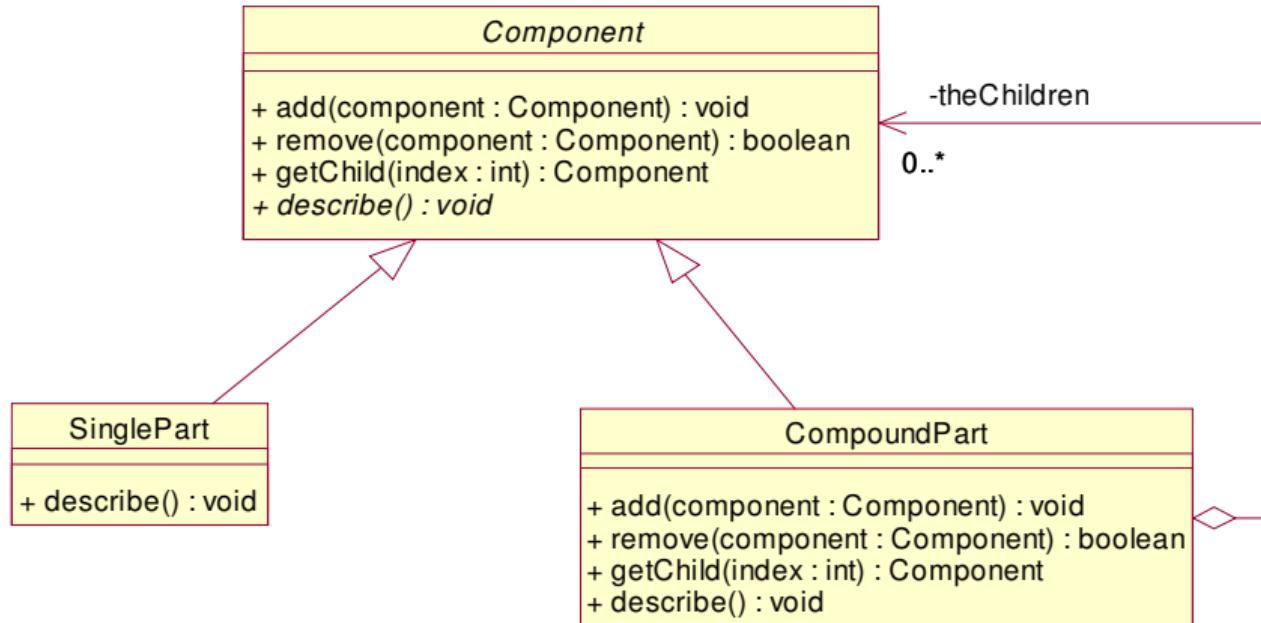
Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 57–62.



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

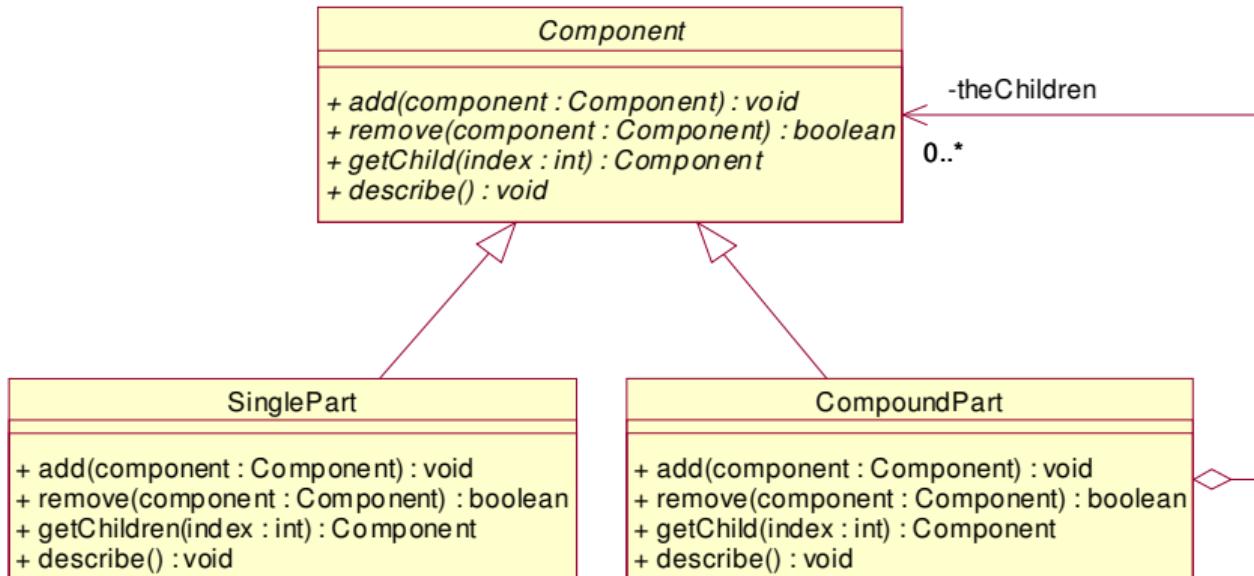
# Applicazione del pattern Composite

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 57–62.

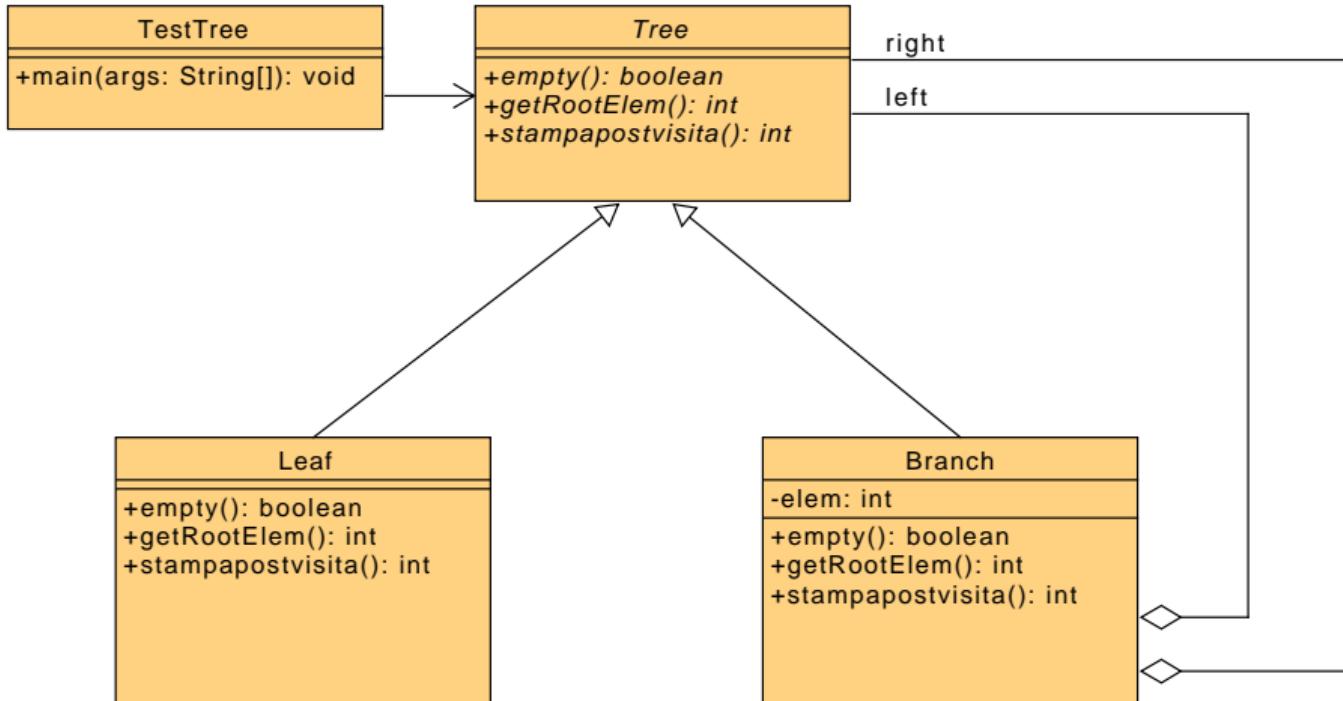


# Applicazione del pattern Composite (abstract component)

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 62–64.



# Applicazione del pattern Composite: Tree



## Applicazione del pattern Composite: Tree

```
1  public abstract class Tree {  
2      public abstract boolean empty();  
3      public abstract int getRootElement();  
4      public abstract void stampaPostvisita();  
5  }  
6  
7  public class TestTree {  
8      public static void main(String[] args) {  
9          ...  
10         System.out.println("Stampo albero postvisita");  
11         t.stampapostvisita();  
12         System.out.println("");  
13     }  
14 }
```

## Applicazione del pattern Composite: Tree

```
1  public class Leaf extends Tree {  
2      public Leaf() { }  
3      public boolean empty() { return true; }  
4      public int getRootElement() {  
5          assert false;  
6          return 0;  
7      }  
8      public void stampaPostvisita() { }  
9  }
```

## Applicazione del pattern Composite: Tree

```
1  public class Branch extends Tree {  
2      private int elem;  
3      private Tree left;  
4      private Tree right;  
5      public Branch(int elem, Tree left,  
6                      Tree right) {  
7          this.elem = elem;  
8          this.left = left;  
9          this.right = right;  
10     }  
11     public boolean empty() { return false; }  
12     public int getRootElem() { return elem; }  
13     public void stampapostvisita() {  
14         right.stampapostvisita();  
15         left.stampapostvisita();  
16         System.out.print(this.getRootElem() + " ");  
17     }  
18 }
```

## In breve

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

## Decorator

**Nome:** Decorator

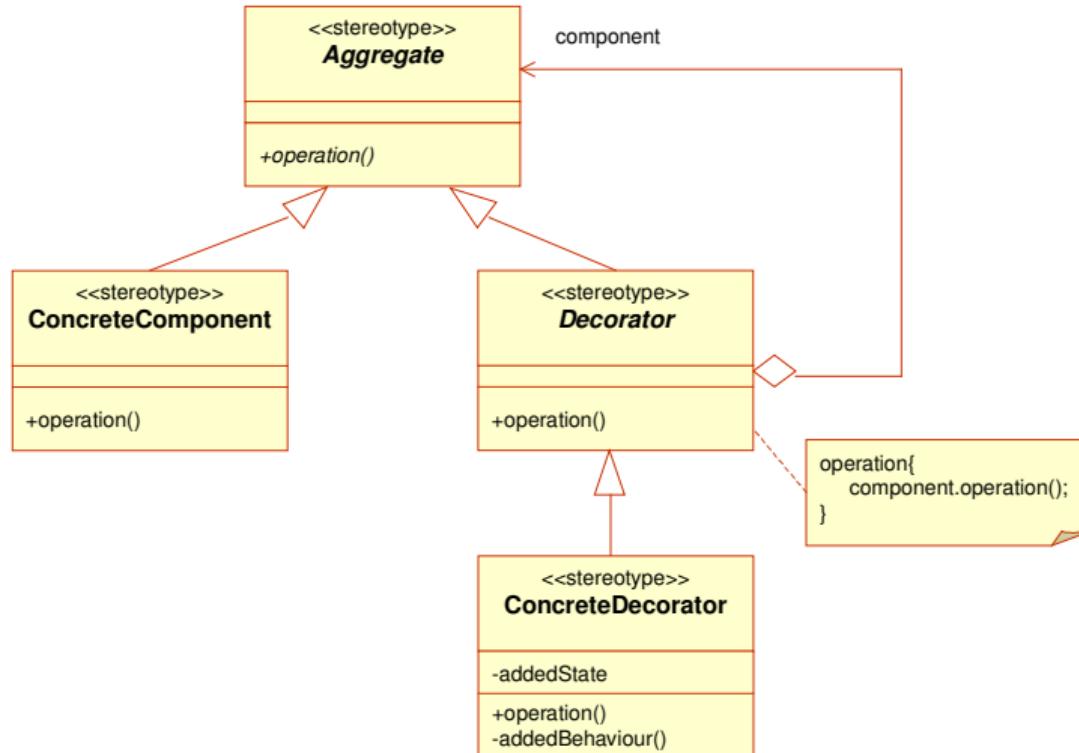
**Problema:** Come permettere di assegnare una o più responsabilità addizionali ad un oggetto in maniera dinamica ed evitare il problema della relazione statica? Come provvedere una alternativa più flessibile al meccanismo di sottoclasse ed evitare il problema di avere una gerarchia di classi complessa?

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

Noto anche come *Wrapper*.

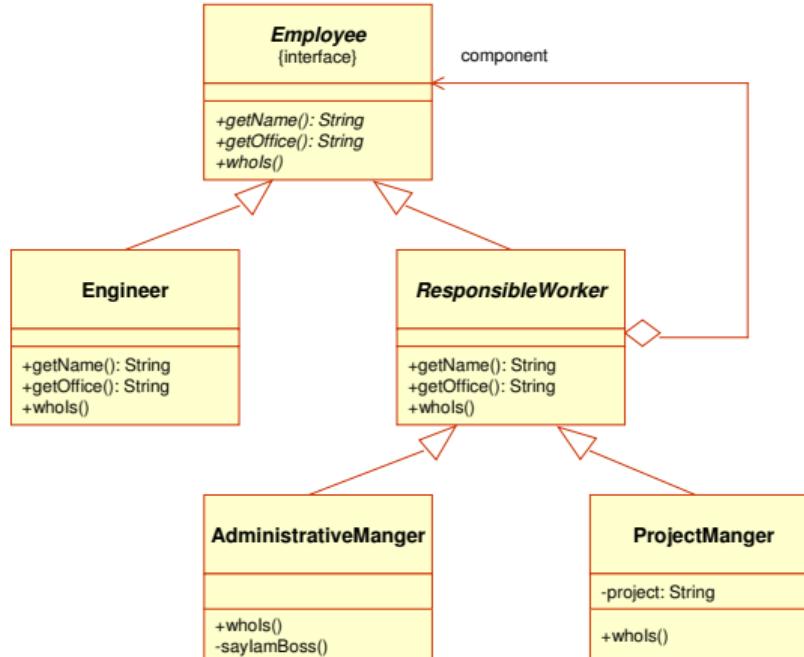
- Permette di **aggiungere responsabilità** ad oggetti **individualmente, dinamicamente** e in modo trasparente, ossia senza impatti sugli altri oggetti
- Le responsabilità possono essere **ritirate**
- Permette di **evitare l'esplosione delle sottoclassi** per supportare un ampio numero di estensioni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

# Struttura del pattern Decorator



# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



```
1 public class DecoratorExample1 {
2     public static void main(String arg[]) {
3
4         Employee thisWillBeFamous = new Engineer( "William Gateway", "Programming Department" );
5
6         System.out.println( "Who are you?" );
7         thisWillBeFamous.whoIs();
8
9         thisWillBeFamous = new AdministrativeManager( "William Gateway", "Programming Department" );
10
11        System.out.println( "Who are you now?" );
12        thisWillBeFamous.whoIs();
13
14        thisWillBeFamous = new ProjectManager( "William Gateway", "Programming Department", "D.O.S.- Doors Operating System" );
15
16        System.out.println( "Who are you now?" );
17        thisWillBeFamous.whoIs();
18
19        thisWillBeFamous = new ProjectManager( "William Gateway", "Programming Department", "EveryoneLoggedToInternet Explorer" );
20
21        System.out.println( "Who are you now?" );
22        thisWillBeFamous.whoIs();
23
24    }
25 }
```

The code editor interface includes tabs for 'Save', 'Three-pane view', and 'Close'. The status bar at the bottom shows 'Java', 'Tab Width: 8', 'Ln 1, Col 1', and 'INS'.

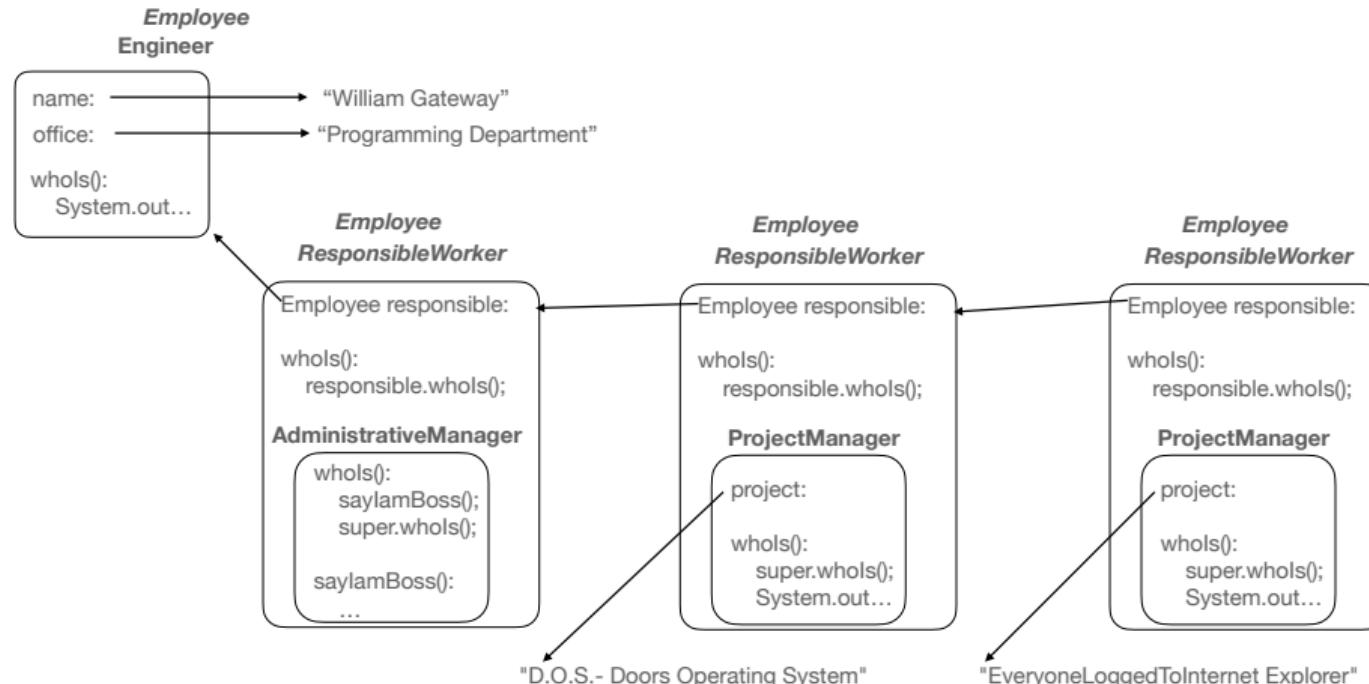
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.

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

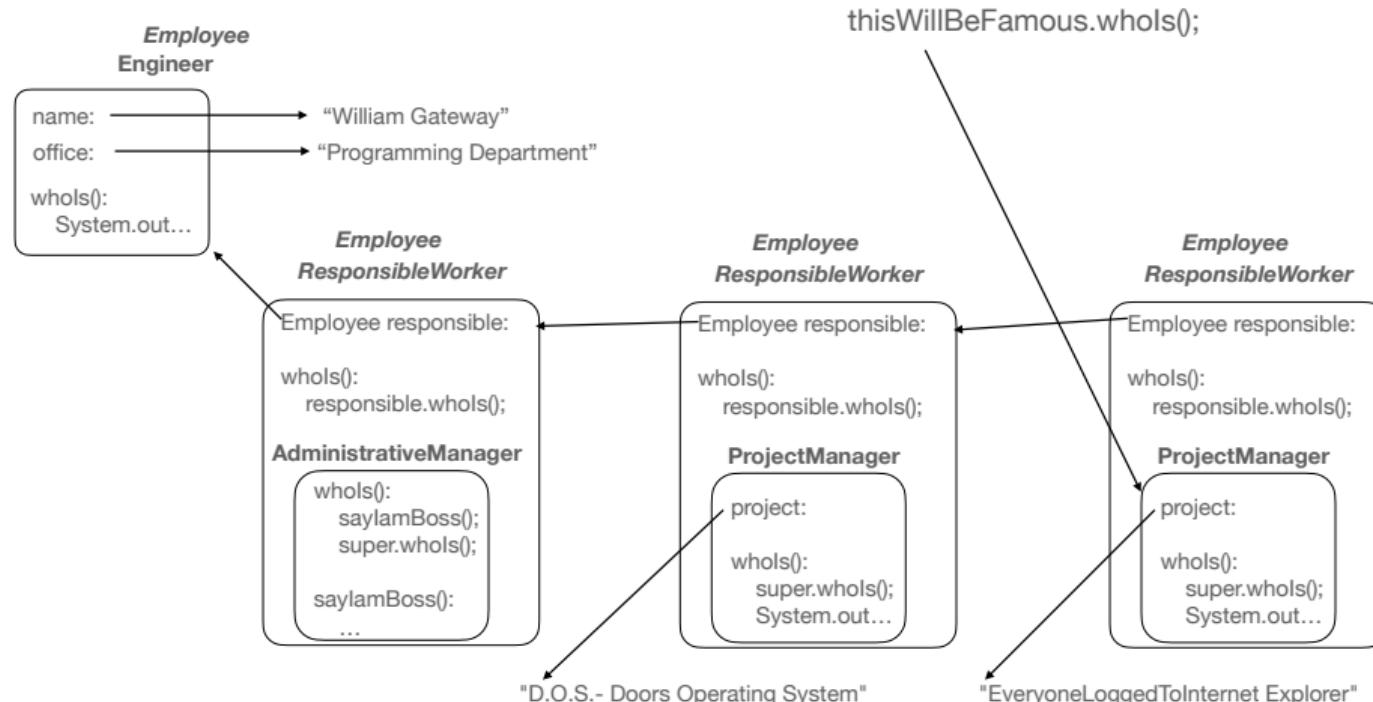
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



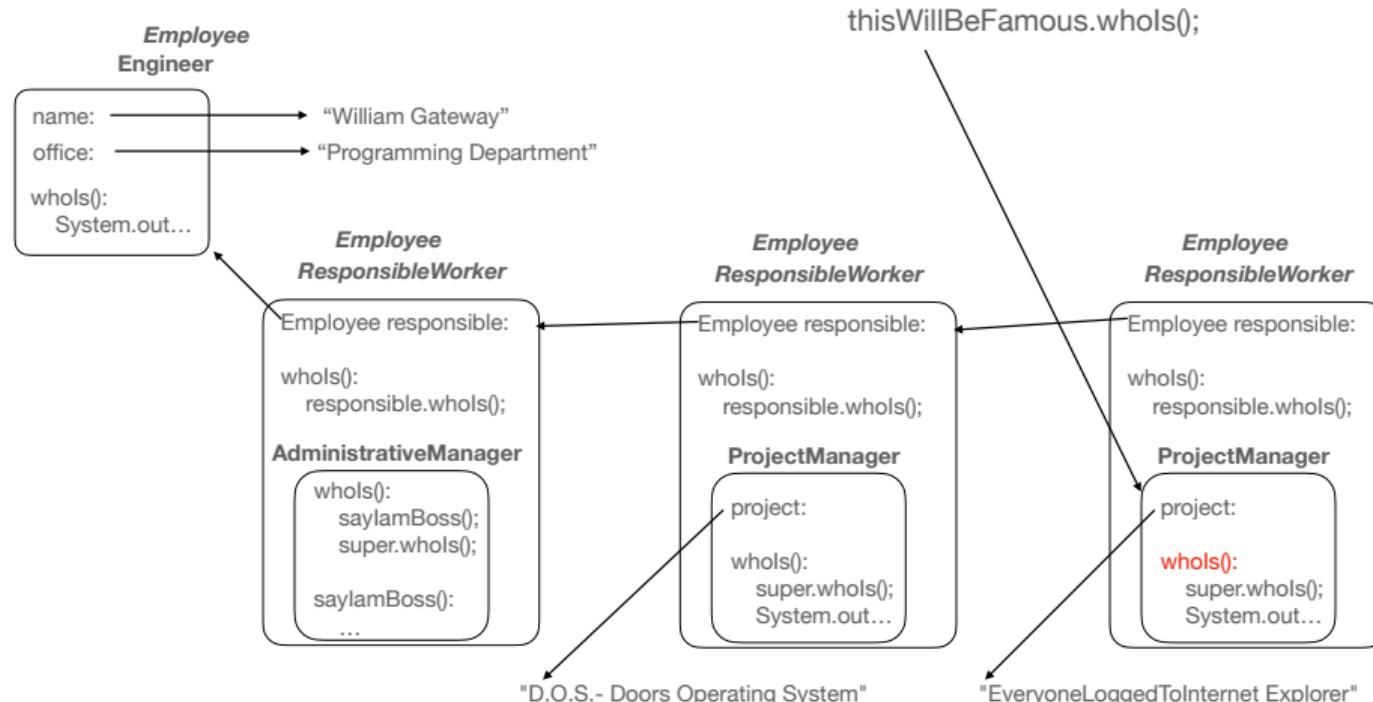
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



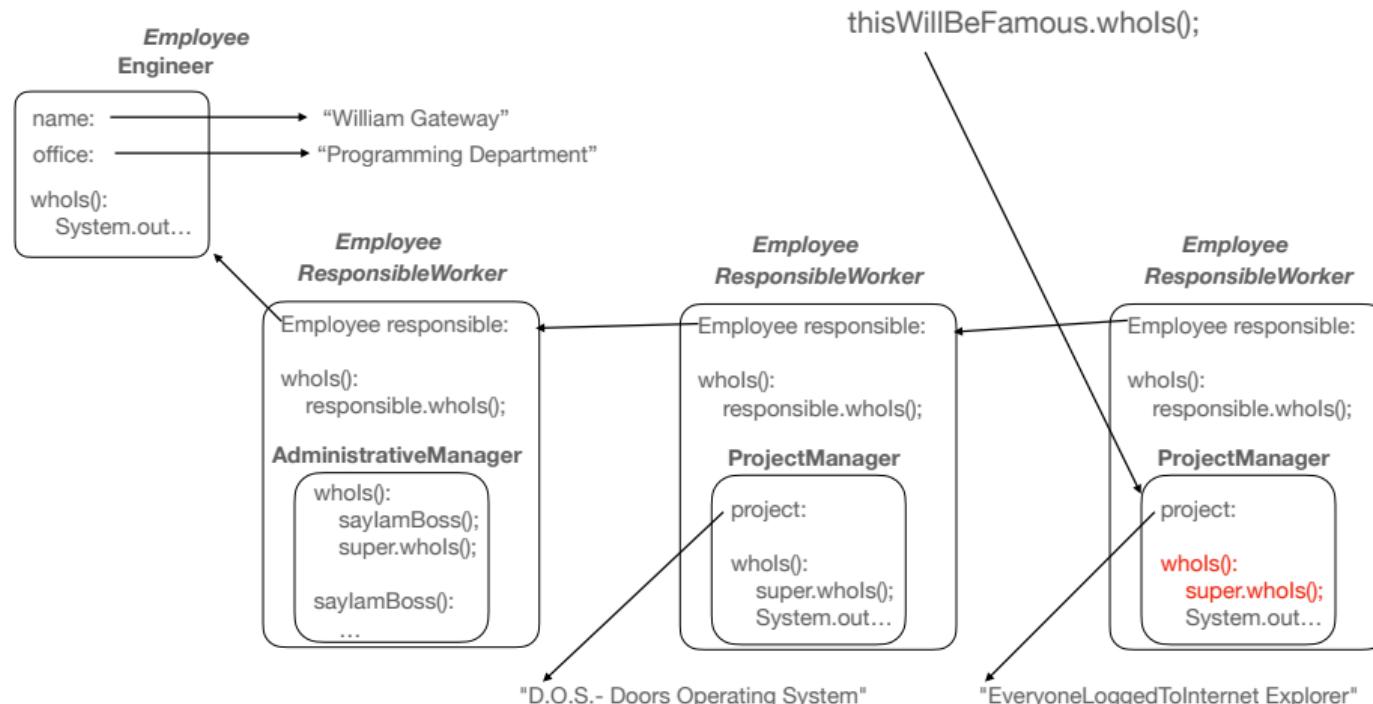
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



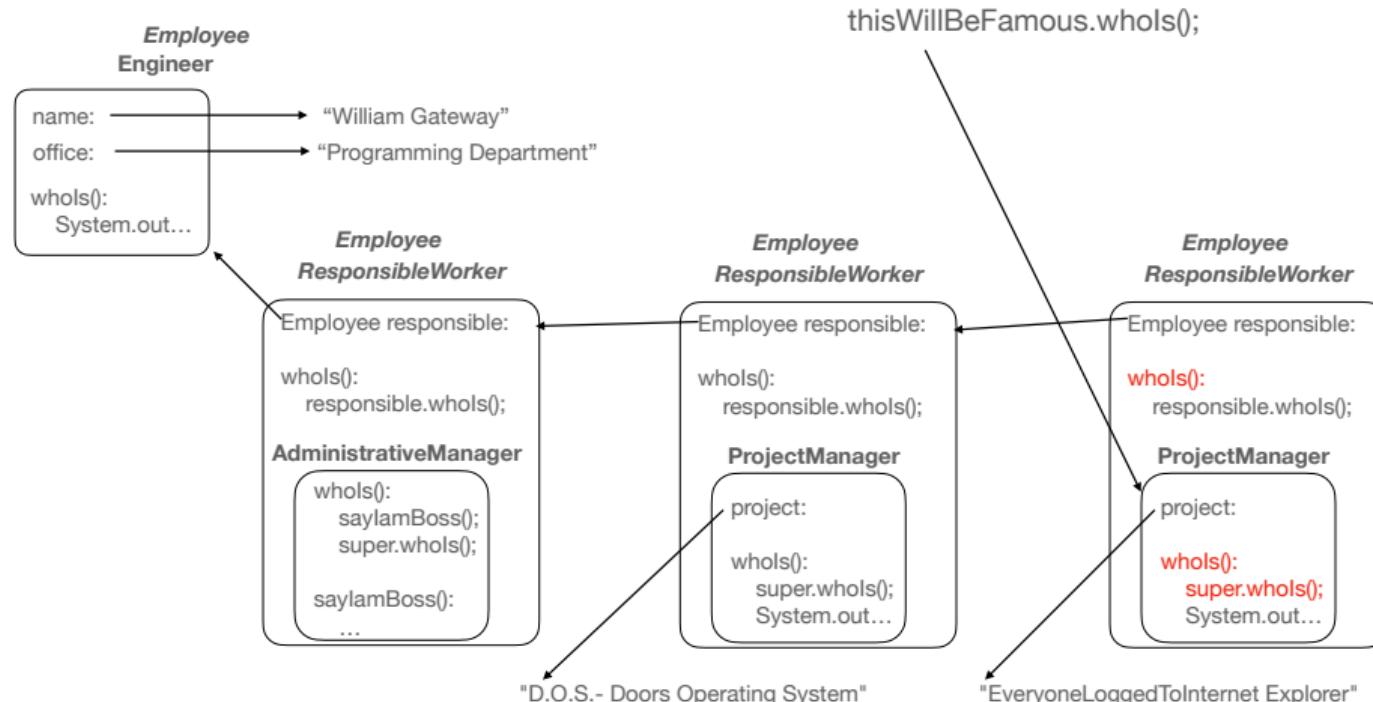
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



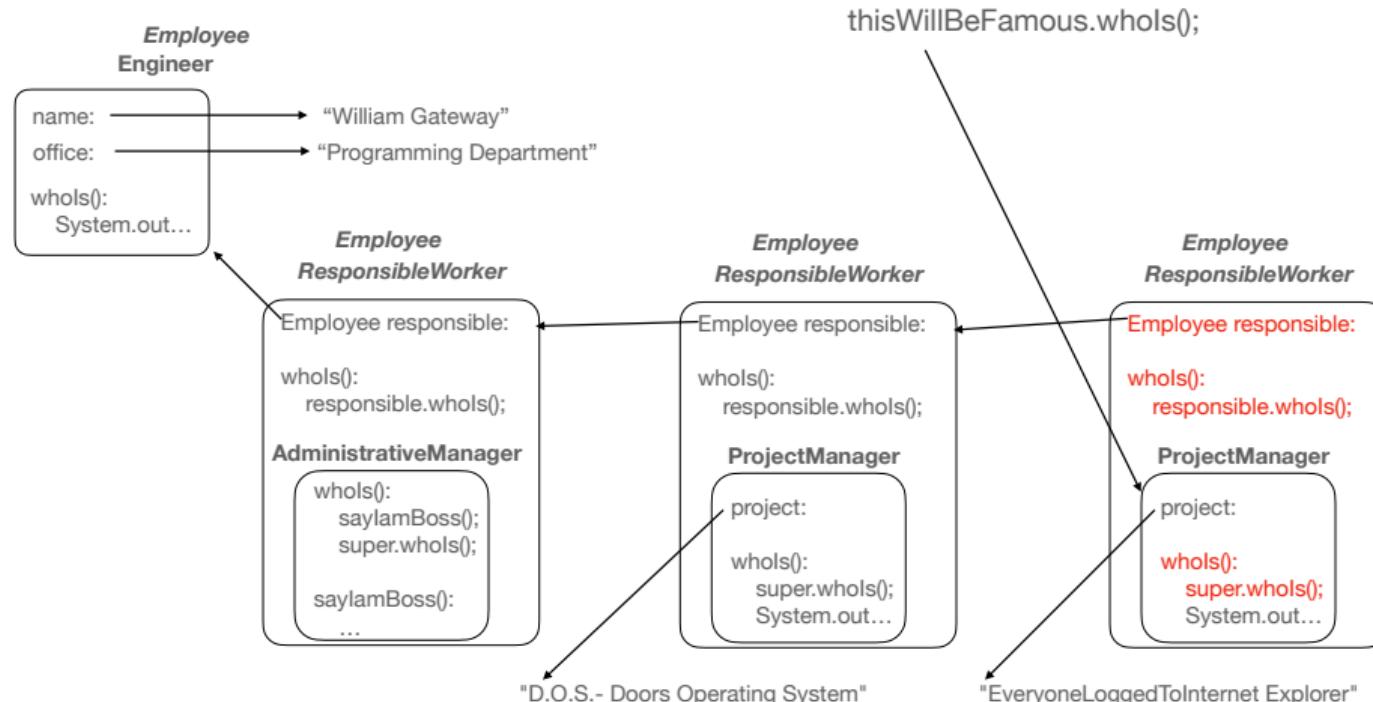
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



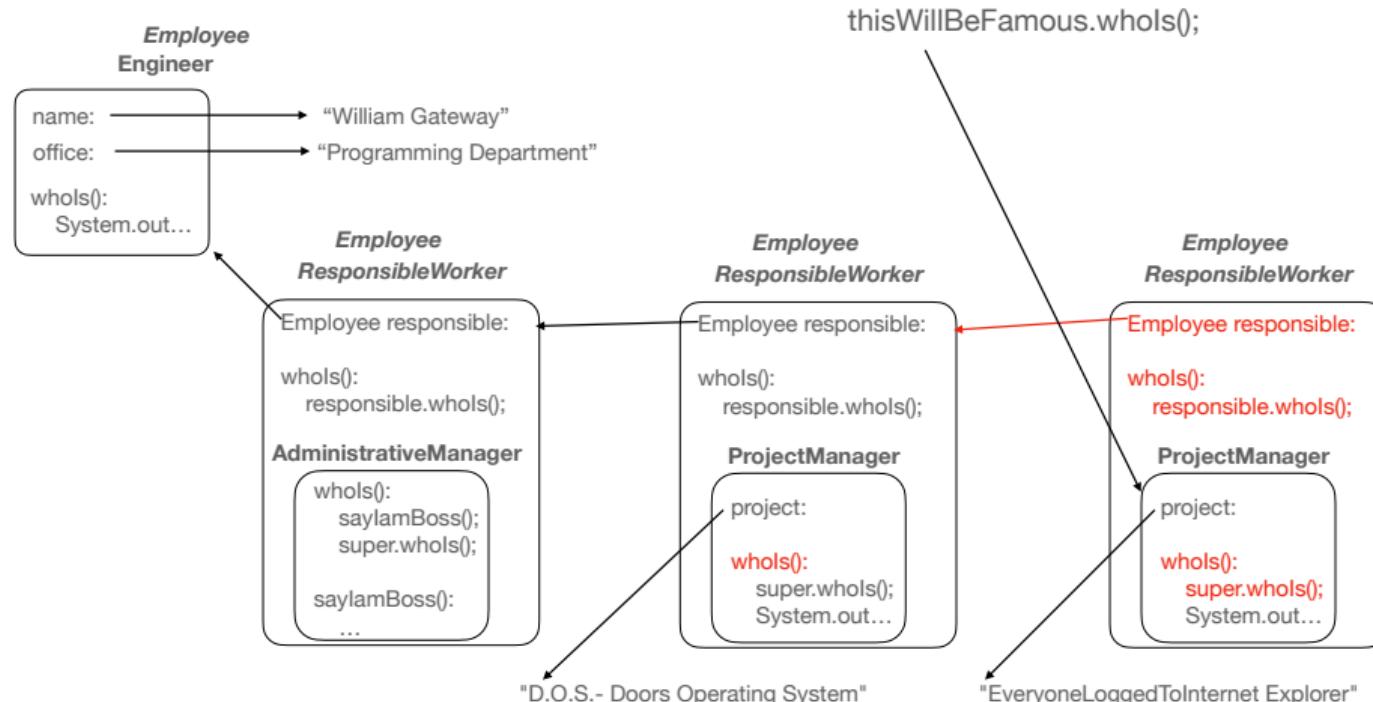
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



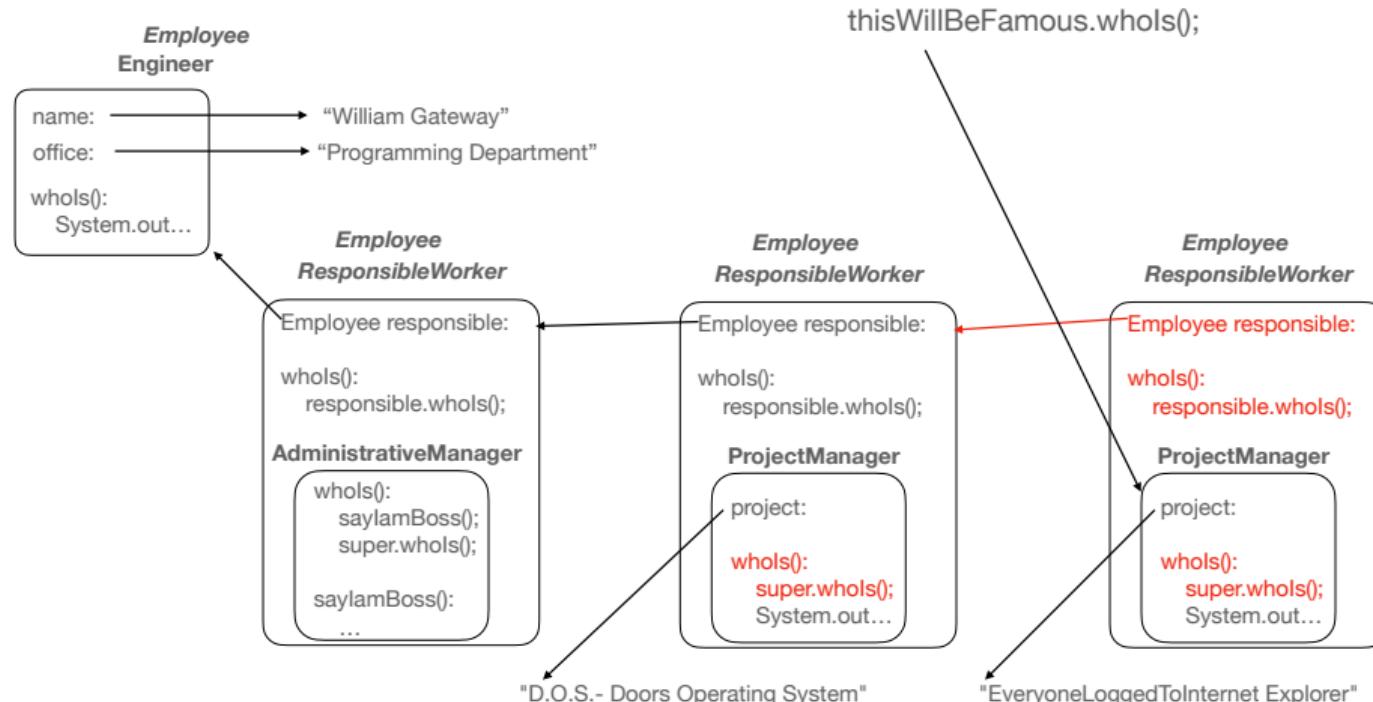
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



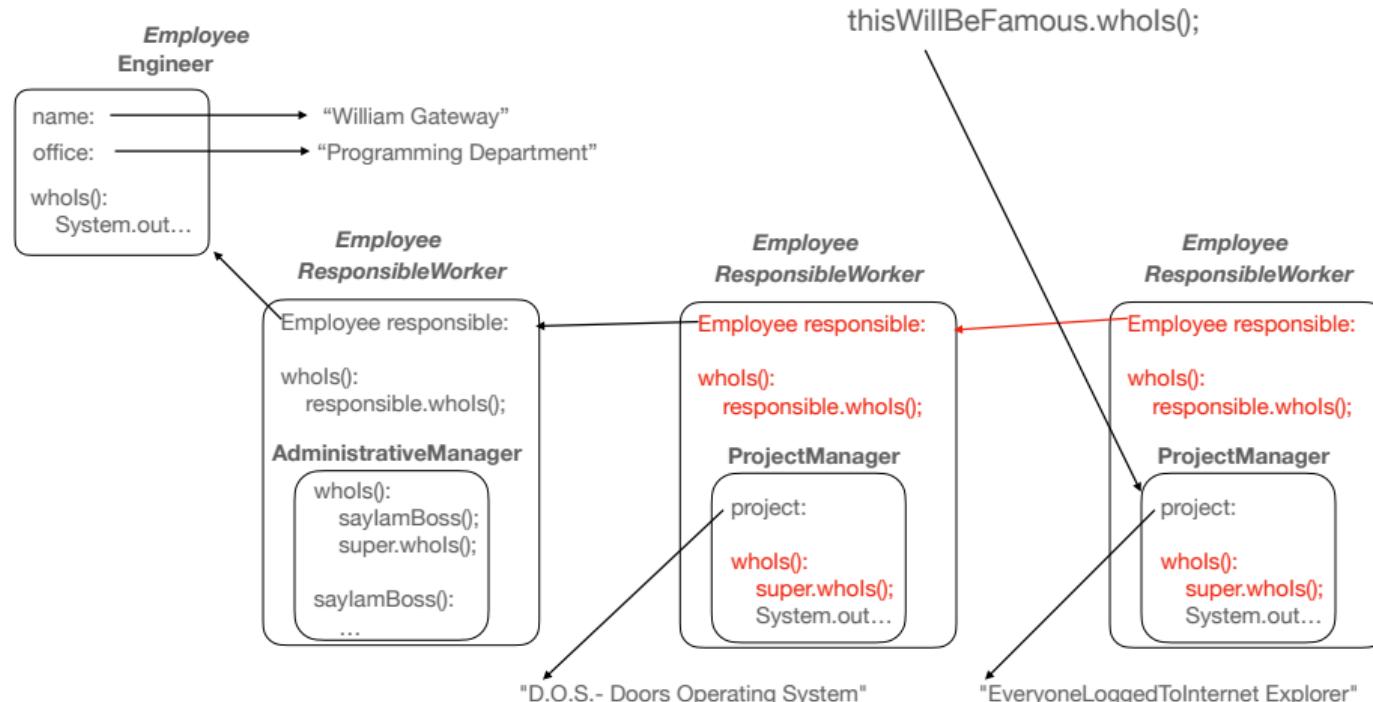
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



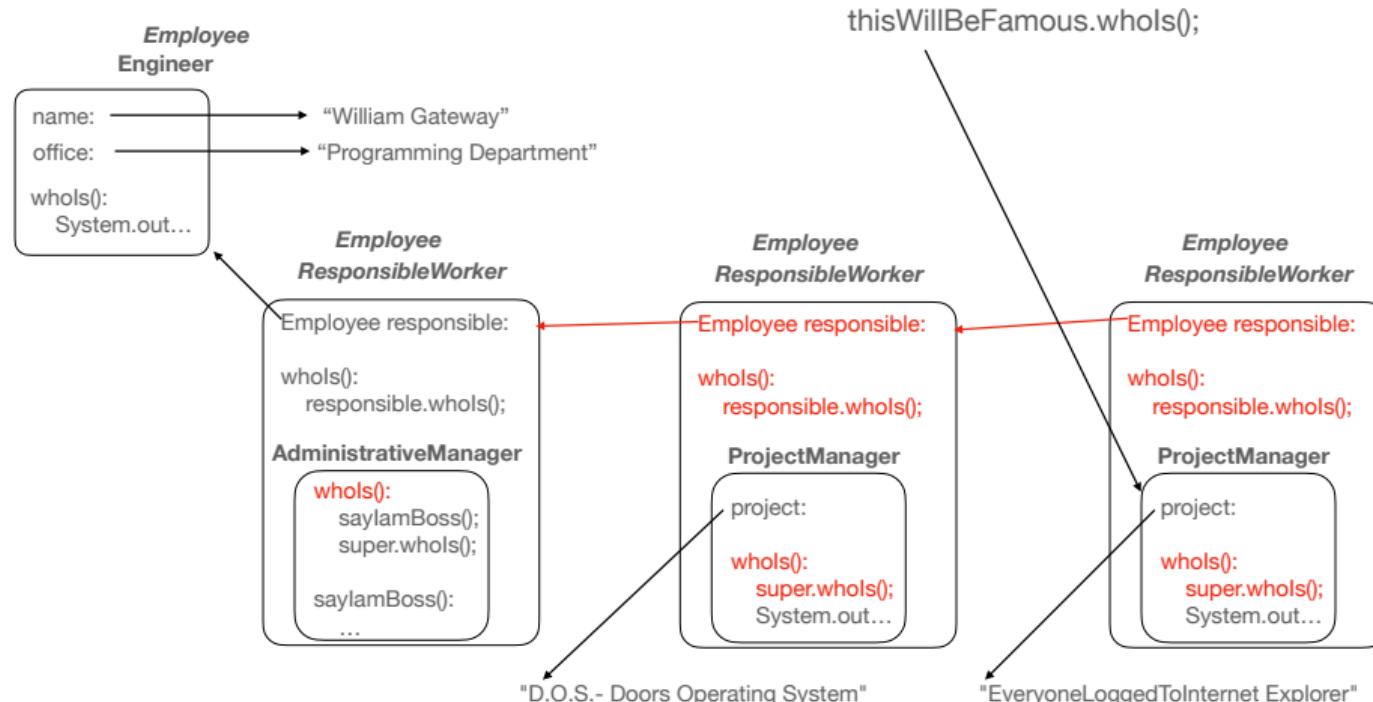
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



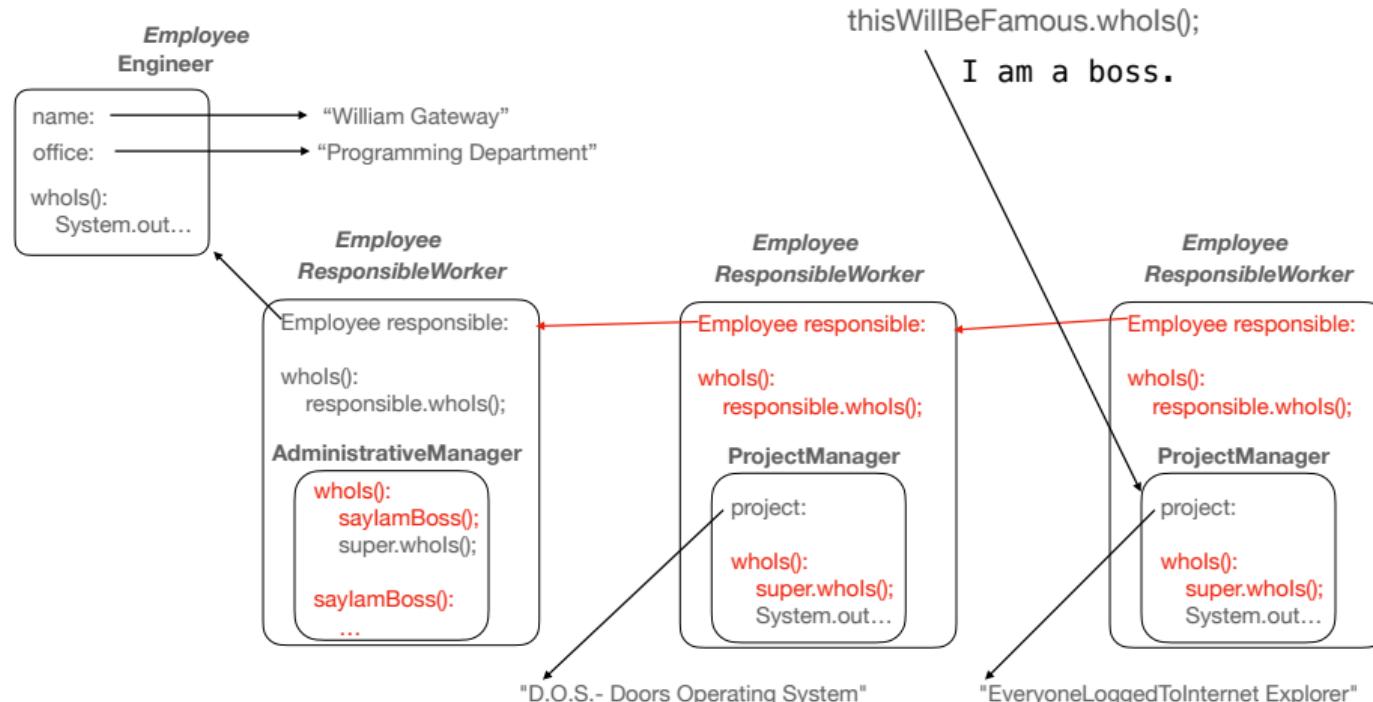
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



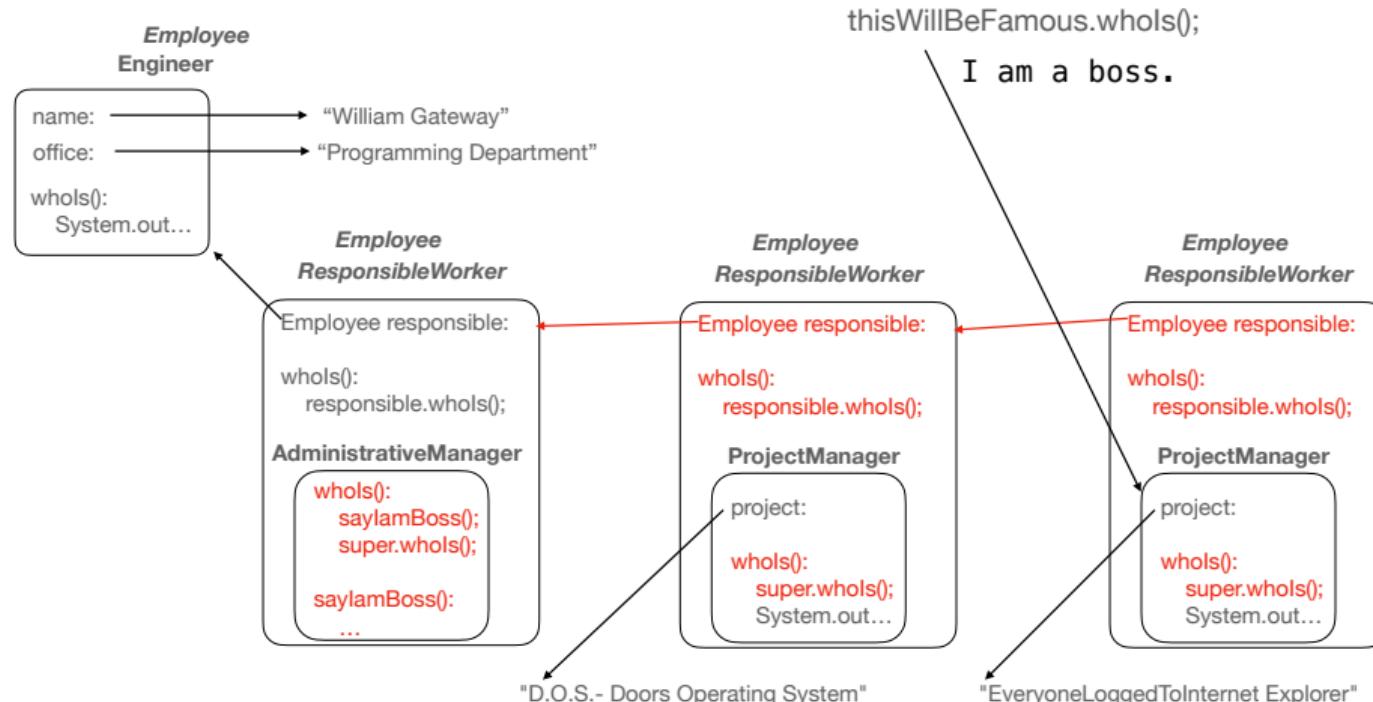
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



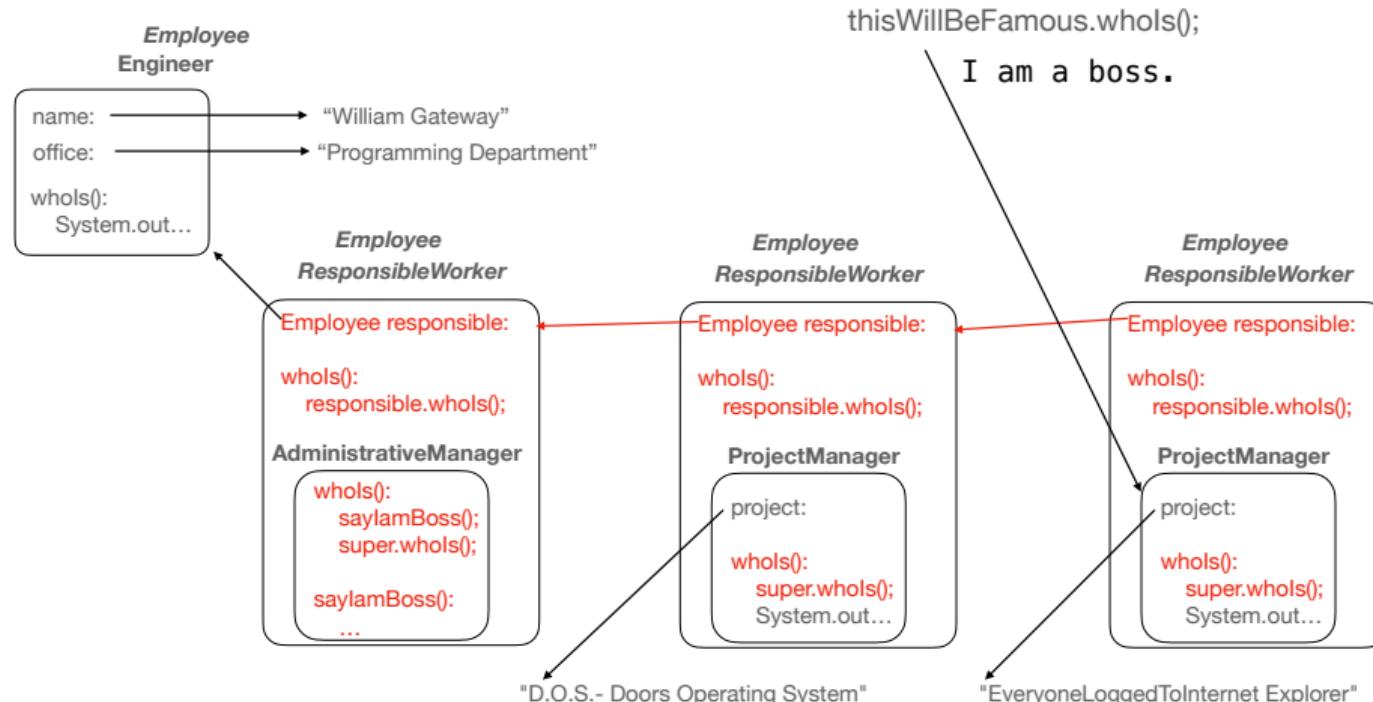
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



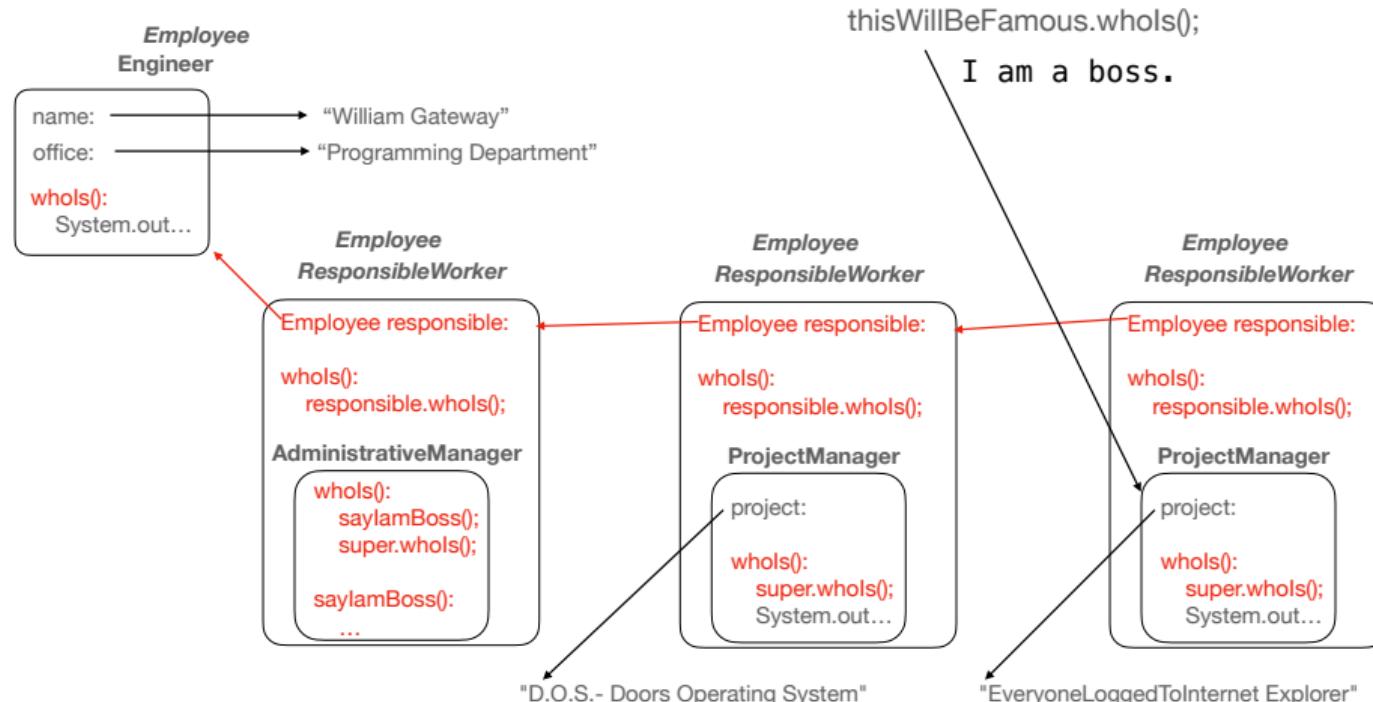
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



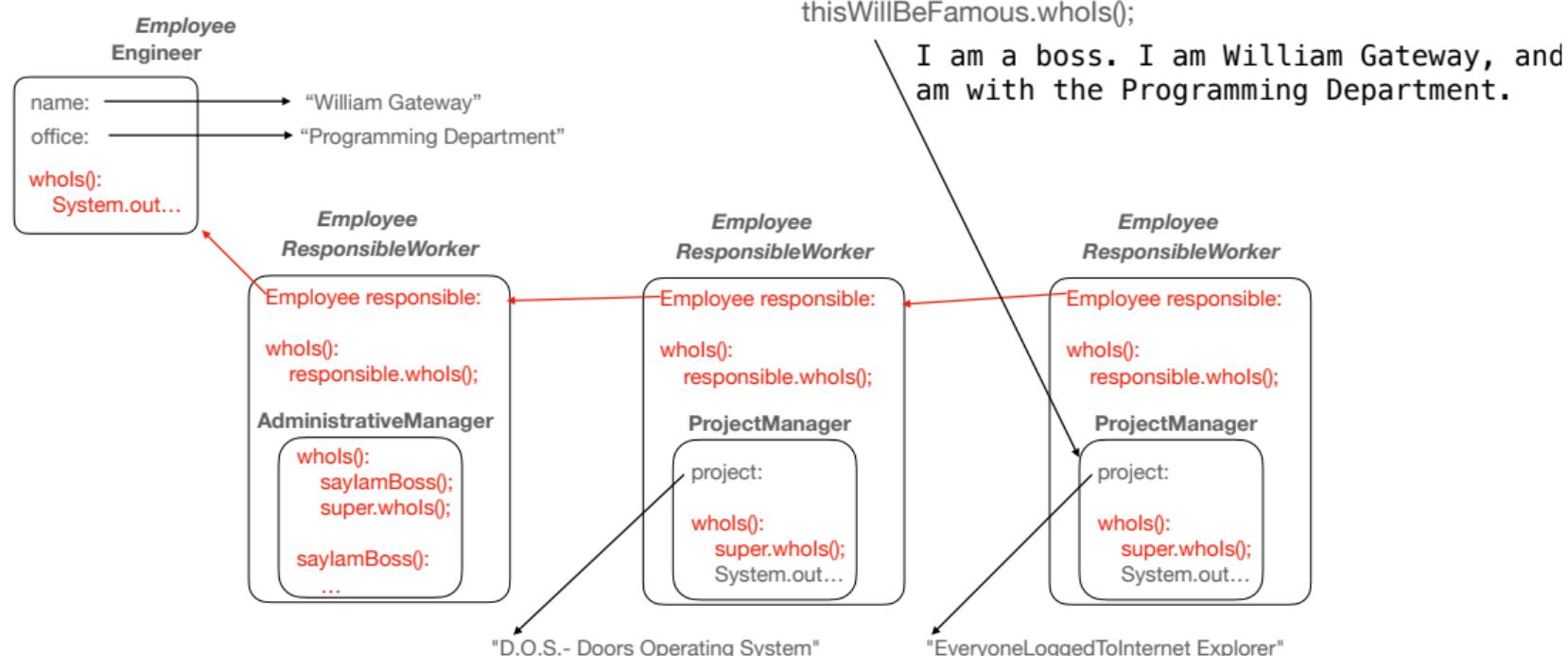
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



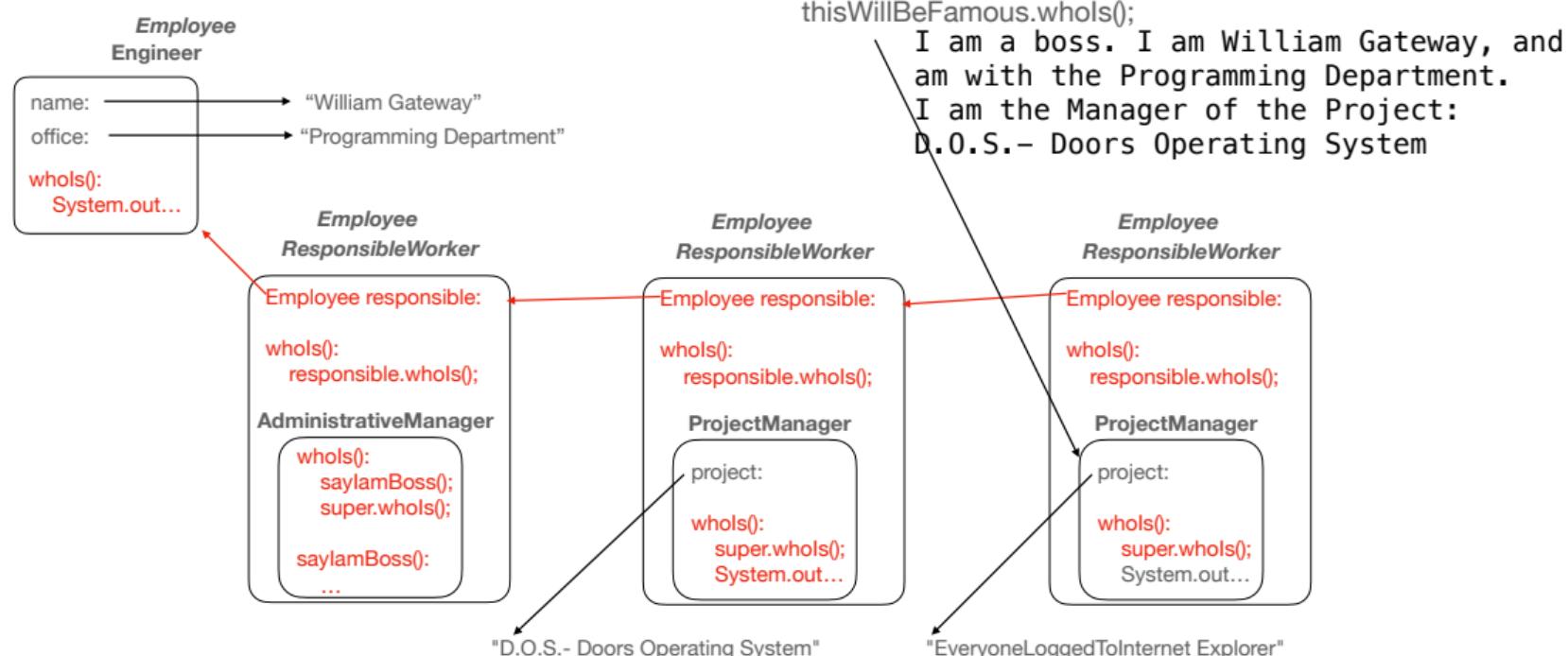
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



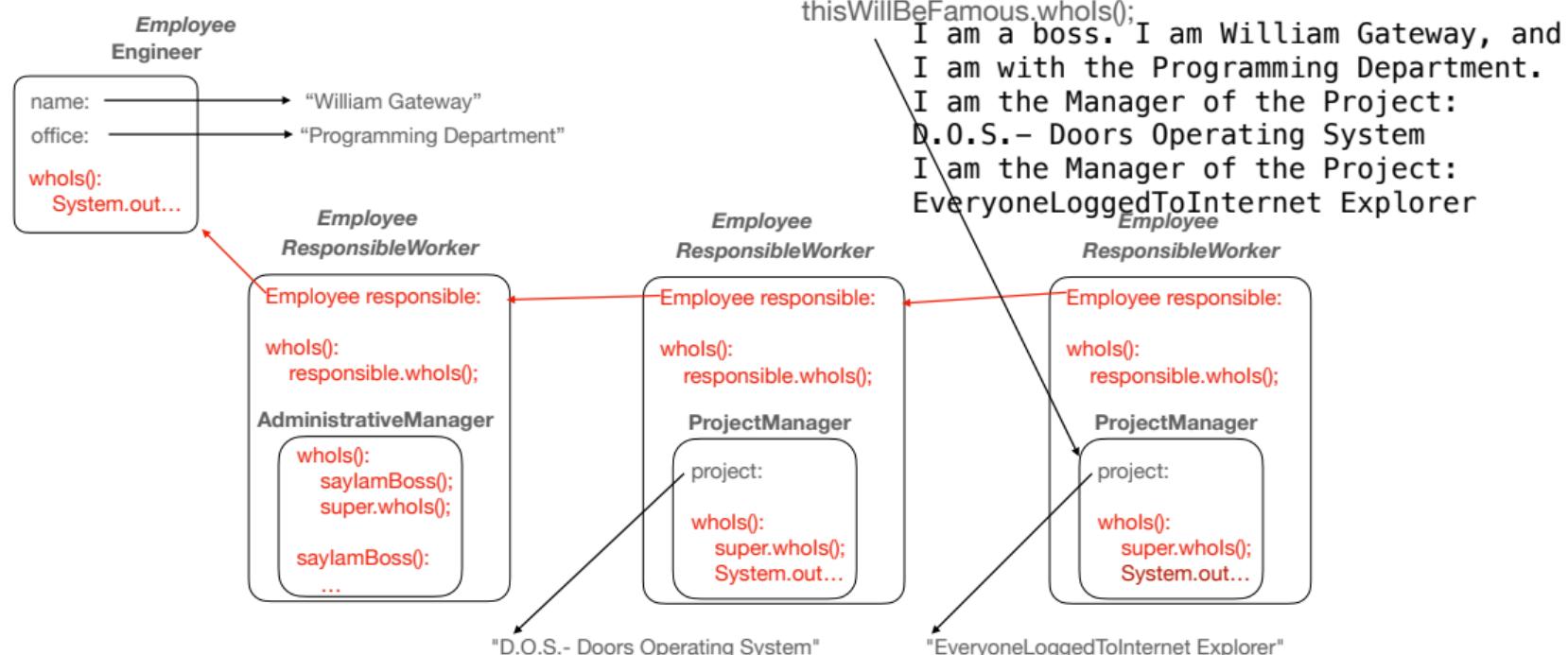
# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.

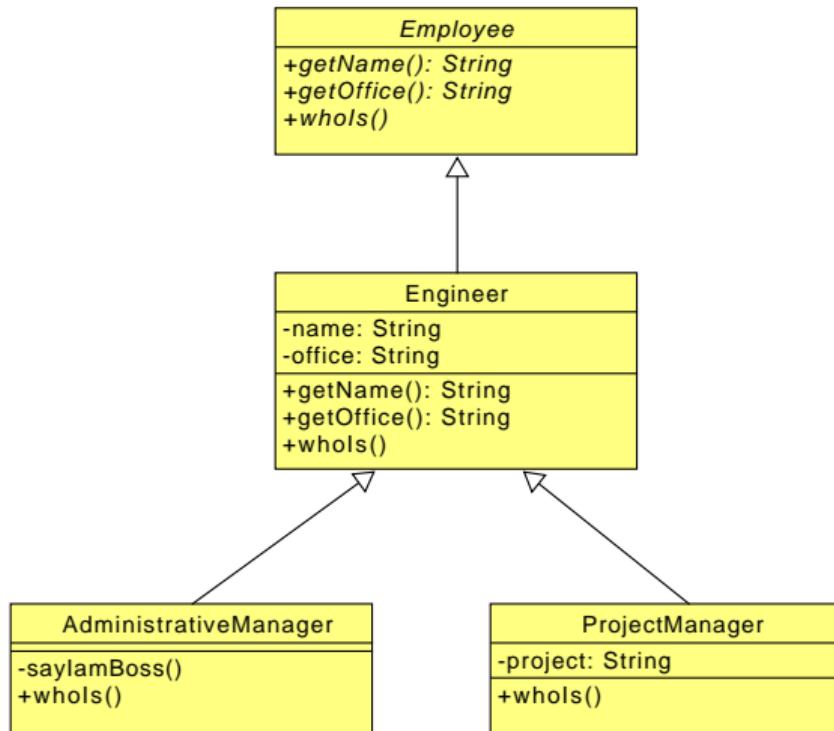


# Applicazione del pattern Decorator

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 65–72.



# Senza usare Decorator: Estensione



# Senza usare Decorator: Estensione



```
NoDecoratorExtendsExample.java
~/Nextcloud/Materiale Corsi/aa2122/...F Strutturali/05-NoDecoratorExtends
Save  ⌂  -  □  ×

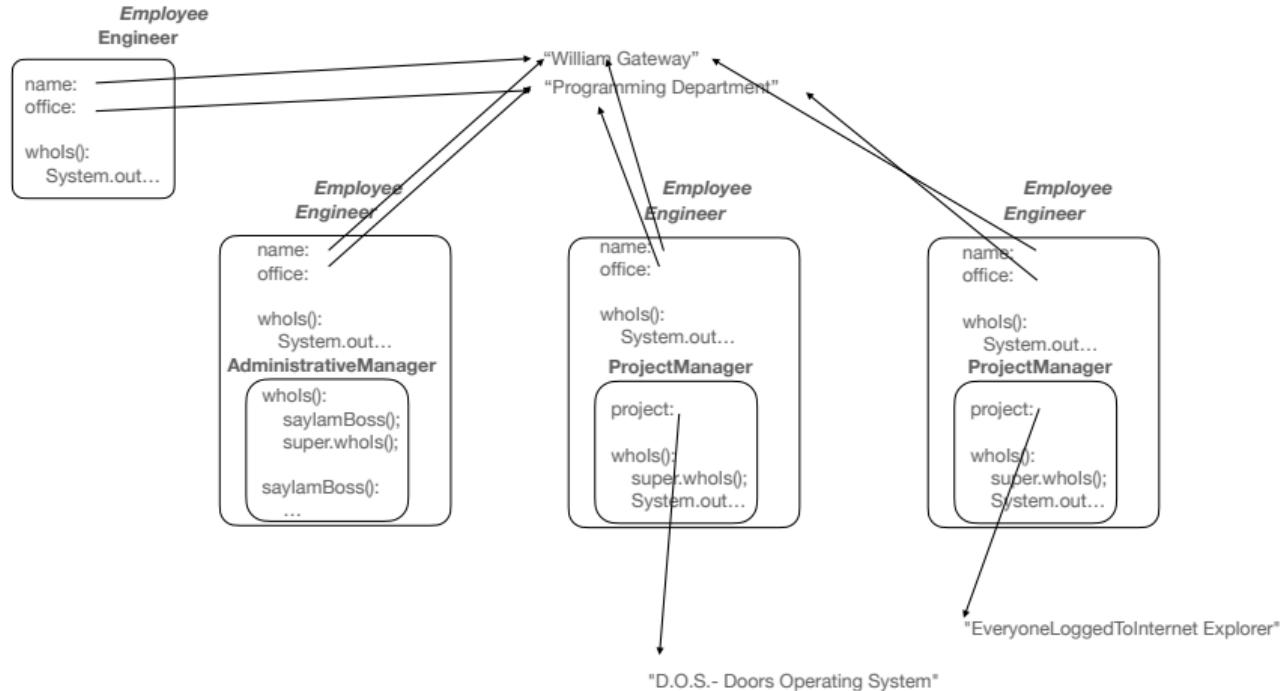
1 public class NoDecoratorExtendsExample {
2     public static void main(String arg[]) {
3
4         Employee thisWillBeFamous = new Engineer( "William Gateway", "Programming Department" );
5
6         System.out.println( "Who are you?" );
7         thisWillBeFamous.whoIs();
8
9         thisWillBeFamous = new AdministrativeManager( "William Gateway", "Programming Department" );
10
11        System.out.println( "Who are you now?" );
12        thisWillBeFamous.whoIs();
13
14        thisWillBeFamous = new ProjectManager( "William Gateway", "Programming Department", "D.O.S.- Doors Operating System" );
15
16        System.out.println( "Who are you now?" );
17        thisWillBeFamous.whoIs();
18
19        thisWillBeFamous = new ProjectManager( "William Gateway", "Programming Department", "EveryoneLoggedToInternet Explorer" );
20
21        System.out.println( "Who are you now?" );
22        thisWillBeFamous.whoIs();
23
24    }
25 }
```

Java ▾ Tab Width: 8 ▾ Ln 1, Col 32 ▾ IINS

# Senza usare Decorator: Estensione

```
baldoni@Shizuka: ~/Nextcloud/MaterialeCorsi/aa2122/SAS/GoF/GoF Strutturali/05-NoDecoratorExtends$ java NoDecoratorExtendsExample
Who are you?
I am William Gateway (Engineer@816f27d), and I am with the Programming Department.
Who are you now?
I am a boss. I am William Gateway (AdministrativeManager@3e3abc88), and I am with the Programming Department.
Who are you now?
I am William Gateway (ProjectManager@53d8d10a), and I am with the Programming Department.
I am the Manager of the Project:D.O.S.- Doors Operating System
Who are you now?
I am William Gateway (ProjectManager@214c265e), and I am with the Programming Department.
I am the Manager of the Project:EveryoneLoggedToInternet Explorer
baldoni@Shizuka: ~/Nextcloud/MaterialeCorsi/aa2122/SAS/GoF/GoF Strutturali/05-NoDecoratorExtends$
```

# Senza usare Decorator: Estensione



**Non è assolutamente quanto desideriamo! Non è possibile ottenere lo stesso risultato!**

Oggetti diversi, administrative manager e project manager in esclusione, project manager di un solo progetto, ...

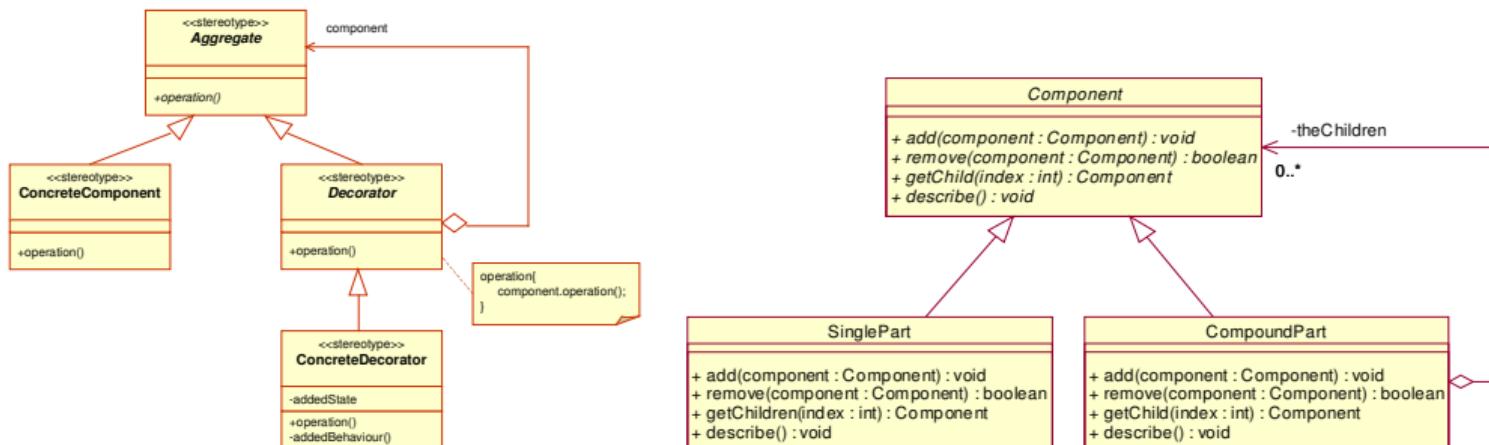
## In breve

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

# Decorator

## Composite vs Decorator

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



## **Comportamentali: Observer, State, Strategy e Visitor**

---

## Observer

**Nome:** 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 un suo modo proprio quando il publisher genera un evento. Inoltre, il publisher vuole mantenere un accoppiamento basso verso i subscriber. Che cosa fare?

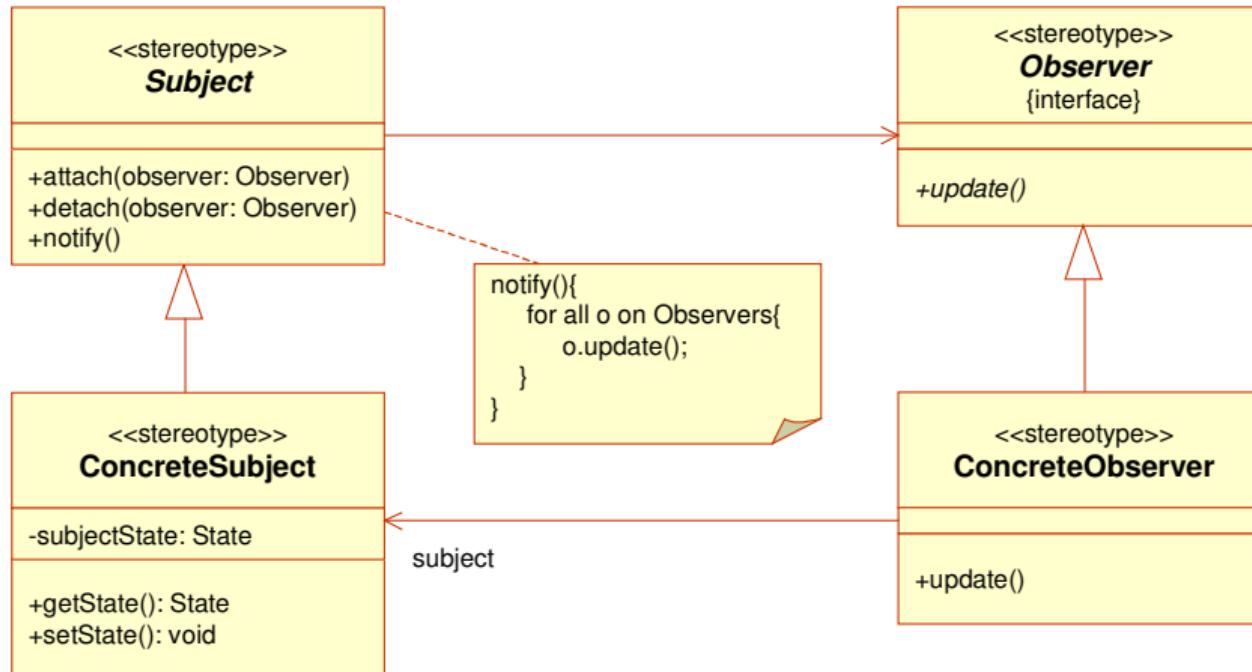
**Soluzione:** Definisci un'interfaccia *subscriber* o *listener* (ascoltatore). Gli oggetti subscriber implementano questa interfaccia. Il publisher registra dinamicamente i subscriber che sono interessati ai suoi eventi, e li avvisa quando si verifica un evento.

Noto anche come *Dependents*, *Publish-Subscribe*.

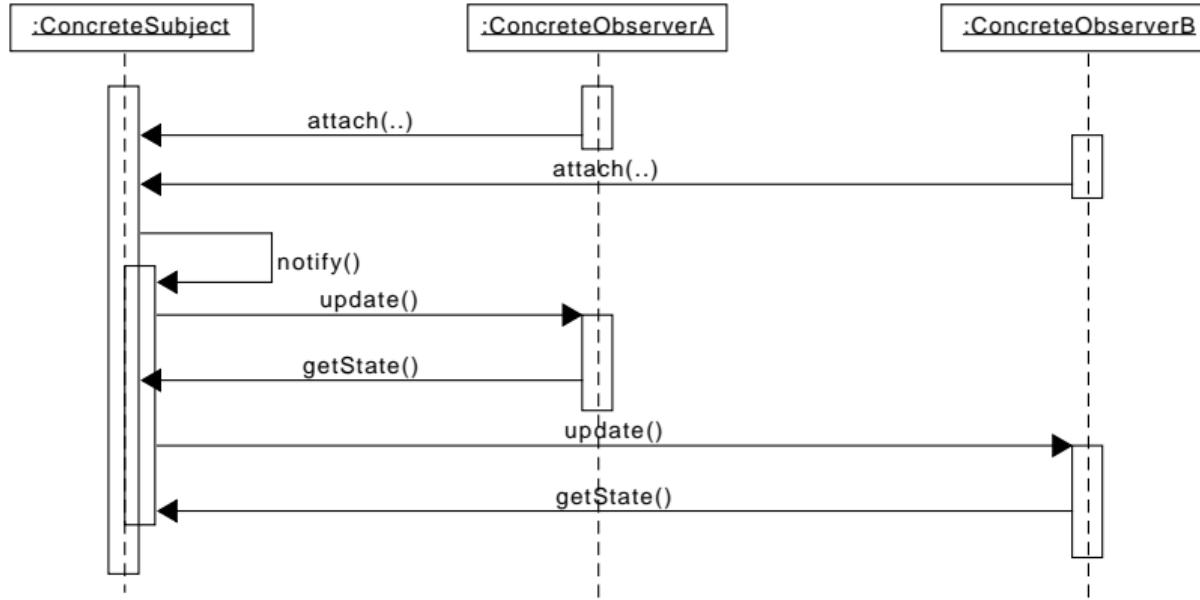
- Definisce una dipendenza tra oggetti di tipo **uno-a-molti**: quando lo stato di un oggetto cambia, tale evento viene notificato a tutti gli oggetti dipendenti, essi vengono automaticamente aggiornati
- L'oggetto che notifica il cambiamento di stato non fa alcuna assunzione sulla natura degli oggetti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

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

# Struttura del pattern Observer



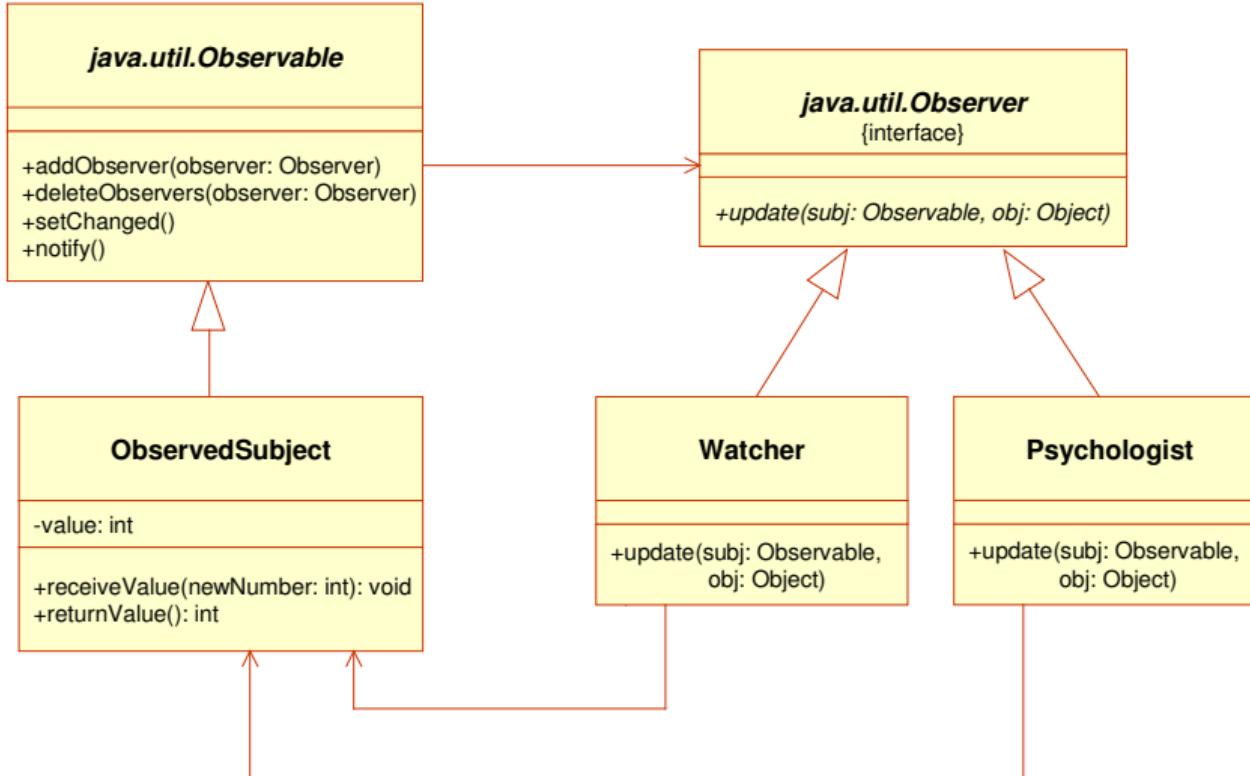
# Struttura del pattern Observer



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

# Applicazione del pattern Observer

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 129–135.



## State

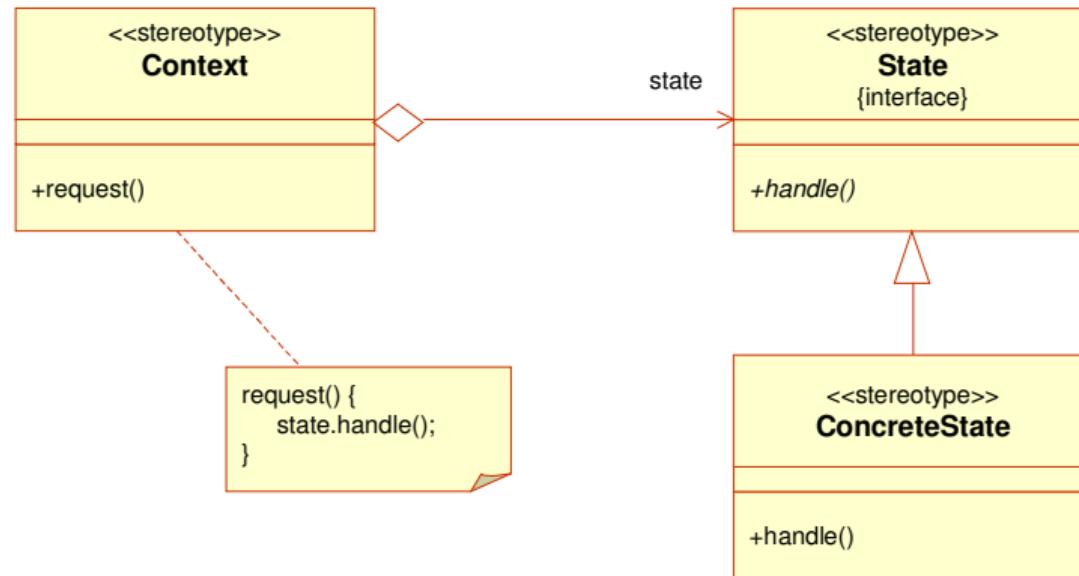
**Nome:** State

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

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

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

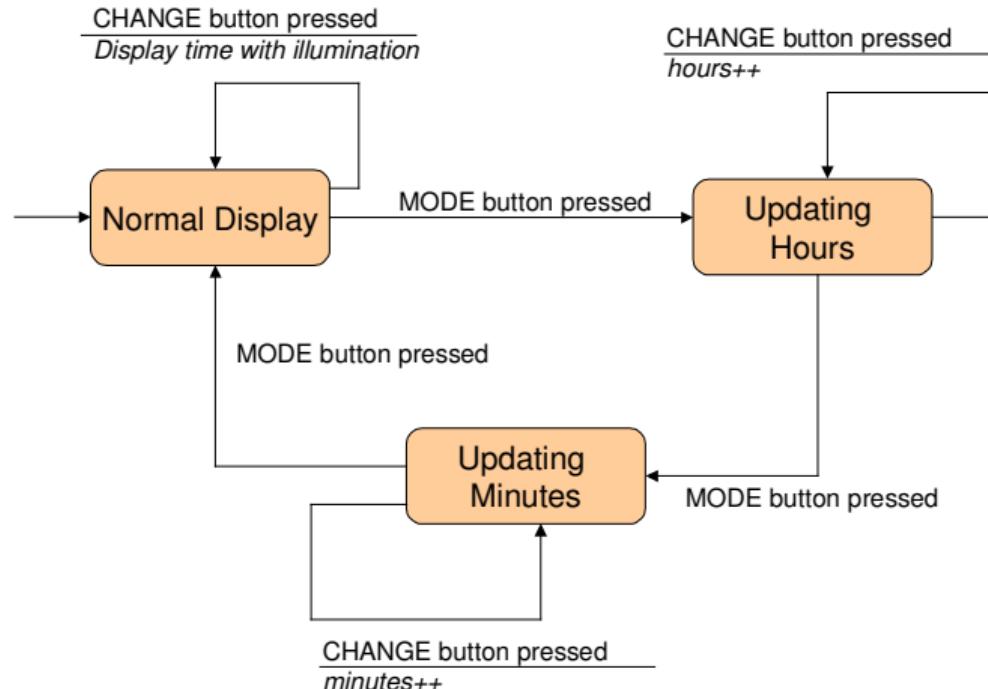
# Struttura del pattern State



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

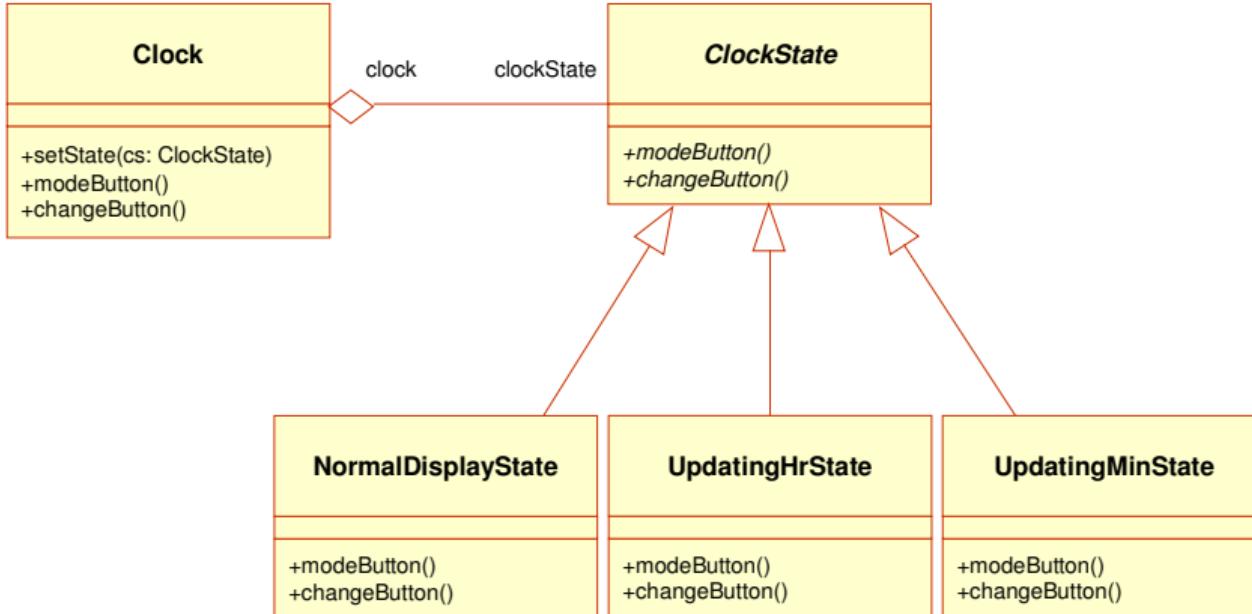
# Applicazione del pattern State

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 136–141.



# Applicazione del pattern State

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 136–141.



## Strategy

**Nome:** Strategy

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

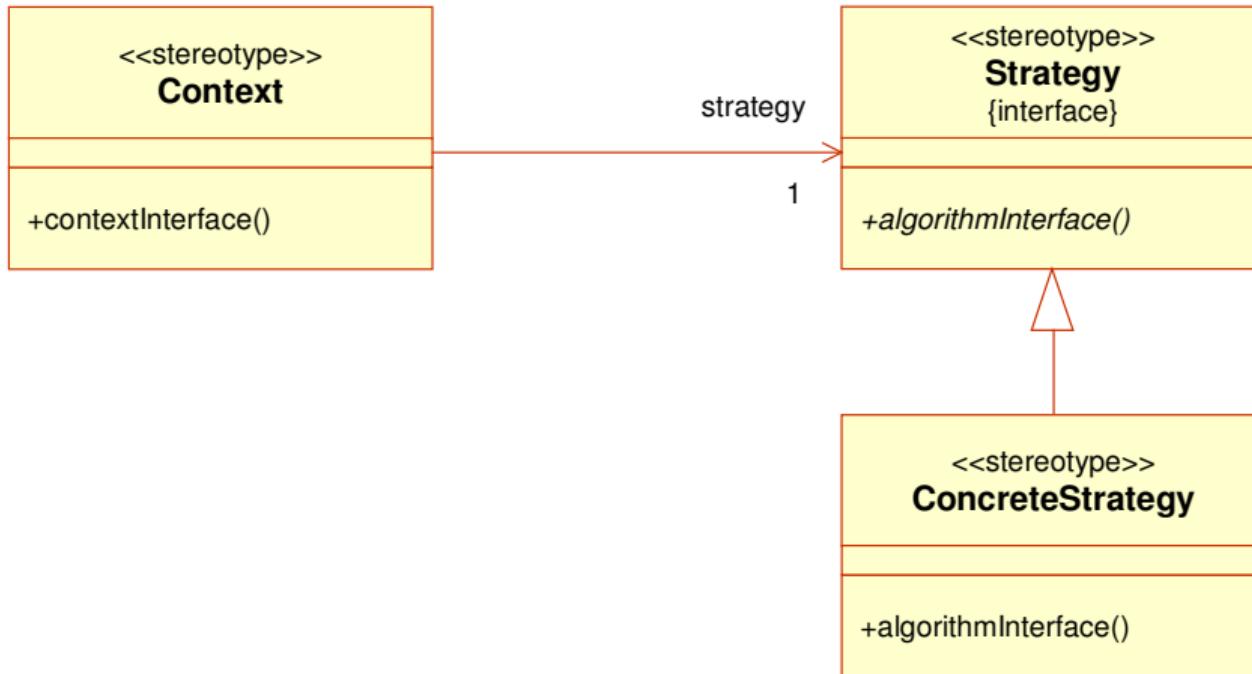
**Soluzione:** Definisci ciascun algoritmo/politica/strategia in una classe separata, con un'interfaccia comune.

Noto anche come *Policy*.

- L'oggetto contesto è l'oggetto a cui va applicato l'algoritmo
- L'oggetto contesto è associato a un oggetto strategia, che è l'oggetto che implementa un algoritmo

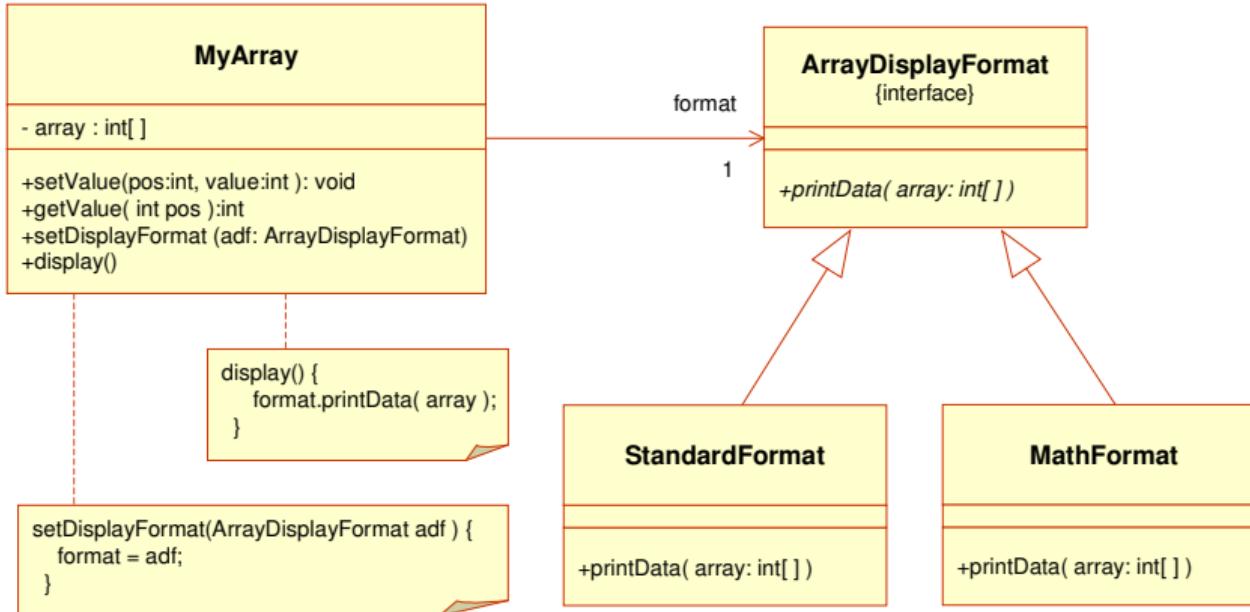
- Consente la definizione di una **famiglia di algoritmi**, incapsula ognuno e li rende intercambiabili tra di loro
- Permette di modificare gli algoritmi in modo **indipendente** dai clienti che fanno uso di essi
- **Disaccoppia** gli algoritmi dai clienti che vogliono usarli dinamicamente
- Permette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

# Struttura del pattern Strategy



# Applicazione del pattern Strategy

Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 142–145.

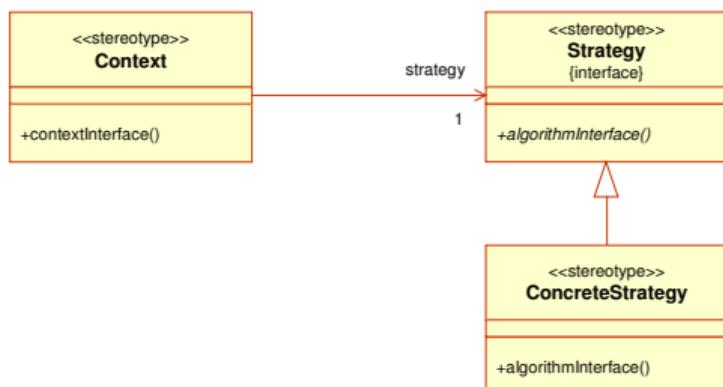


©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

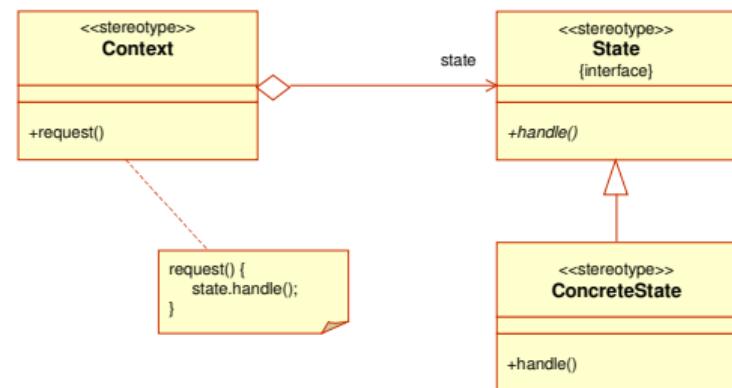
# Strategy vs State

In pratica sono molto simili...

## Pattern Strategy



## Pattern State

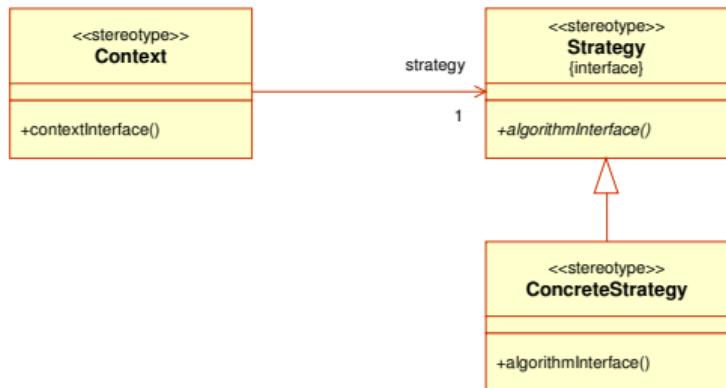


©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

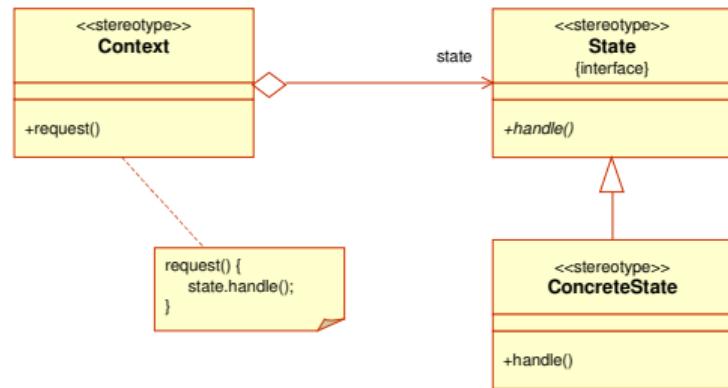
©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

# Strategy vs State

## Pattern Strategy



## Pattern State



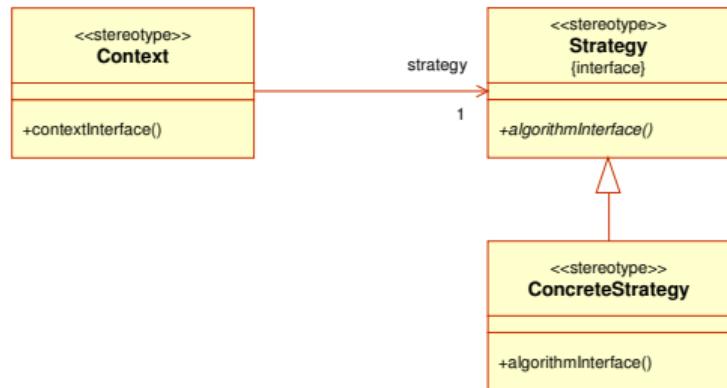
©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

Il **pattern State** si occupa di che cosa (stato o tipo) un oggetto è (al suo interno) e incapsula un comportamento dipendente dallo stato. Fare cose diverse in base allo stato, lasciando il chiamante sollevato dall'onere di soddisfare ogni stato possibile.

# Strategy vs State

## Pattern Strategy

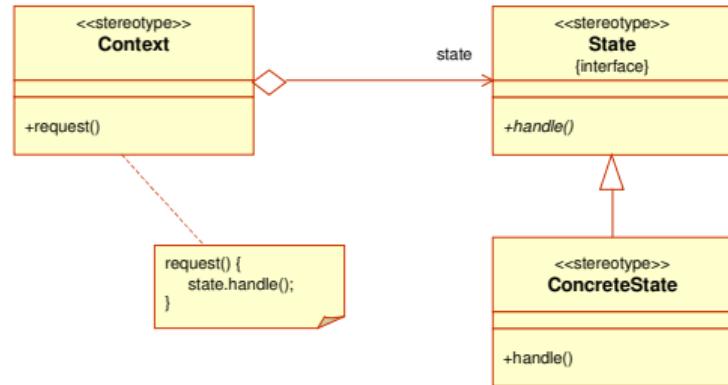


©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

mentre...

Il **pattern Strategy** si occupa del modo in cui un oggetto esegue un determinato compito: incapsula un algoritmo. Un'implementazione diversa che realizza (fondamentalmente) la stessa cosa, in modo che un'implementazione possa sostituire l'altra a seconda della strategia richiesta.

## Pattern State



©F. Guidi Polanco. GoF's Design patterns in Java, 2002.

## Visitor

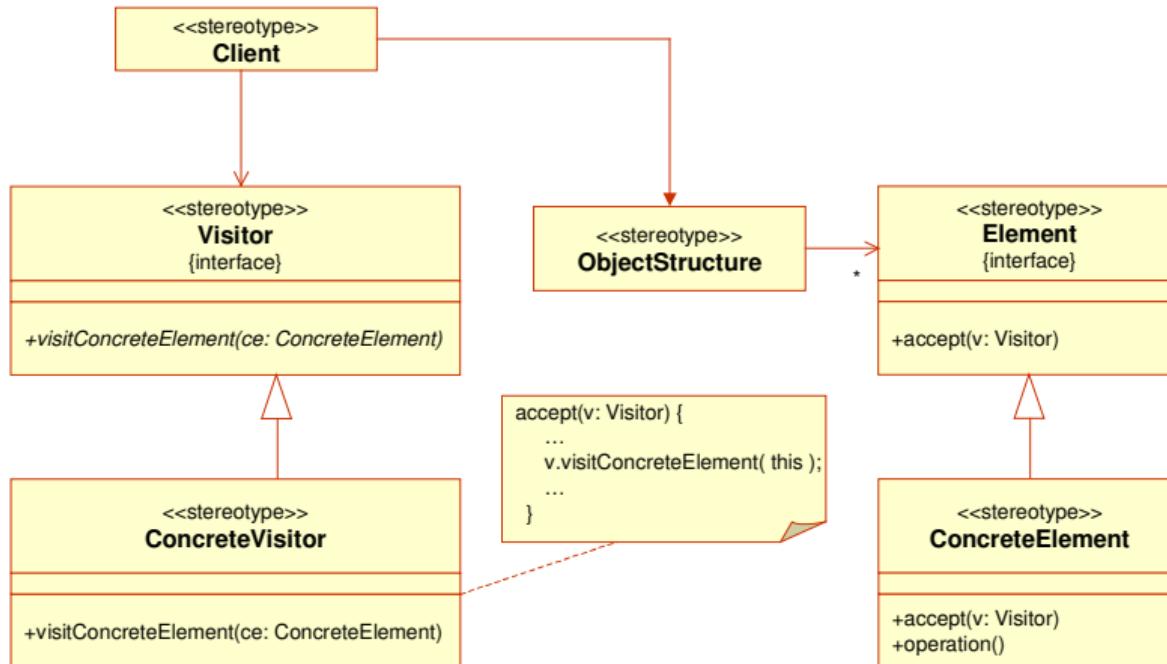
**Nome:** Visitor

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

**Soluzione:** Creare un oggetto (ConcreteVisitor) che è in grado di percorrere la collezione, e di applicare un metodo proprio su ogni oggetto (Element) visitato nella collezione (avendo un riferimento a questi ultimi come parametro). Ogni oggetto della collezione aderisce ad un'interfaccia (Visitable) che consente al ConcreteVisitor di essere accettato da parte di ogni Element. Il Visitor analizza il tipo di oggetto ricevuto, fa l'invocazione alla particolare operazione che deve eseguire.

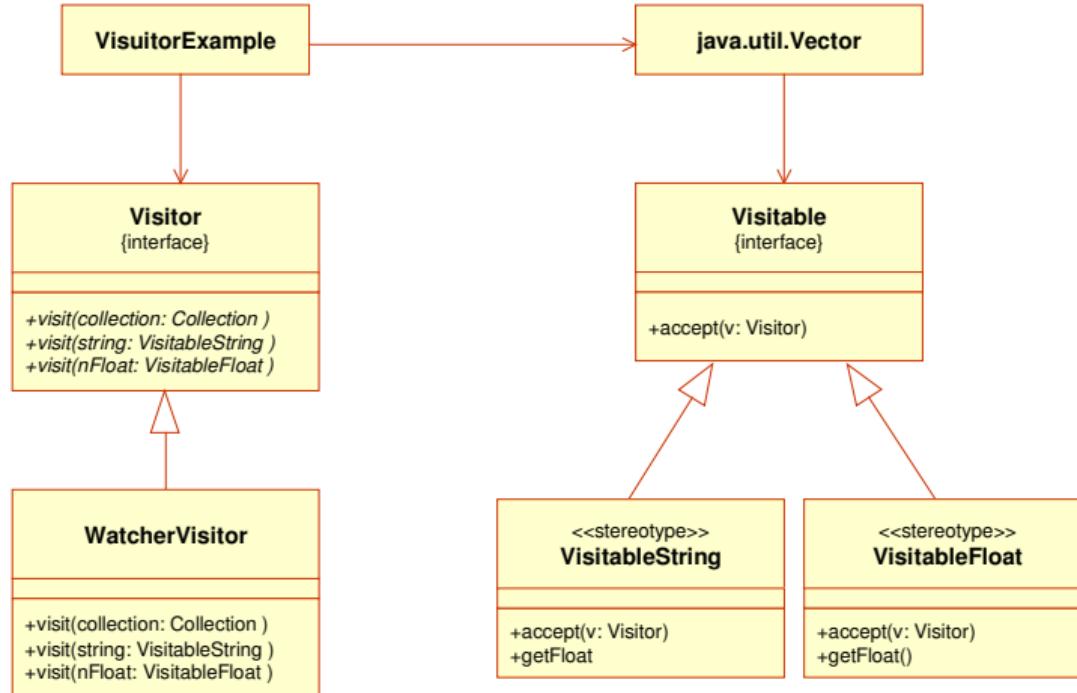
- Flessibilità delle operazioni
- Organizzazione logica
- Visita di vari tipi di classe
- Mantenimento di uno stato aggiornabile ad ogni visita
- Le diverse modalità di visita della struttura possono essere definite come sottoclassi del Visitor.

# Struttura del pattern Visitor

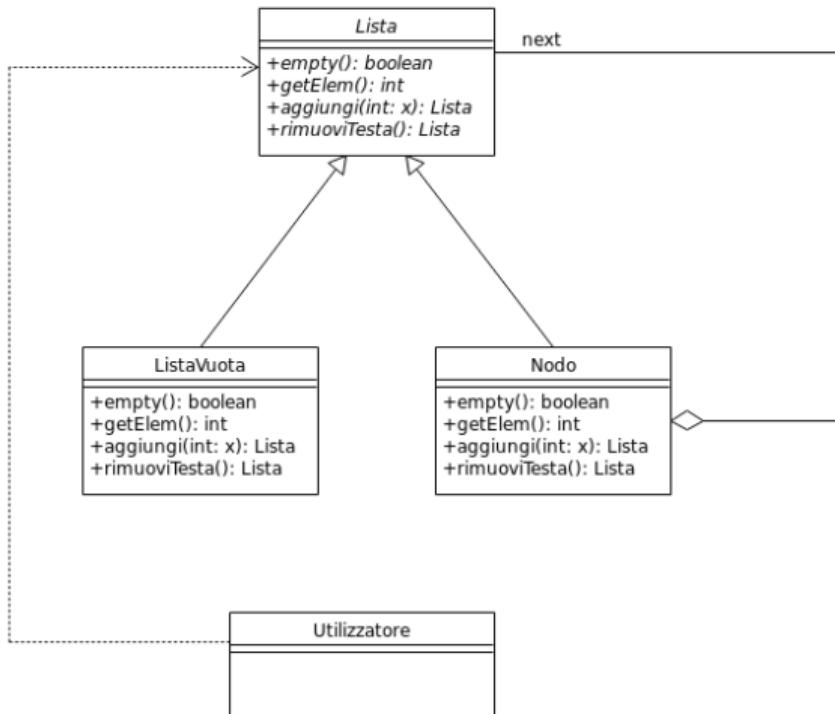


# Applicazione del pattern Visitor

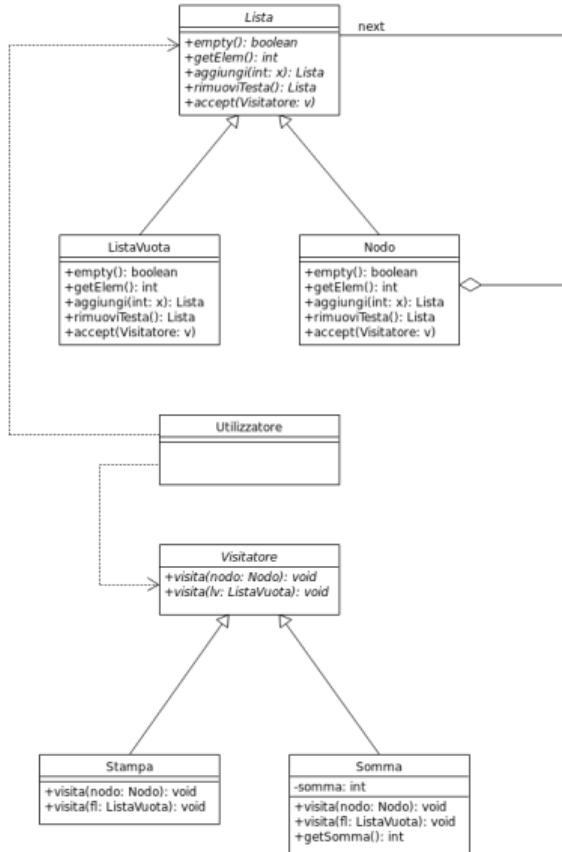
Si veda F. Guidi Polanco. GoF's Design patterns in Java, 2002, pagg. 152–159.



# Applicazione del pattern Visitor



# Applicazione del pattern Visitor



# **12 . Dal progetto al codice**

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2021/22

Università degli Studi di Torino - Dipartimento di Informatica

### **Si noti che**

questi lucidi sono basati sul libro di testo del corso “C. Larman, *Applicare UML e i Pattern*, Pearson, 2016”.

# Table of contents

---

1. Trasformare i progetti in codice
2. Sviluppo guidato dai test e refactoring

## Trasformare i progetti in codice

---

# Dal progetto al codice

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

# Dal progetto al codice

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

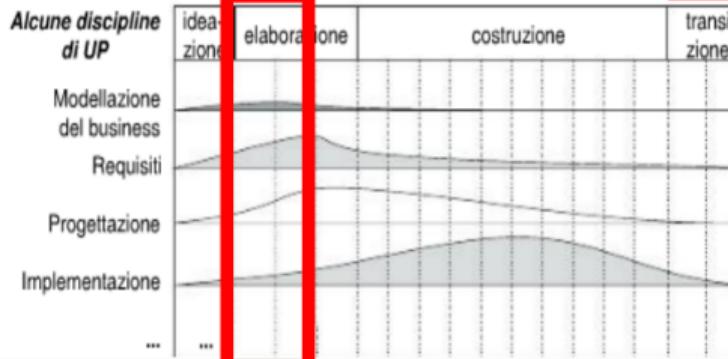
Siamo pronti per la realizzazione del **Modello di Implementazione**, costituito da tutti gli elaborati dell'implementazione, come il **codice sorgente**, la definizione delle **basi di dati**, le pagine JSP/XML/HTML, ecc.

# UP maps

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

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

## Alcune discipline di UP

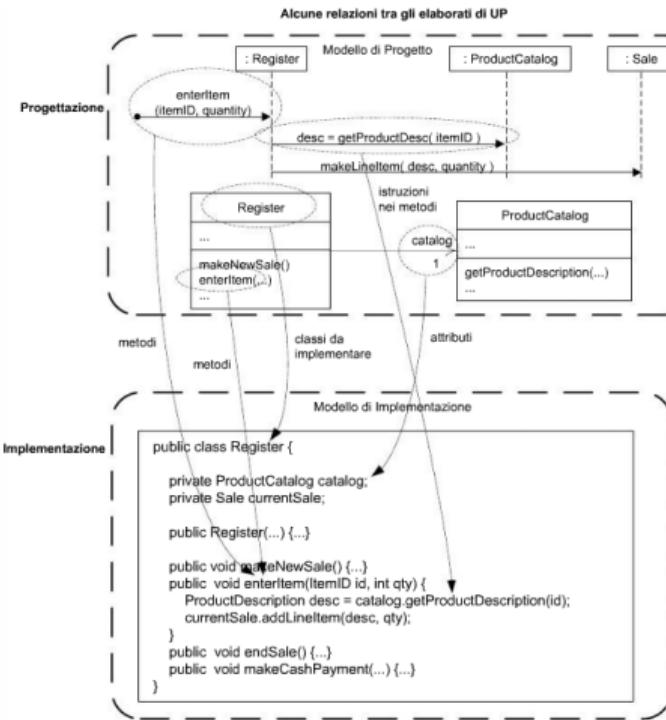


L'impegno relativo nelle discipline cambia a seconda delle fasi.

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

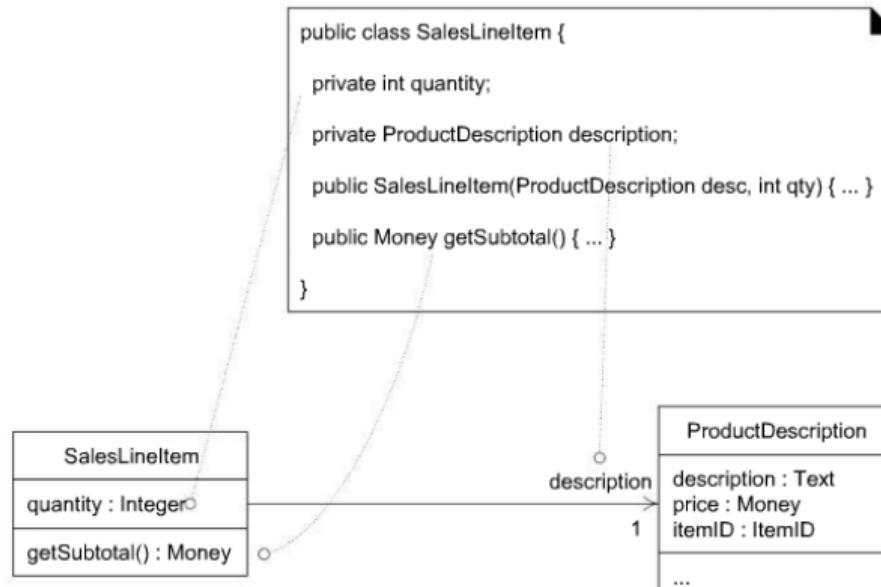
# Dagli elaborati della progettazione a quelli dell'implementazione

- Java è solo un linguaggio di esempio!  
Molti altri linguaggi object-oriented possono essere utilizzati.
- L'implementazione è un processo di traduzione relativamente meccanico. Tuttavia durante la programmazione ci si devono aspettare e si devono pianificare numerosi cambiamenti e deviazioni rispetto al progetto realizzato.
- Questo è un atteggiamento essenziale e pragmatico nei metodi iterativi ed evolutivi.



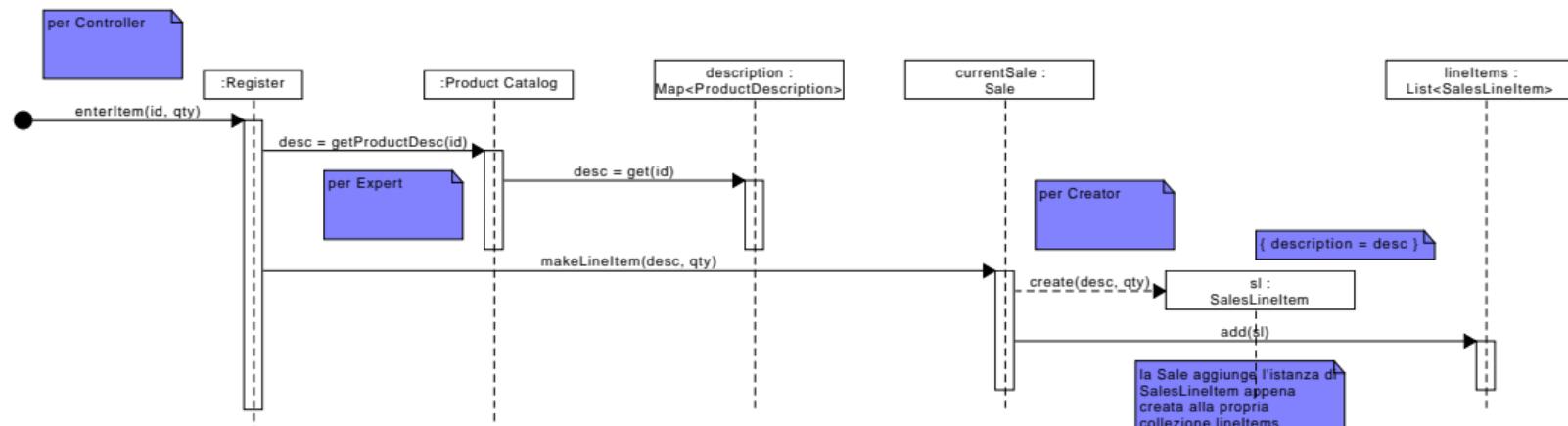
# Definire una classe con gli attributi e le firme dei metodi

La traduzione in termini di definizione di attributi e di firme di metodi è spesso immediata.  
Ecco l'esempio della classe *SalesLineItem*.



# Creare metodi dai diagrammi di interazione

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



# Collezioni



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

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

La definizione dell'attributo *lineItems*.

```
private List<SalesLineItem> lineItems = new ArrayList<>();
```

# Collezioni



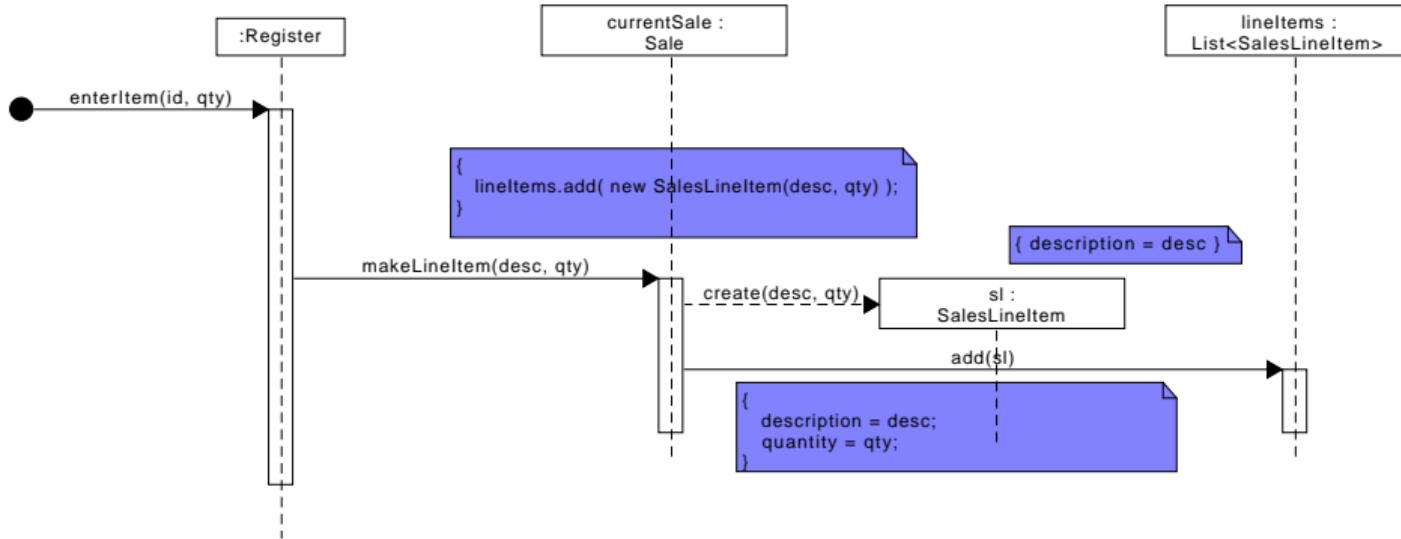
Le relazioni *uno-a-molti* sono implementate di solito con l'introduzione di un oggetto collezione.

La definizione dell'attributo *lineItems*.

```
private List<SalesLineItem> lineItems = new ArrayList<>();
```

Nota: se un oggetto implementa un'interfaccia, si dichiari la variabile in termini dell'interfaccia, non della classe concreta.

## Altro esempio: Sale.makeLineItem

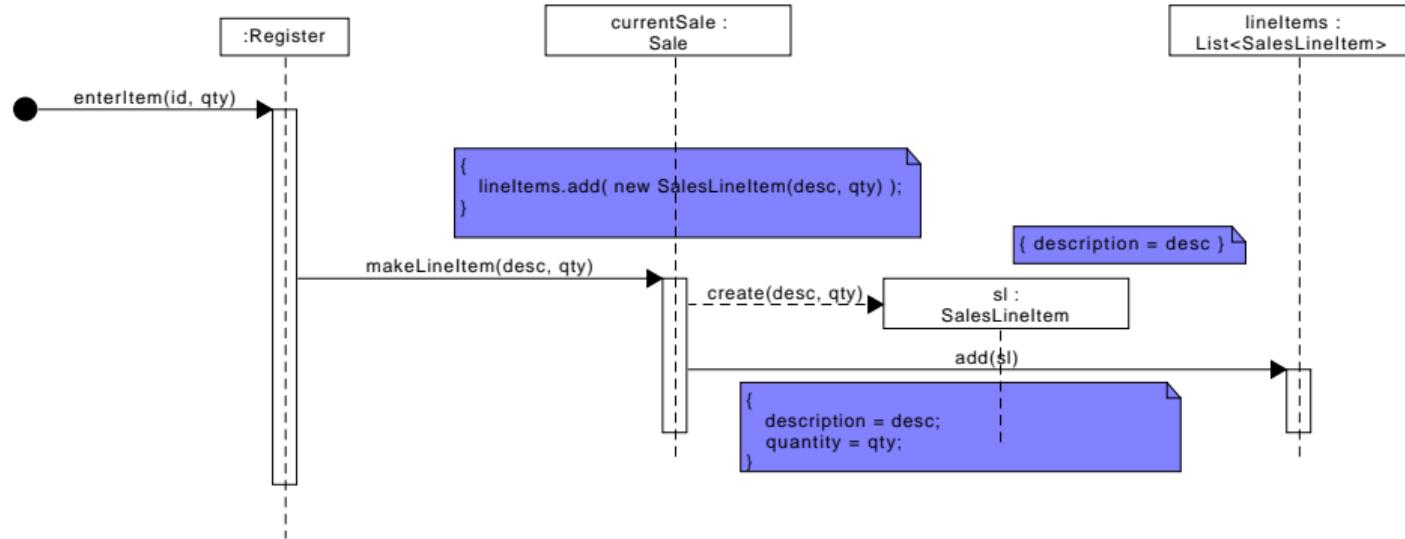


Il metodo `makeLineItem` della classe `Sale` può essere scritto per ispezione del diagramma di collaborazione per `enterItem`.

## Costruttore

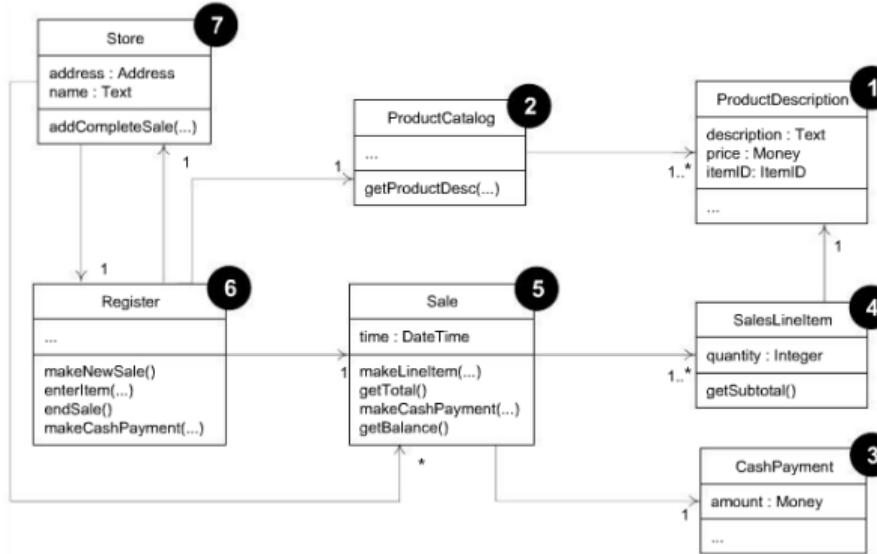
Il *costruttore* della classe *SalesLineItem* è generato, per ispezione, da una versione parziale del diagramma di interazione per *enterItem*.

# Costruttore



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

# Ordine di implementazione



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

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

Si veda libro di testo dalla pagina 428 alla pagina 432 (da 381 a 388 quinta edizione)

# Ordine di implementazione

```
package com.foo.nextgen.domain;

Classe ProductDescription
public class ProductDescription {

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

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

# Ordine di implementazione

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

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

Classe ProductCatalog
public class ProductCatalog {

    private Map<ItemID, ProductDescription> descriptions;

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

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

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

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

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

# Ordine di implementazione

```
Classe Register
public class Register {

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

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

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

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

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

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

Classe Sale
public class Sale {

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

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

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

# Ordine di implementazione

```
public Money getTotal() {
    Money total = new Money(0);
    Money subtotal;
    for (SalesLineItem lineItem : lineItems) {
        subtotal = lineItem.getSubtotal();
        total = total.add(subtotal);
    }
    return total;
}
public void makeCashPayment(Money cashTendered) {
    payment = new CashPayment(cashTendered);
}
public Money getBalance() {
    return payment.getAmount().minus(getTotal());
}
}

Classe SalesLineItem
public class SalesLineItem {

    private int quantity;
    private ProductDescription description;

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

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

Classe CashPayment
public class CashPayment {

    private Money amount;

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

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

# Ordine di implementazione

## Classe Store

```
public class Store {  
  
    private ProductCatalog catalog;  
    private Register register;  
    private List<Sale> completedSales;  
  
    public Store() {  
        catalog = new ProductCatalog();  
        register = new Register(this, catalog);  
        completedSales = new ArrayList<>();  
    }  
  
    public Register getRegister() { return register; }  
  
    public void addSale(Sale s) {  
        completedSales.add(s);  
    }  
}
```

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

## **Sviluppo guidato dai test e refactoring**

---

## Extreme Programming (XP) e test

Extreme Programming ha promosso la pratica dei *test*: scrivere i test per *primi*.

## Extreme Programming (XP) e refactoring

Extreme Programming ha inoltre promosso il *refactoring continuo* del codice per migliorare la qualità: meno duplicazioni, maggiore chiarezza, ecc.

## Test-Driven Development (TDD)

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

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

## Vantaggi dello sviluppo guidato dai test

- I test unitari (ovvero i test relativi a singole classe e metodi) vengono effettivamente scritti
- La soddisfazione del programmatore porta a una scrittura più coerente dei test<sup>1</sup>
- Chiarimento dell'interfaccia e del comportamento dettagliati
- Verifica dimostrabile, ripetibile e automatica
- Fiducia nei cambiamenti

---

<sup>1</sup>Lo sviluppo seguito dai test è noto anche come sviluppo solo per questa volta eviterò di scrivere i test...

## Vantaggi dello sviluppo guidato dai test

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

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

Un metodo di test unitario è logicamente composto da quattro parti:

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

# Esempio di test per il metodo Sale.makeLineItem

```
import static org.junit.Assert.*;
import org.junit.*;

import com.foo.nextgen.domain.*;

public class SaleTest {

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

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

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

## Refactoring

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

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

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

## Refactoring e test

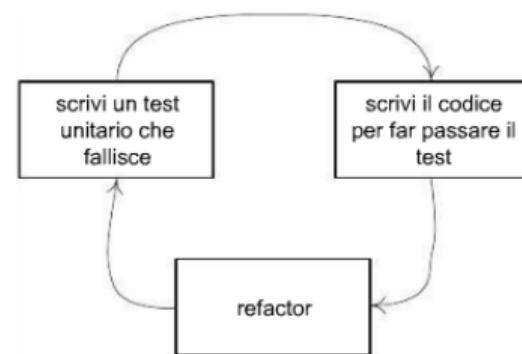
Dopo ciascuna trasformazione, i test unitari vengono eseguiti nuovamente per dimostrare che il refactoring non abbia provocato una *regressione* (un *fallimento*).

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

# Refactoring e test

Regole per il TDD e refactoring:

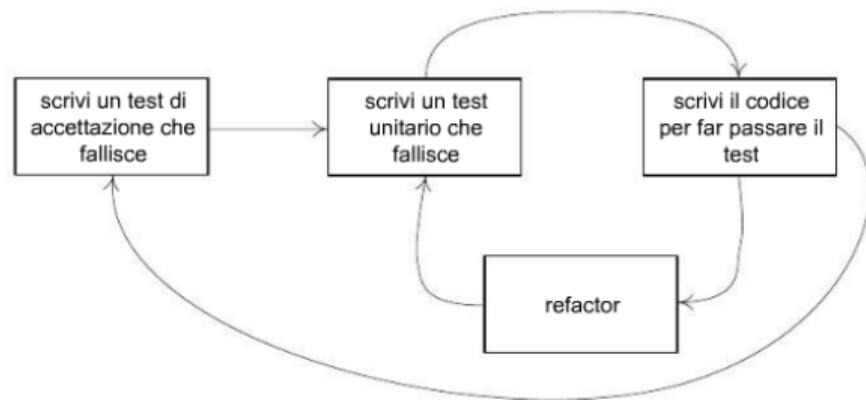
- Scrivi un test unitario che fallisce, per dimostrare la mancanza di una funzionalità o di codice
- Scrivi il codice più semplice possibile per far passare il test
- Riscrivi o ristruttura (refactor) il codice, migliorandolo, oppure passa a scrivere il prossimo test unitario



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

# Refactoring e test

Estensione del ciclo di base del TDD con un doppio ciclo, più ampio. Il ciclo più ampio è relativo ai test di accettazione, ad esempio per un'intera esecuzione di uno specifico caso d'uso.



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

# Obiettivi del refactoring

---

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

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

# Alcuni refactoring

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 new e la chiamata di un costruttore con l'invocazione di un metodo di supporto che crea l'oggetto (nascondendo i dettagli).

# Esempio: refactoring con Extract Method

Prima.

```
public class Player {  
    private Piece piece;  
    private Board board;  
    private Die[] dice;  
    // ...  
  
    public void takeTurn() {  
        int rollTotal = 0;  
        for (int i=0; i<dice.length; i++) {  
            dice[i].roll();  
            rollTotal += dice[i].getFaceValue();  
        }  
        Square oldLoc = piece.getLocation();  
        Square newLoc = board.getSquare(oldLoc, rollTotal);  
        piece.setLocation(newLoc);  
    }  
}
```

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

Dopo.

```
public class Player {  
    private Piece piece;  
    private Board board;  
    private Die[] dice;  
    // ...  
  
    public void takeTurn() {  
        // chiama il metodo ottenuto dal refactoring  
        int rollTotal = rollDice();  
  
        Square oldLoc = piece.getLocation();  
        Square newLoc = board.getSquare(oldLoc, rollTotal);  
        piece.setLocation(newLoc);  
    }  
  
    // metodo di supporto estratto dal refactoring  
    private int rollDice() {  
        int rollTotal = 0;  
        for (int i=0; i<dice.length; i++) {  
            dice[i].roll();  
            rollTotal += dice[i].getFaceValue();  
        }  
        return rollTotal;  
    }  
}
```

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

# Esempio: refactoring con Introduce Explaining Variable

Prima.

```
// buon nome del metodo, ma la logica del corpo non è chiara
public boolean isLeapYear( int year ) {
    return ( year % 400 ) == 0 ||
           ( ( year % 4 ) == 0 && ( year % 100 ) != 0 );
}
```

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

Dopo.

```
// così va meglio
public boolean isLeapYear( int year ) {
    boolean isFourthYear = ( year % 4 ) == 0;
    boolean isHundredthYear = ( year % 100 ) == 0;
    boolean is4HundredthYear = ( year % 400 ) == 0;

    return is4HundredthYear || ( isFourthYear && ! isHundredthYear );
}
```

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