



# 10 . General Responsibility Assignment Software Patterns (GRASP)

Sviluppo di Applicazioni Software

---

Matteo Baldoni

a.a. 2023/24

Università degli Studi di Torino - Dipartimento di Informatica

## Attenzione!



©2024 Copyright for this slides by Matteo Baldoni. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

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

## Si noti che

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

# Table of contents

1. Responsabilità
2. Responsibility-Driven Development
3. General Responsibility Assignment Software Pattern
4. I nove pattern GRASP

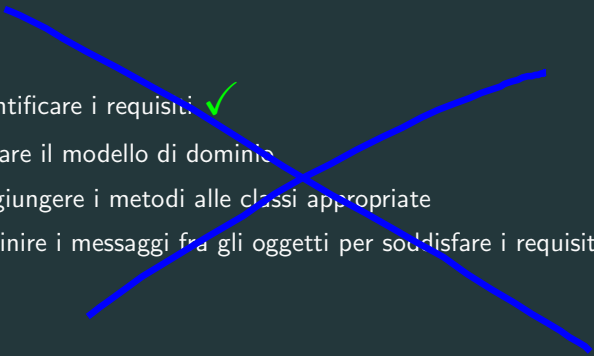
# Responsabilità

---

# 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

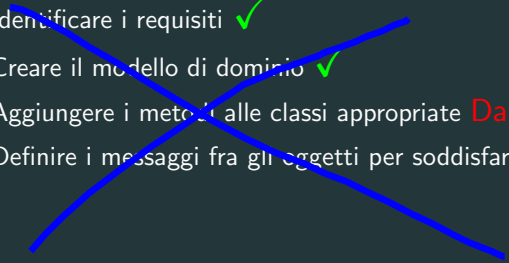
# Progettazione Object-Oriented (OODesign) e responsabilità

- 
- 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!

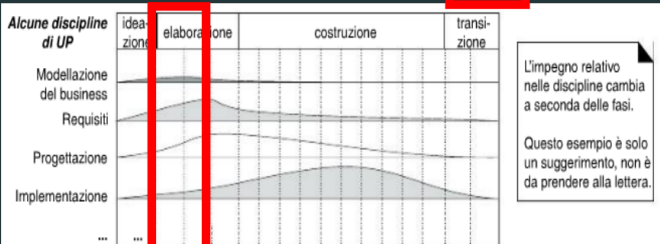
## Martin Fowler

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. Et...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 Specifica	i	r		
		Supplementare	i	r		
Progettazione	modellazione agile sviluppo guidato dai test	Modello di Progetto		i	r	
		Documento dell'Architettura Software		i		
		Modello dei Dati		i	r	
	programmazione a coppie integrazione continua standard di codifica					
Gestione del progetto	gestione del progetto agile riunioni Scrum giornaliere	...				
...						



## Dove siamo? Sempre alla prima iterazione...

Il primo <i>workshop dei requisiti della durata di due giorni</i> è finito.	Il chief architect e il responsabile dell'organizzazione concordano di implementare e testare alcuni <i>scenari del caso d'uso Elabora Vendita nella prima</i> iterazione timeboxed <i>di tre settimane</i> .
<i>Tre dei venti casi d'uso</i> , 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 <i>Elabora Vendita</i> (UP raccomanda la prassi tipica dei metodi iterativi, cioè analizzare solo il <b>10%-20% dei requisiti</b> in modo dettagliato prima di iniziare a programmare).	<i>Altri elaborati</i> sono stati iniziati: Specifiche Supplementari, Glossario e Modello di Dominio.
<i>Esperimenti di programmazione</i> hanno risolto le domande tecniche più eclatanti, per esempio se le UI Java Swing funzionano con i monitor touch screen.	Il chief architect ha disegnato <i>alcune idee per l'architettura logica su larga scala</i> , 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 Suppletive</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*.

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)**.

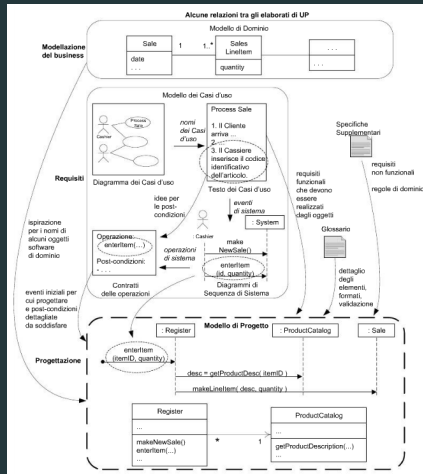


# Output della progettazione a oggetti

**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**

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

Le responsabilità di conoscere di un oggetto comprendono:

- conoscere i propri dati privati incapsulati
- 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.



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**

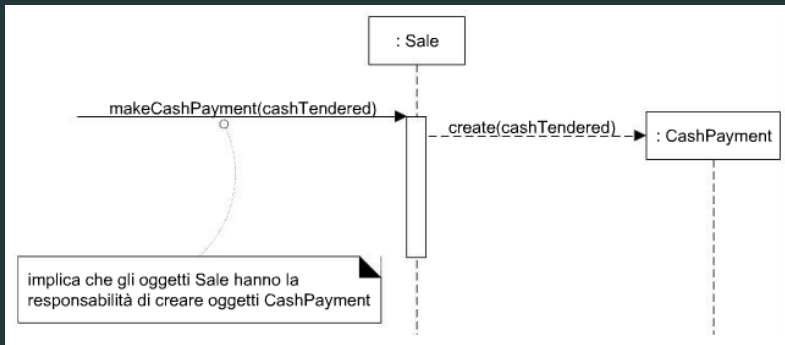
---

# 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

# GRASP e UML

- 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).



- 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 nove 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

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

# Progettazione modulare

## Progettazione modulare

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

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


## Progettazione modulare

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

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*?

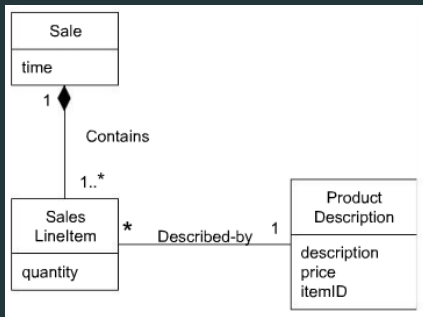
Secondo Creator, si deve cercare una classe che aggrega, contiene, istanza di *SalesLineItem*.

# 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*.

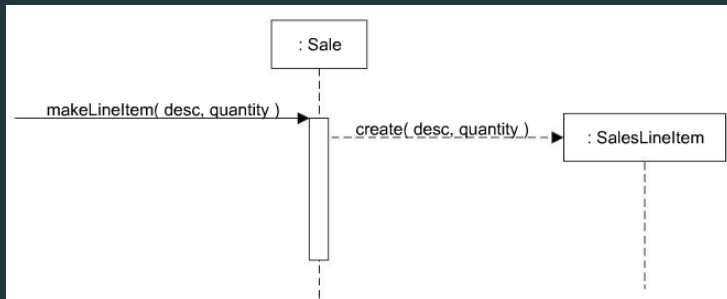
Un oggetto composto è un ottimo candidato per creare le sue parti. Quindi è *Sale* un buon candidato ad avere la responsabilità di creare istanze di *SalesLineItem*.

## 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 Suppletive (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.

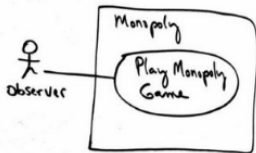


Figura 7.6 Diagramma dei casi d'uso ("diagramma di contesto") per il sistema Monopoly.

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:**

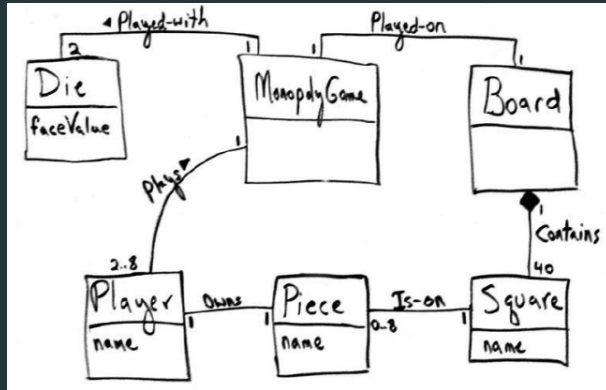
- Fornire per la traccia di gioco sia una modalità grafica che una modalità testuale.



## Altro esempio per Creator

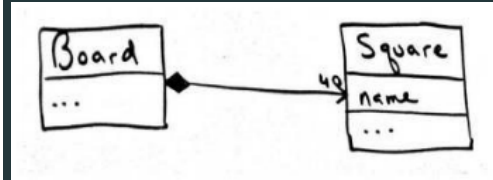
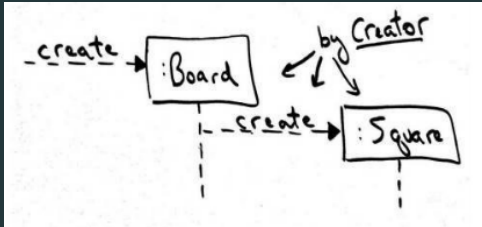
Si consideri l'esempio del gioco *Monopoly* (si veda libro di testo, Sez. 7.21).

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, a **l'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, ...

Nell'applicazione POS NextGen, qualche classe ha bisogno di conoscere il totale complessivo di una vendita

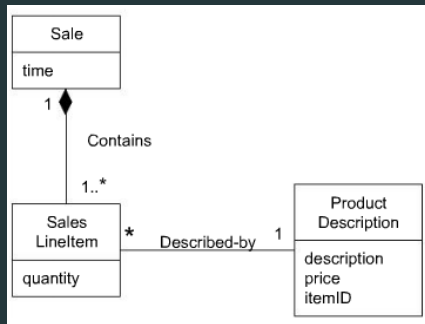
**Enunciare la responsabilità in modo chiaro!**

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.



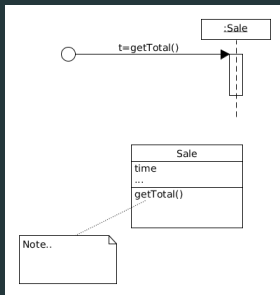
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

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).



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

Nota: quello riportato è un diagramma delle interazioni e delle classi parziali.



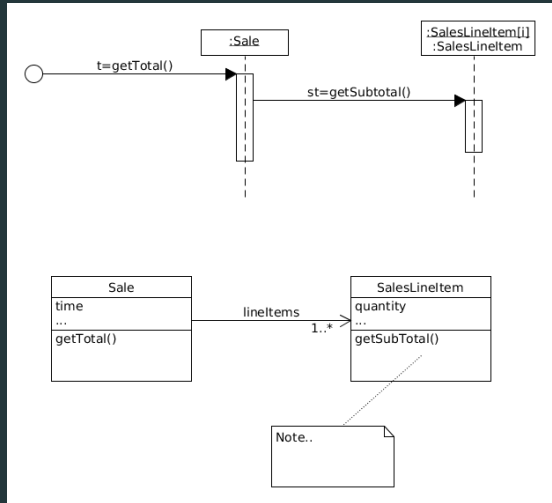
## Esempio per Expert

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

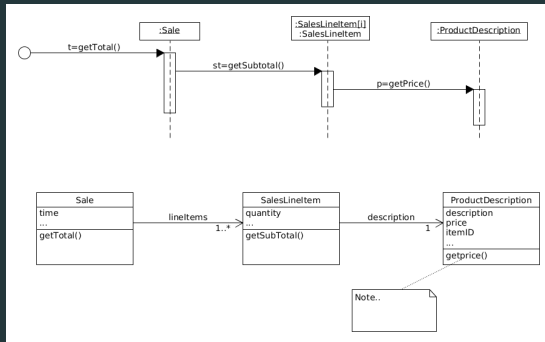
## Esempio per Expert

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* il messaggio *getPrice* chiedendogli il prezzo del prodotto.

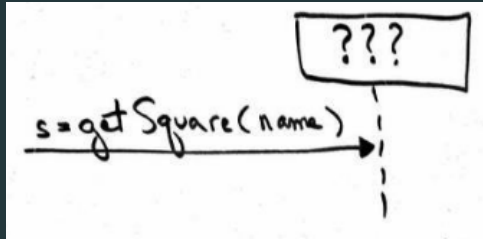


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

## Altro esempio per Expert

Si consideri l'esempio del gioco *Monopoly* (si veda libro di testo, Sez. 7.21).

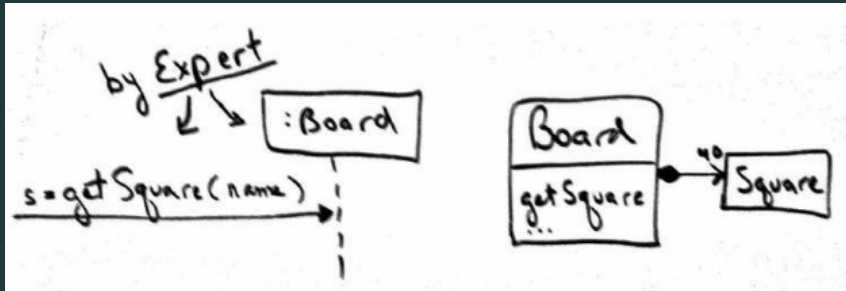
Si supponga che alcuni oggetti debbano essere in grado di trovare una particolare *Square*, dato il suo nome unico. Chi deve essere responsabile di conoscere una *Square*, dato il suo identificatore?



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

## 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.

- 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?

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

---

# 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

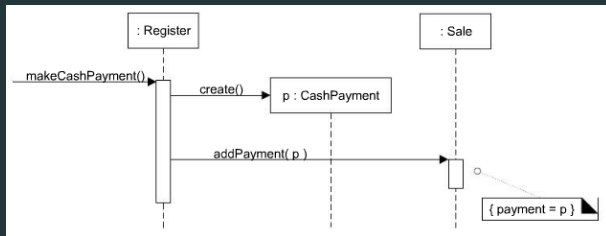
Una classe con un accoppiamento alto (o forte) dipende da molte altre classi. Tali classi fortemente accoppiate possono essere inopportune, e alcune di esse presentano i seguenti problemi:

- 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 ri usare**, 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*



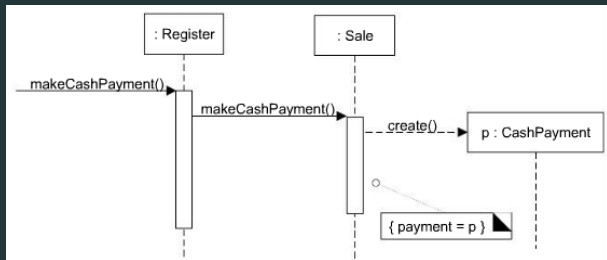
Per determinare gli accoppiamenti si contano le **dipendenze** che sono **direzionali**:

1. Register --> 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.



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

Per determinare gli accoppiamenti si contano le **dipendenze** che sono **direzionali**:

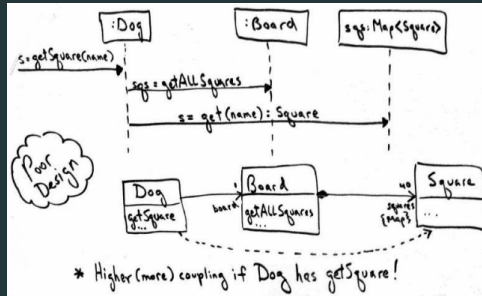
1. Register --> Sale
2. Sale --> CashPayment

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**

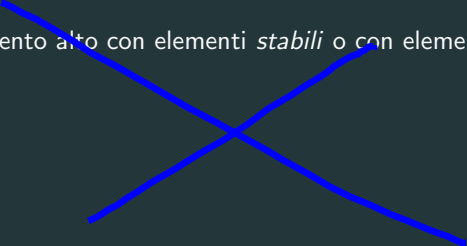
## Low Coupling: osservazioni

- 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

---

# 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.

# 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.**

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

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

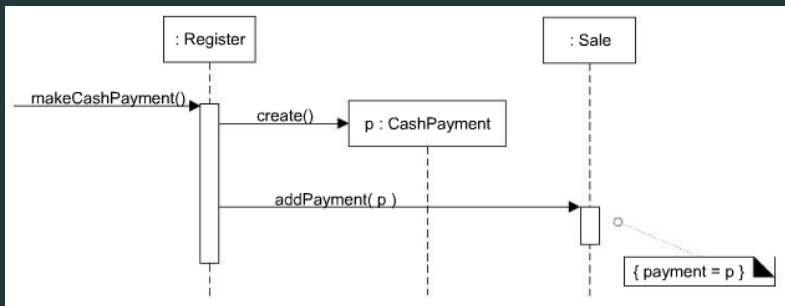
Le classi a coesione bassa spesso rappresentano un'astrazione a “grana molto grossa” o hanno assunto responsabilità che avrebbero dovuto essere delegate ad altri oggetti.



## Esempio per High Coesion

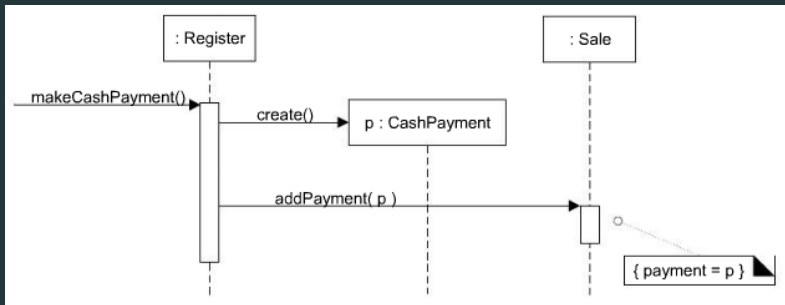
Siano *CashPayment*, *Register* e *Sale* tre classi di progetto dell'applicazione POS NextGen:

- 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 Coesion

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

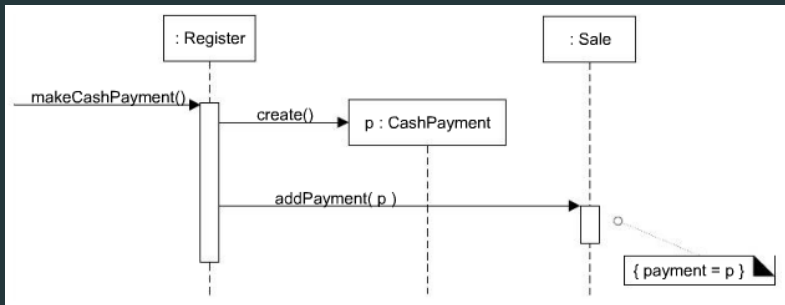


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

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 Coesion

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

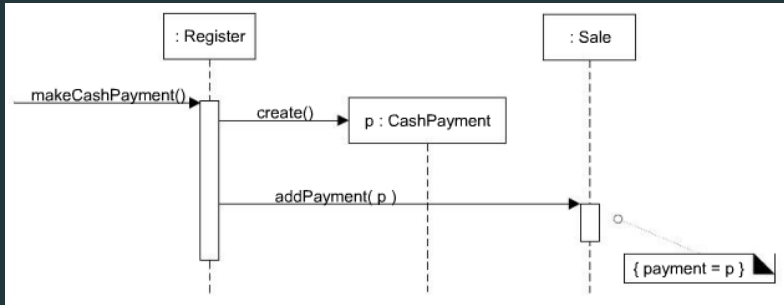


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

Se si continua a rendere la classe **Register** responsabile di eseguire una parte del lavoro o l'intero lavoro relativo a sempre più operazione di sistema, essa diventerà sempre più carica di compiti, e diventerà **non coesa**.

## Esempio per High Coesion

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



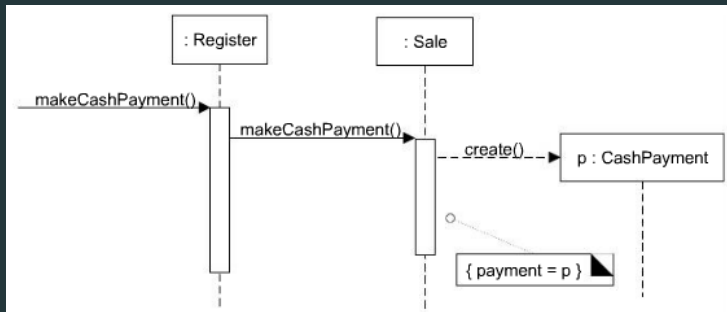
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

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 Coesion

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.

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

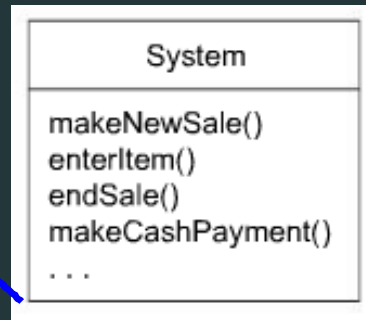
Nota: **le classi dello strato UI** (le classi “*finestre*”, “*viste*” e “*documenti*”) non sono Controller, **non devono svolgere i compiti associati agli eventi di sistema**, ma normalmente ricevono questi eventi e li delegano a un controller.

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.

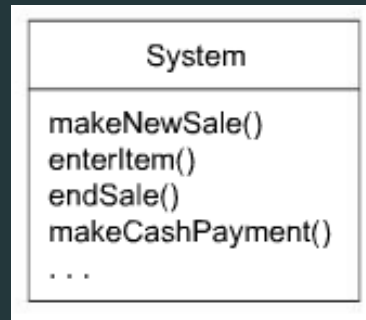


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# 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.

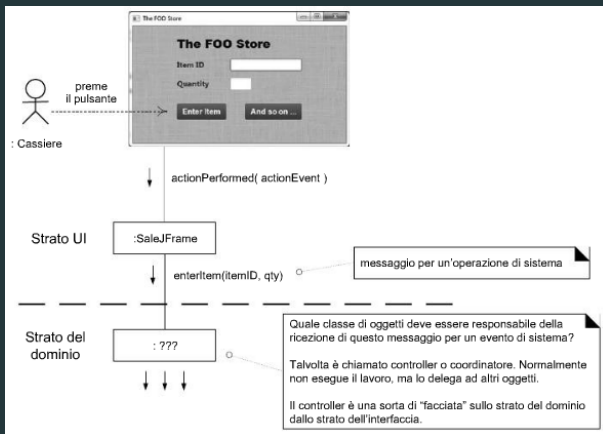


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# 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*?

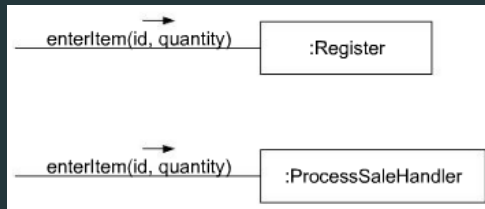


# Un esempio di Controller

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

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

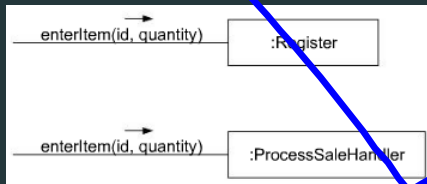
**Due possibili soluzioni.**



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

# Un esempio di Controller

Due possibili soluzioni:

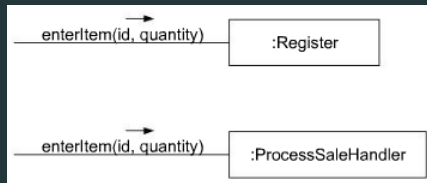


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

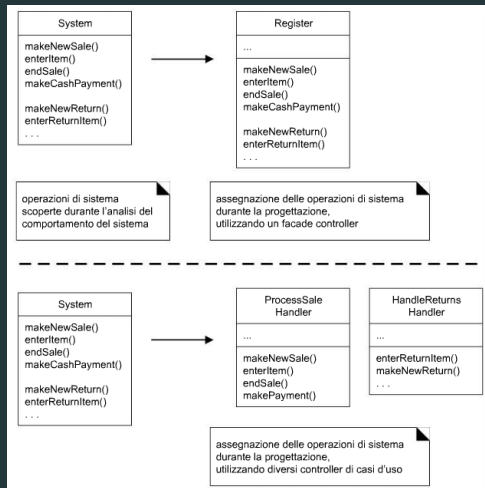


# Un esempio di Controller

Due possibili soluzioni:



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



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

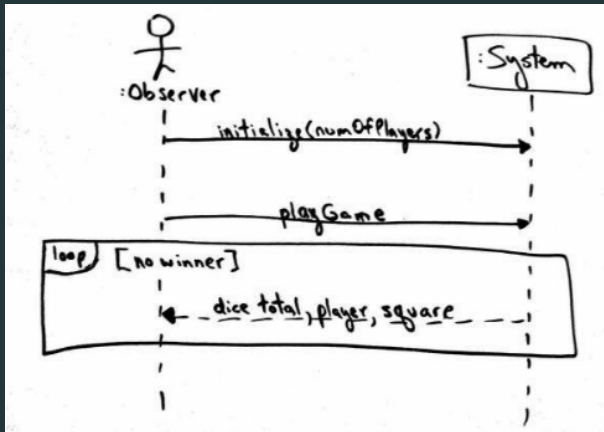
## Un altro esempio di Controller

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.

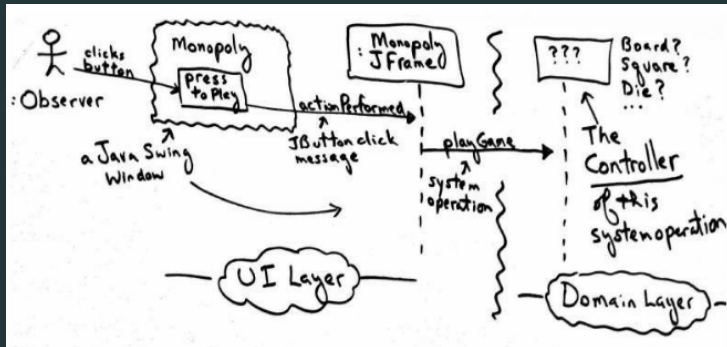
Nell'esempio, chi gestisce *playGame*?



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

## Un altro esempio di Controller

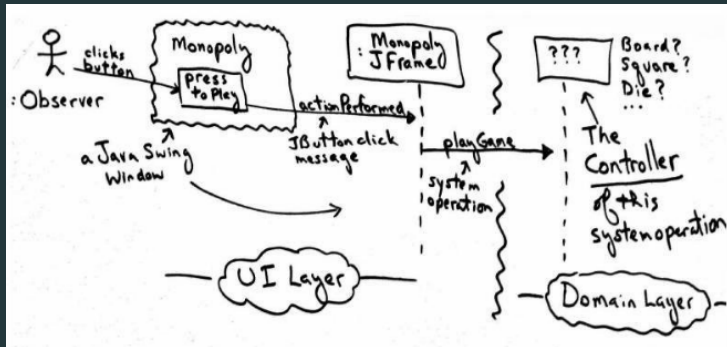
La finestra *JFrame* deve adattare quel messaggio *actionPerformed* a qualcosa di semanticamente più significativo, come un messaggio *playGame* (per corrispondere all'analisi del SSD), e delegare il messaggio *playGame* a un oggetto di dominio nello strato del dominio.



©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?*



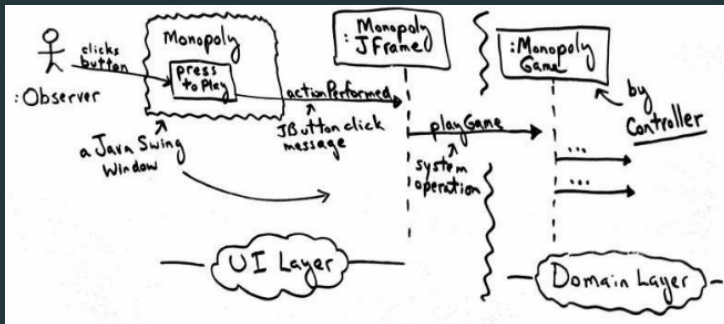
©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

# 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.*

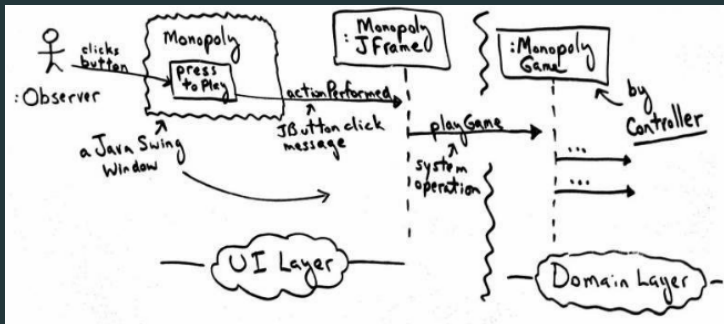


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

## 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).

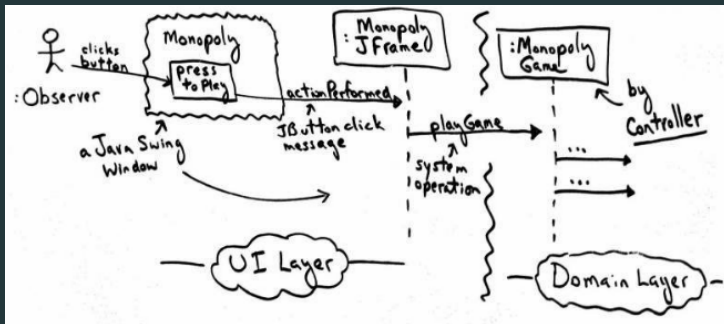


©C. Larman. Applicare UML e i Pattern. Pearson, 2016.

## 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 “**faccia**” **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.

### 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.

Entrambi 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.

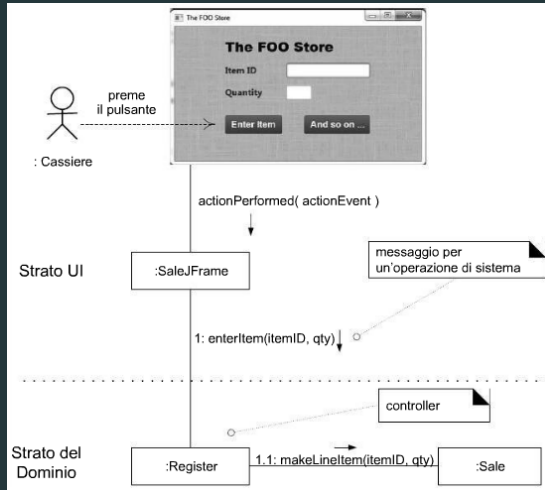
# Controller: osservazioni

- (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
```

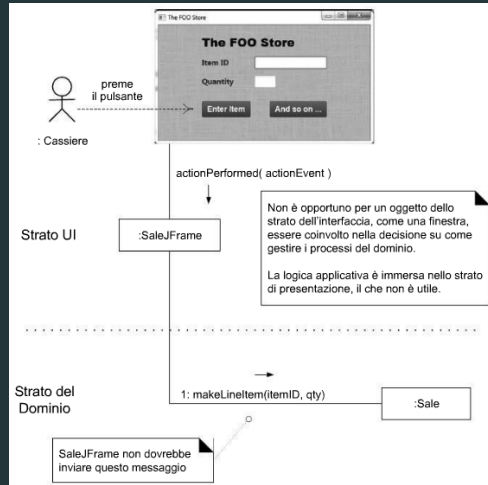
# Controller: osservazioni

Accoppiamento accettabile dallo strato UI allo strato del dominio.

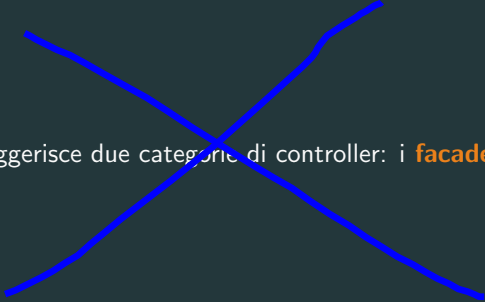


# Controller: osservazioni

Accoppiamento meno opportuno dello strato dell'interfaccia allo strato del dominio.



Il pattern Controller suggerisce due categorie di controller: i **facade controller** o i **controller di caso d'uso**.



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 **o 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).