Design pattern elementari (schemi di progettazione oo)

Prof. Paolo Ciancarini
Corso di Ingegneria del Software
CdL Informatica
Università di Bologna

Obiettivi della lezione

- Introduzione ai design pattern
- La progettazione di oggetti e responsabilità
- GRASP: alcuni pattern elementari
- I principi SOLID

Riferimento:

C.Larman, *Applying UML and Patterns*, Chapter 17

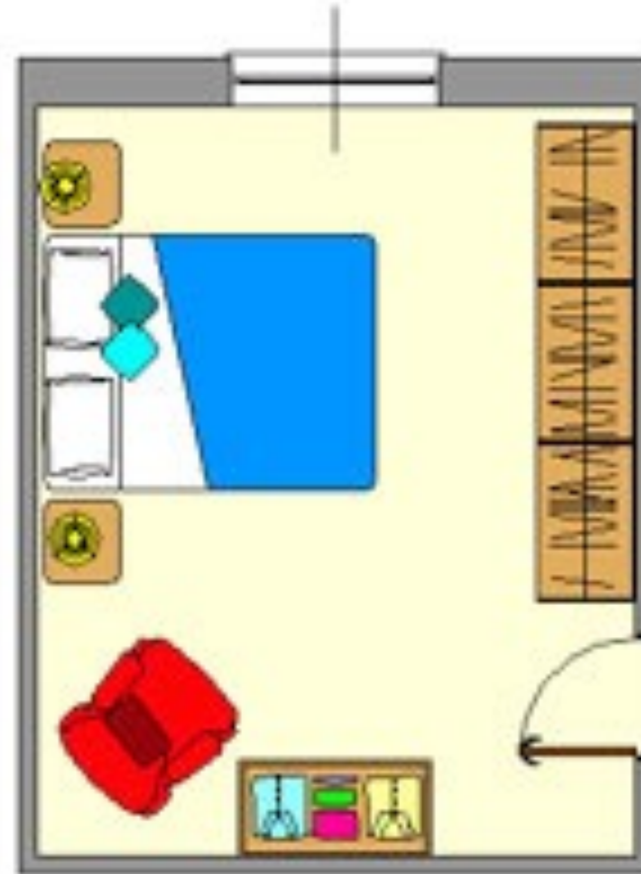
Un lungo corridoio

Una stanza di passaggio?

- Questa stanza ti sembra accogliente?
- Ci staresti volentieri?
- Perché?
  - Luce
  - Proporzioni
  - Simmetria
  - Mobilio
  - altro…

Esempio: da http://blog.lavorincasa.it/?cat=12

- In camera da letto può essere utile disporre di una poltrona, magari nei pressi della **finestra**, per poter leggere
- Quando le dimensioni sono regolari, la soluzione più opportuna è quella di porre la poltrona in un angolo. Se la finestra è lontana bisogna prevedere la presenza di una **lampada**



Un pattern di architettura d’interni

Schema descrittivo di un design pattern

Contesto situazione problematica di progetto

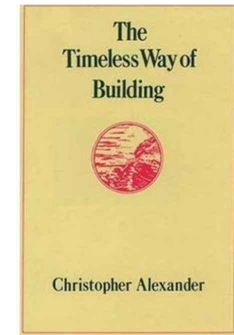
Problema insieme di *forze in gioco* in tale contesto

Soluzione forma o regola da applicare per risolvere tali forze

Esempio – punto luce

- forze
  - **Vogliamo star seduti e comodi in una stanza**
  - **Saremo attirati dalla luce**
- soluzioni
  - In soggiorno, almeno una finestra dev’essere un “punto luce”
  - Se una poltrona è lontana da un punto luce occorre crearne uno

Christopher Alexander

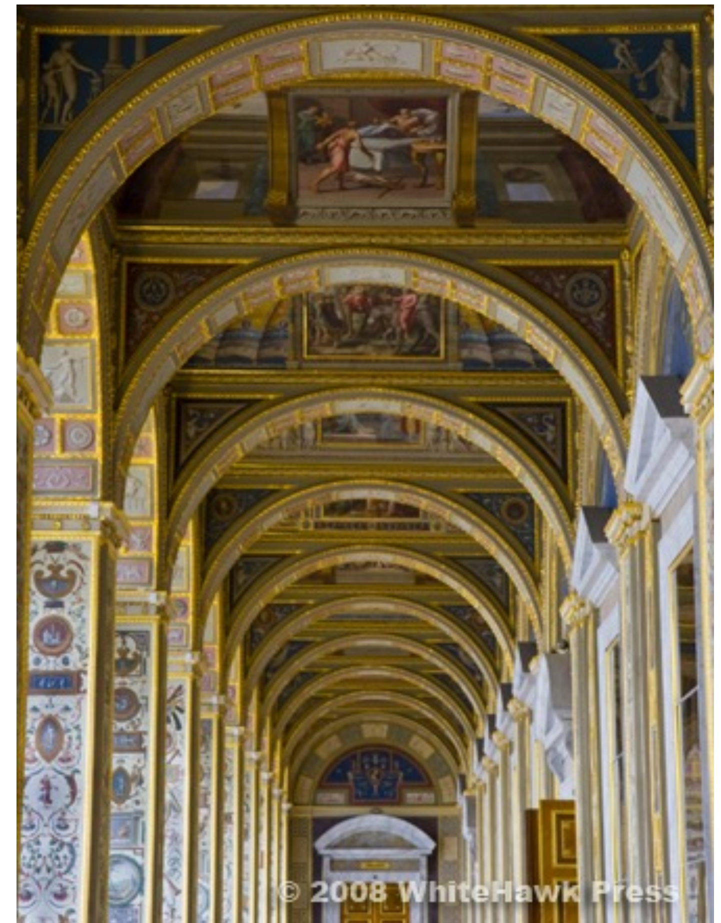


- L'architetto C. Alexander introdusse l'idea di **linguaggio di pattern** nel libro *The Timeless Way of Building*
- Un linguaggio di pattern è un insieme di regole che costituisce un vocabolario di progetto in un certo ambito o dominio
- Esempio: In un altro libro, *A Pattern Language*, Alexander introduce un linguaggio speciale con sottolinguaggi per progettare città (per urbanisti) edifici (per architetti), e costruzioni in genere (per costruttori).
- Esempi:
  - **Schemi per città** Rotatorie, casette a schiera, quartieri per vita notturna
  - **Schemi per edifici** Tetti giardino, luce solare in interno, alcove
  - **Schemi costruttivi** Cemento armato, colonne, ecc.

http://www.patternlanguage.com

Design patterns in architettura

- Un pattern è una soluzione ricorrente ad un problema standard, in un contesto
- "A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
- *Perché rilevanti per il software?*

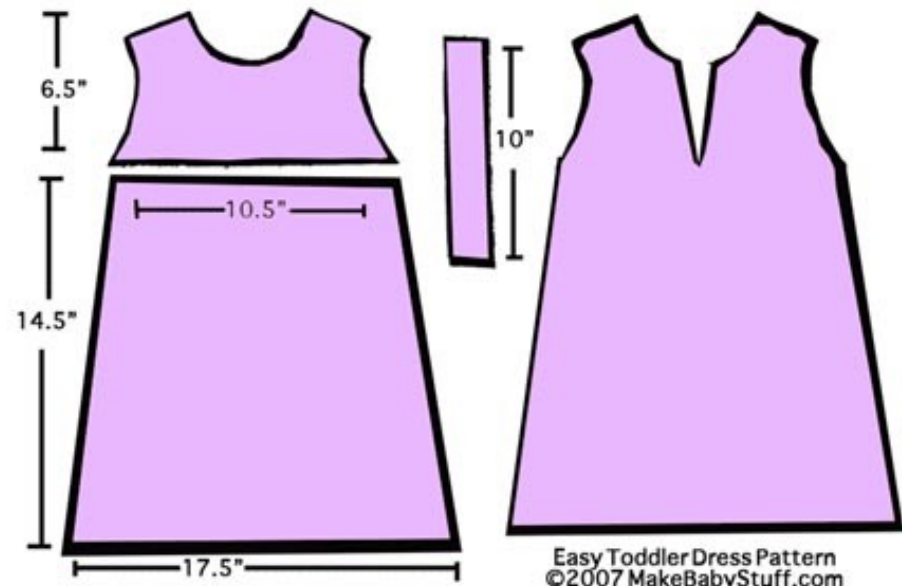


Schemi per vestiti

- Jim Coplein, ingegnere del software:

"*I like to relate this definition to dress patterns …*

I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea



what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself."

Cos' è un design pattern?

Descrizione di un problema frequente e nucleo delle sue possibili soluzioni

Una soluzione di un tipico problema di progettazione

Cos’è un design pattern

“Ogni pattern descrive un problema che compare di continuo nel nostro ambito,

e quindi descrive il nucleo di una soluzione a tale problema,

in modo che si possa usare tale soluzione un milione di volte,

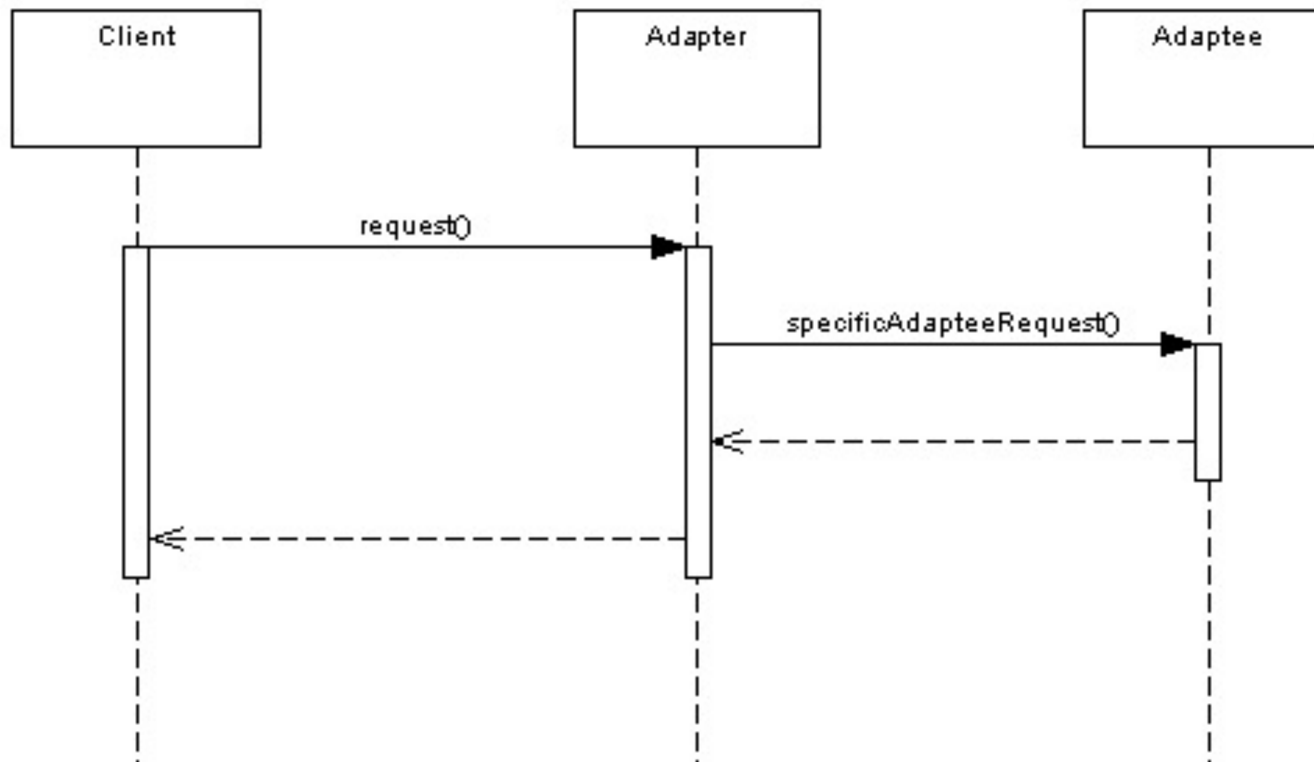
senza mai farlo allo stesso modo”

(C. Alexander e altri, *“A Pattern Language: Towns/Buildings/ Construction*”, Oxford University Press, New York, 1977)

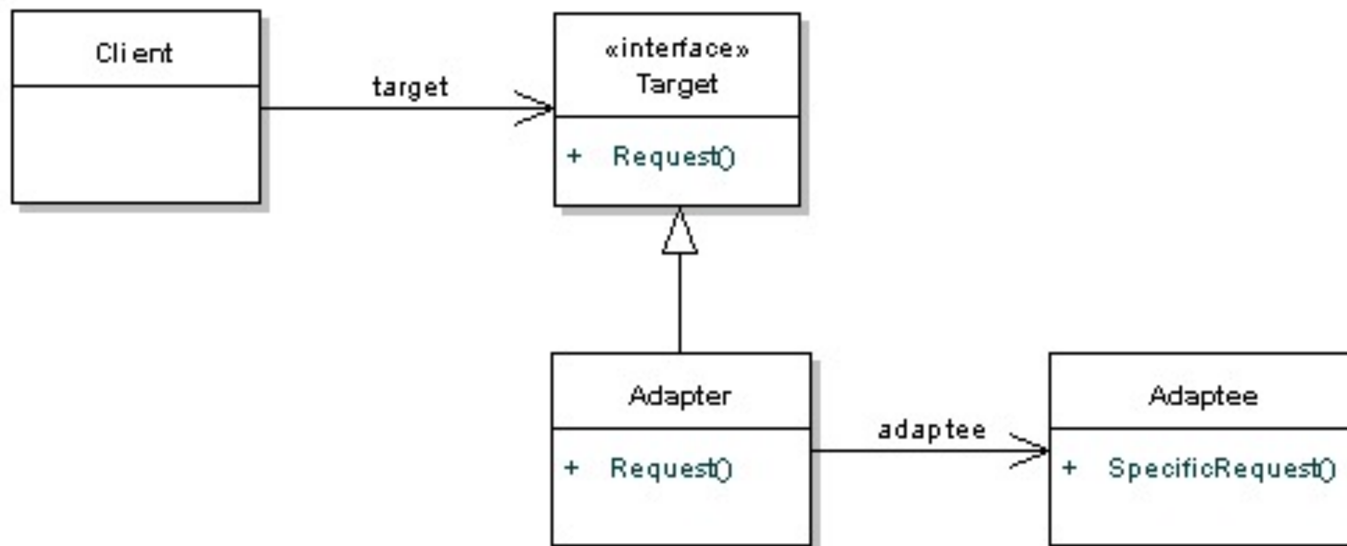
Design pattern nel software

- Un pattern descrive un problema che capita spesso, assieme ad una soluzione consigliata e comprovata del problema
- Ogni pattern si presenta come descrizione di relazioni tra classi o tra oggetti per risolvere un problema progettuale in uno specifico contesto
- Esempio: *adapter*, un pattern che permette l'interazione tra oggetti con interfaccia incompatibile

Adapter (comportamento)



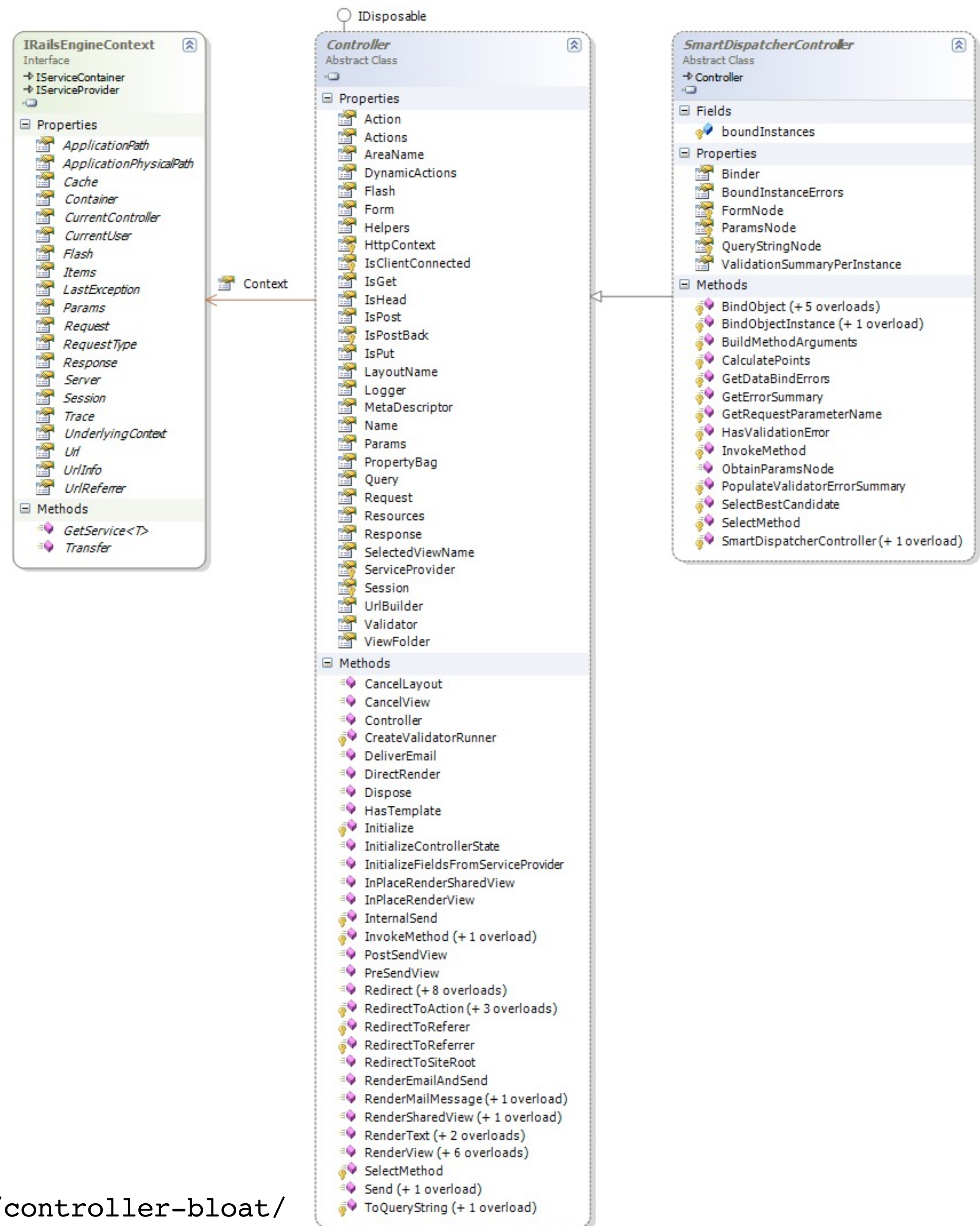
Adapter (struttura)



Anti pattern

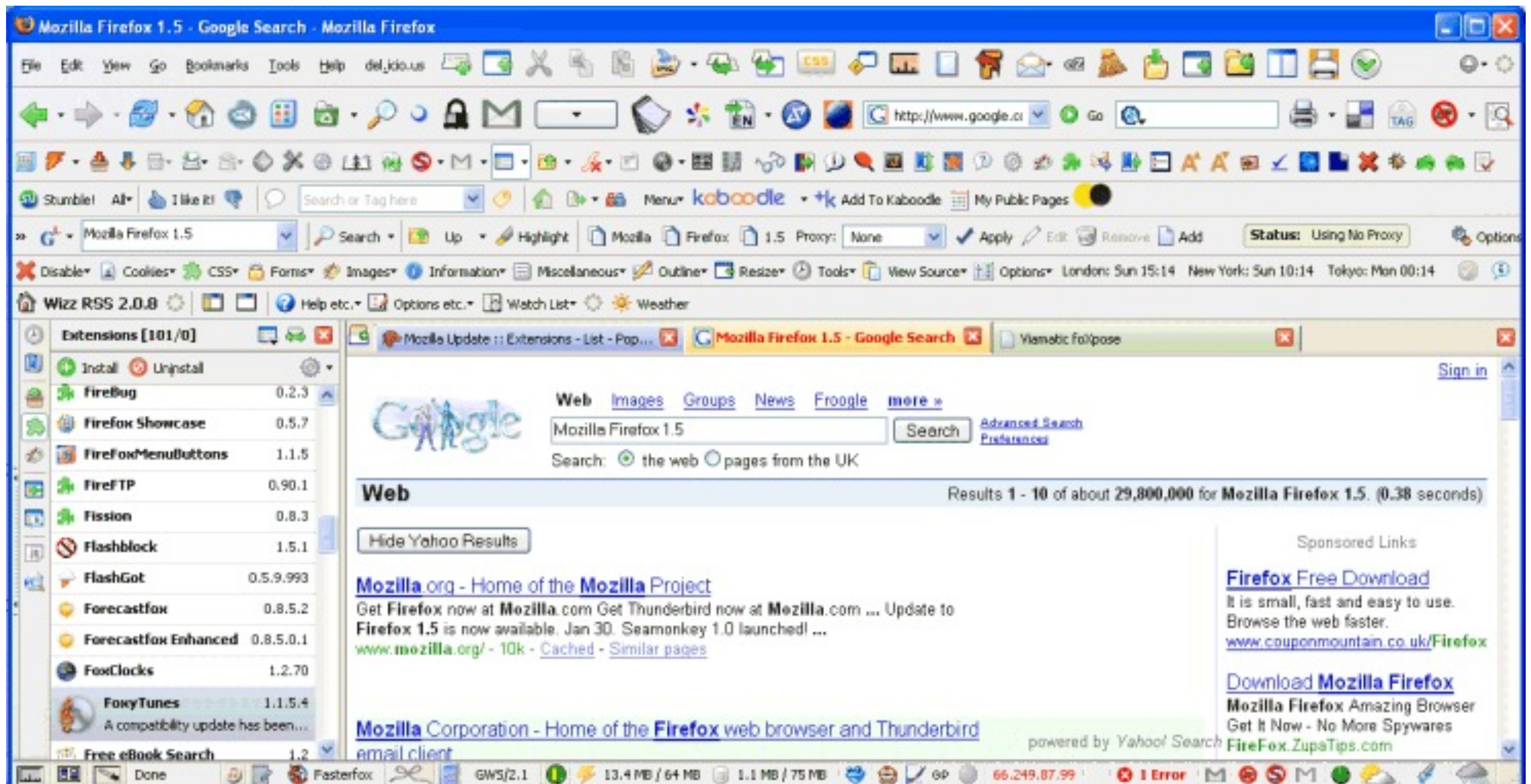
- Oltre ai pattern utili esistono anche gli anti-pattern, che descrivono situazioni ricorrenti e soluzioni notoriamente dannose
- Esempio: *Interface bloat*, che consiste nel rendere un’interfaccia così potente da renderla impossibile da implementare (o usare!)

Interface bloat



lostechies.com/jimmybogard/2008/03/20/controller-bloat/

Interface bloat

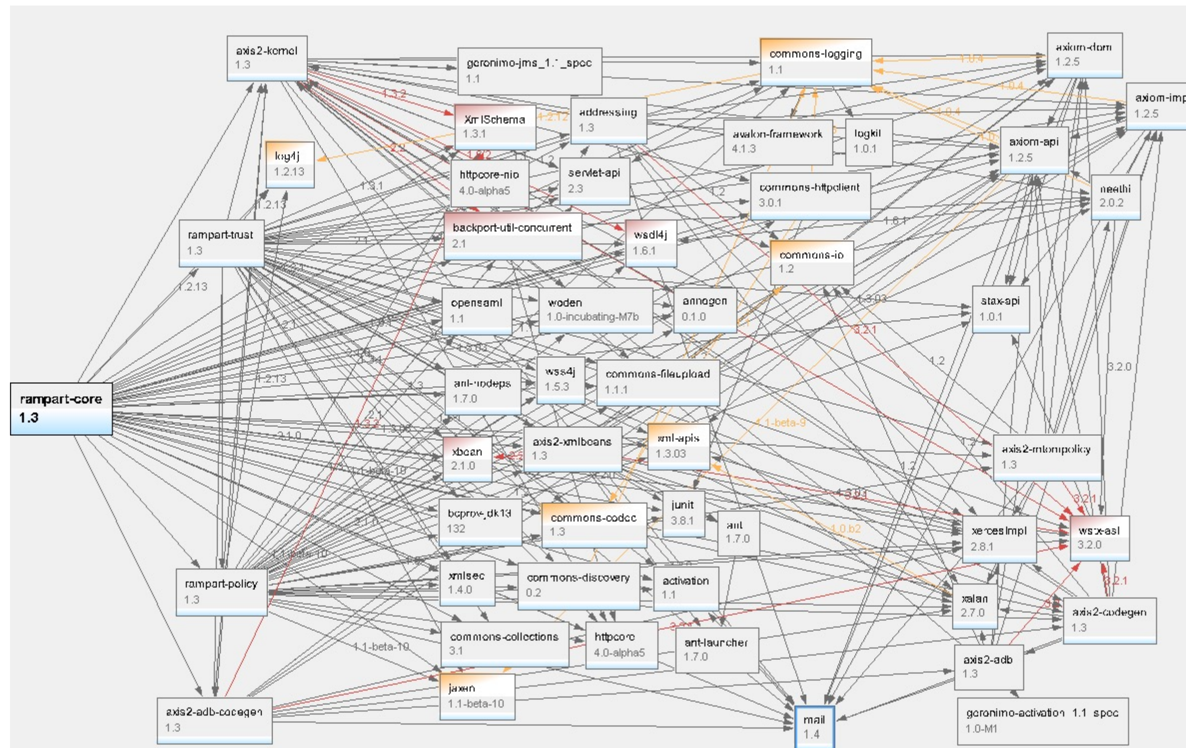


Discussione

Che cos’ è un “tipico problema di progettazione sw”?

Dipendenze tra oggetti

- **Information hiding**: nascondere l’implementazione mostrando solo l’interfaccia – il suo scopo è diminuire le dipendenze tra oggetti…



https://www.taringamberini.com/it/blog/java/controllore-di-vulnerabilita-di-dipendenze-java-pronto-all-uso/

Come si progetta il software?

- Tutti conosciamo
  - Strutture dati
  - Algoritmi
  - Classi
- Quando si descrive un software, le strutture dati, le classi e gli algoritmi formano il vocabolario progettuale
- Esistono però livelli più alti di design

Schemi progettuali

- *Come usa i pattern chi progetta edifici o macchine?*
  - Le discipline più mature hanno manuali che descrivono soluzioni a problemi noti. Esempio: Manuale Colombo
  - Chi progetta una Ferrari o un ponte non inizia dalle leggi della fisica: si riusano schemi che hanno funzionato in passato, e poi si impara dall'esperienza
- *Perché chi progetta software dovrebbe usare i pattern (e i linguaggi di pattern)?*
  - È costoso progettare da zero
  - I pattern promuovono il riuso di soluzioni bune comprovate
  - I linguaggi di pattern abilitano l'idea di *architetture software,* cioè strutture riusabili di componenti

Motivazioni

- I pattern permettono di sfruttare l'esperienza collettiva dei progettisti esperti
- Catturano esperienze reali e ripetute di sviluppo di sistemi sw, e promuovono buone prassi progettuali
- Ogni pattern riguarda un problema specifico e ricorrente che capita "spesso" quando si implementa un sistema software
- I pattern possono guidare il refactoring
- I pattern si possono combinare per costruire architetture software con proprietà specifiche

Definizioni

- Alexander: “Un pattern è una soluzione ricorrente ad un problema standard, in un contesto”
- Larman: “Un *pattern* è la descrizione di un problema; il pattern ha un nome; la soluzione si può applicare in nuovi contesti. Il pattern consiglia come applicare la soluzione in circostanze diverse e considera le forze in gioco e i compromessi”
- Queste definizioni in cosa si somigliano?
- Ed in cosa differiscono?

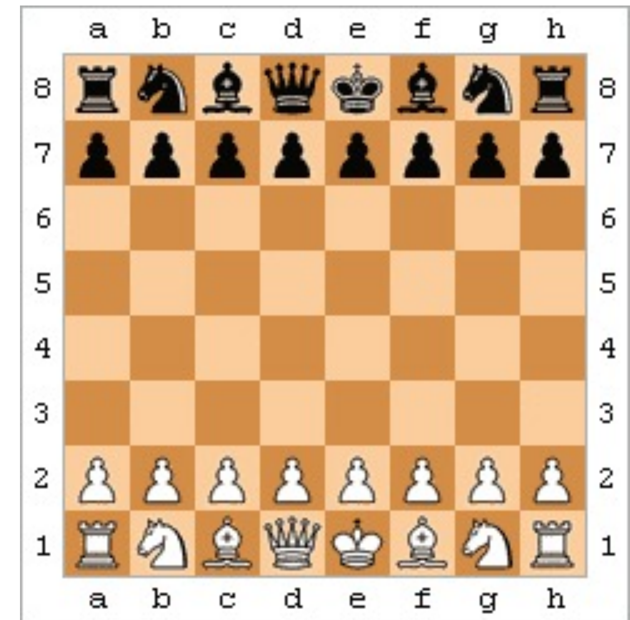
I nomi dei pattern sono importanti!

- Gli schemi progettuali del software hanno nomi suggestivi:
  - Observer, Singleton, Strategy …
- Perché i nomi sono importanti?
  - Supportano il ***chunking***
  - ovvero fissano il concetto nella nostra memoria e ci aiutano a capirlo
  - Facilitano la comunicazione tra progettisti

Star and Plume Quilt

I nomi dei pattern sono importanti!

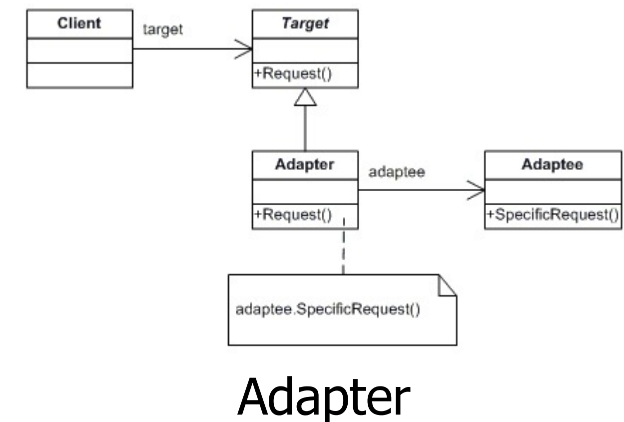
- Gli schemi progettuali del software hanno nomi suggestivi:
  - Observer, Singleton, Strategy …
- Perché i nomi sono importanti?
  - Supportano il ***chunking***
  - ovvero fissano il concetto nella nostra memoria e ci aiutano a capirlo
  - Facilitano la comunicazione tra progettisti



Matto del barbiere

I nomi dei pattern sono importanti!

- Gli schemi progettuali del software hanno nomi suggestivi:
  - Observer, Singleton, Strategy …
- Perché i nomi sono importanti?
  - Supportano il ***chunking***
  - ovvero fissano il concetto nella nostra memoria e ci aiutano a capirlo
  - Facilitano la comunicazione tra progettisti



Adapter

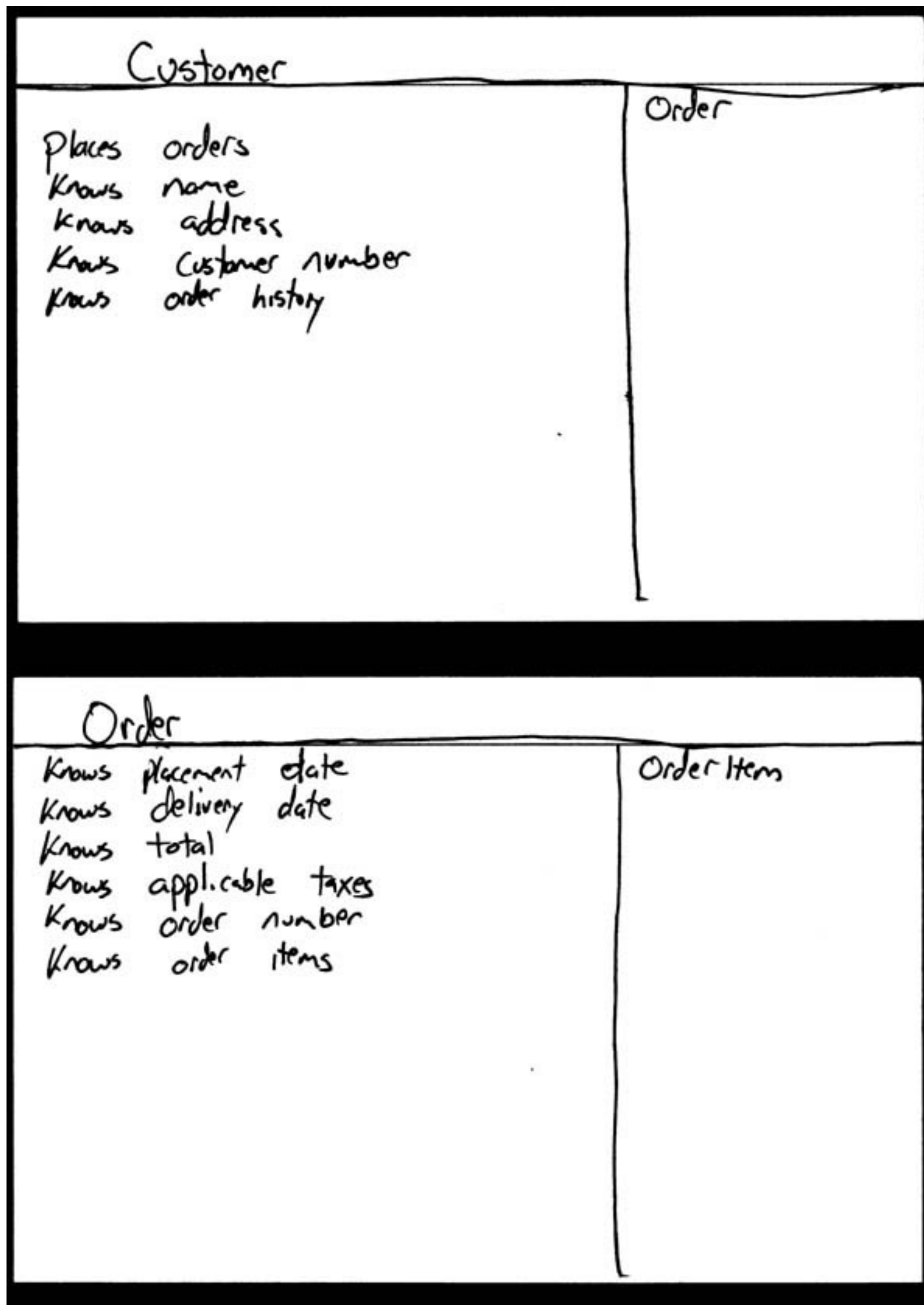
Schede CRC: uno strumento di design

- Una scheda CRC (Class Responsibility Collaboration) è uno strumento di design
- Basato sull'idea di *responsabilità*

Class	Collaborators
Responsibilities	

Professor	
teaches assigns tasks does exams grades	Assistant Technician Student

http://agilemodeling.com/artifacts/crcModel.htm



Enrollment	
Mark(s) received Average to date Final grade Student Seminar	Seminar

Transcript	
See the prototype Determine average mark	Student Seminar Professor Enrollment

Student Schedule	
See the prototype	Seminar Professor Student Enrollment Room

Room	
Building Room number Type (Lab, class, ...) Number of Seats Get building name Provide available time slots	Building

Professor	
Name Address Phone number Email address Salary Provide information Seminars instructing	Seminar

Seminar	
Name Seminar number Fees Waiting list Enrolled students Instructor Add student Drop student	Student Professor

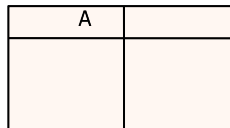
Student	
Name Address Phone number Email address Student number Average mark received Validate identifying info Provide list of seminars taken	Enrollment

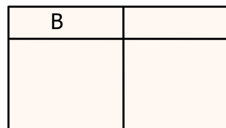
Building	
Building Name Rooms Provide name Provide list of available rooms for a given time period	Room

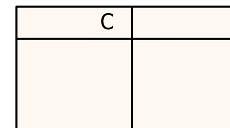
http://agilemodeling.com/artifacts/crcModel.htm

Problema: Design by responsibility

Vogliamo assegnare una responsabilità ad una classe







Quale classe sarà responsabile di una nuova funzionalità?

- Evitare dipendenze inutili
- Massimizzare coesione e minimizzare accoppiamento
- Aumentare la possibilità di riuso
- Minimizzare la necessità di manutenzione
- Aumentare la comprensibilità del progetto
- etc.

GRASP

- Nome scelto per sottolineare l’importanza di afferrare (**grasp**ing) i principi fondamentali della progettazione corretta orientata agli oggetti
- Acronimo per **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
- Descrivono i principi base di progettazione oo mediante le responsabilità
- Espressi come pattern
- Inventati da C. Larman

Craig Larman

GRASP
General Responsibility Assignment Software Patterns

- Expert
- Creator
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected variations (Law of Demeter)

Pattern: Information Expert

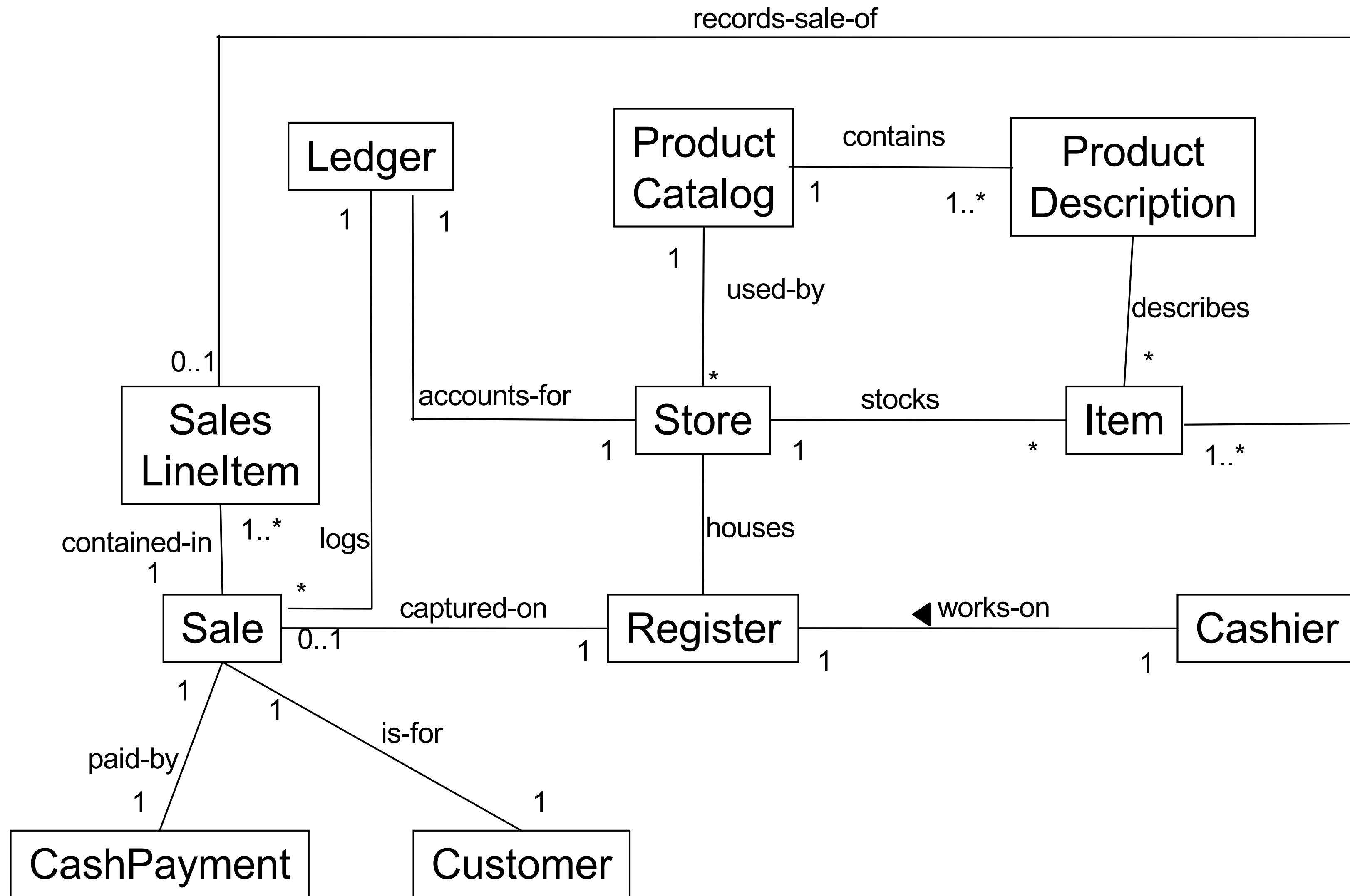
Problema:

Nella progettazione qual è il principio da seguire per assegnare una responsabilità ad una classe?

Soluzione:

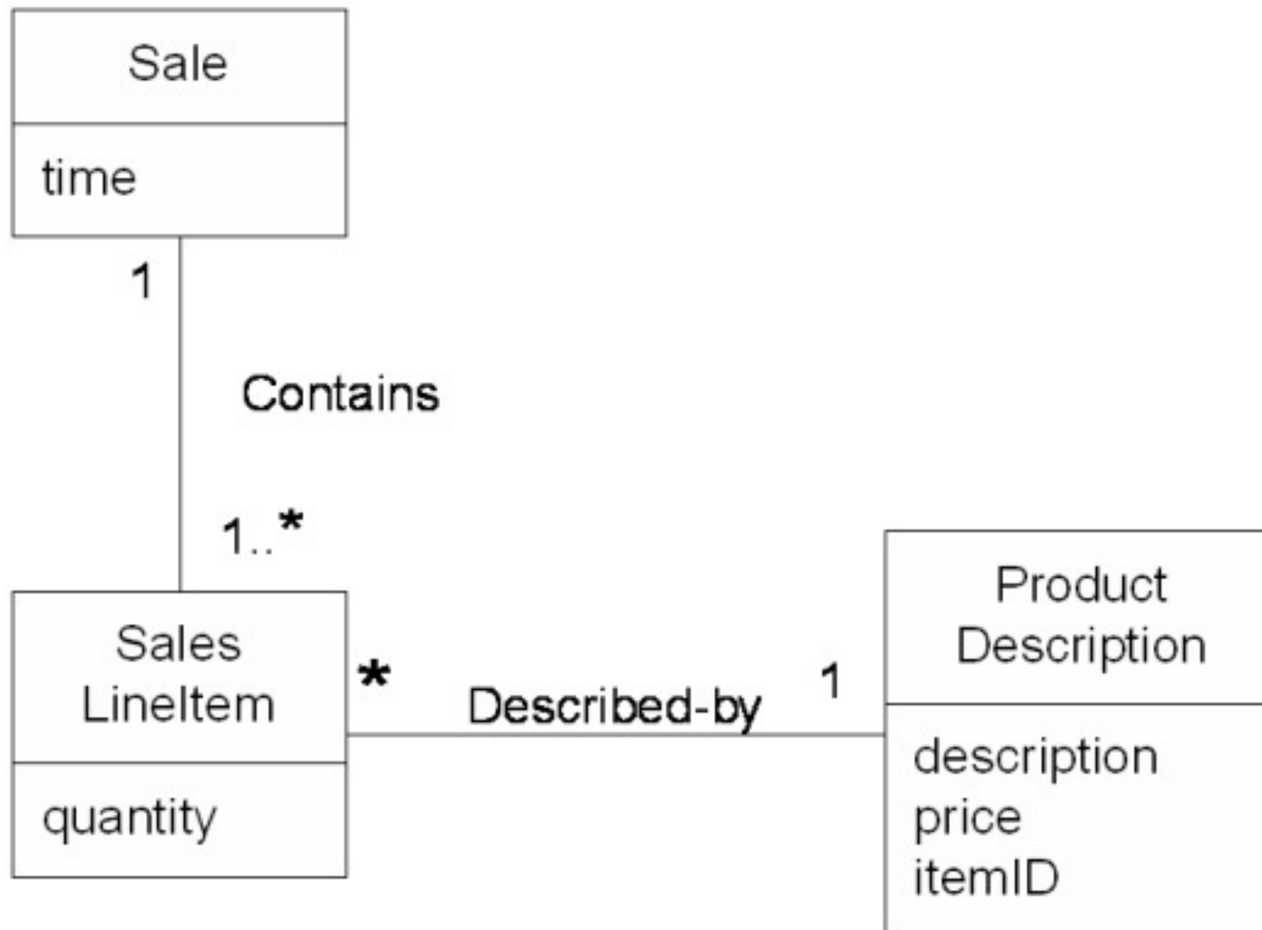
Si assegni la responsabilità alla classe che ha l'*informazione necessaria* per assumersi tale responsabilità

Un dominio: Nextgen POS (da Larman fig. 9.17)



Information Expert: esempio

Quale classe dovrebbe calcolare la somma totale di uno scontrino in un'applicazione POS?



Information Expert: esempio

Occorrono tutte le istanze *SalesLineItem* ed i sottototali

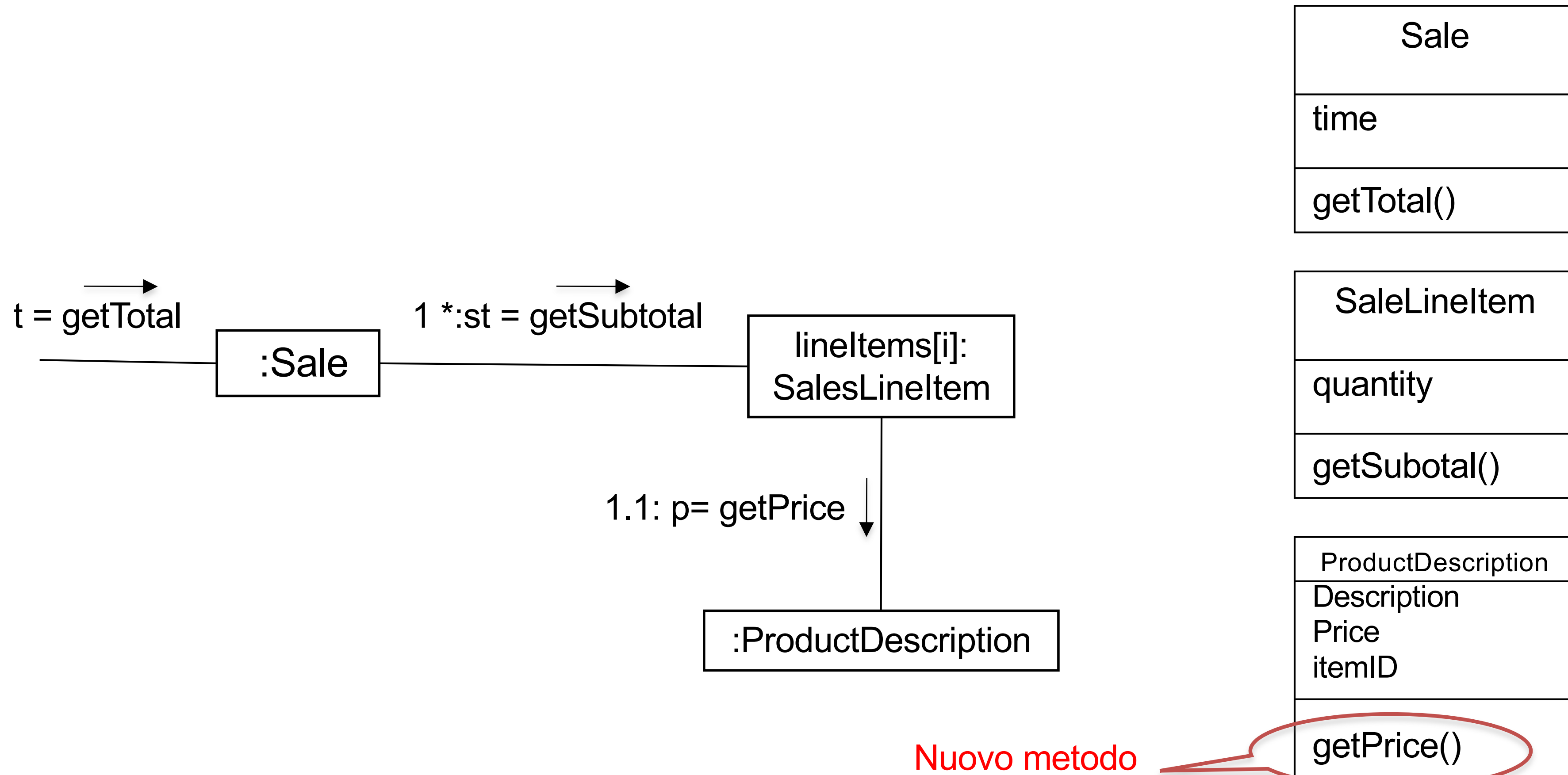
Sale li conosce, solo lei li conosce, dunque *Sale* è la classe “esperta”

Sale
date time
getTotal()

Information Expert: esempio

I sottototali sono necessari per ciascun elemento (quantità per prezzo)

Applicando di nuovo Information Expert, osserviamo che *SalesLineItem* conosce la quantità ed ha l'associazione con *ProductSpecification* utile per conoscere il prezzo



Expert : esempio

Dunque ecco le responsabilità assegnate alle tre classi

Classe	Responsabilità
Sale	conosce il totale della vendita
SalesLineItem	conosce il subtotale di linea
ProductSpecification	conosce il prezzo del prodotto

Pro e contro di Information Expert

- Conserva l'incaspulamento
- Promuove il disaccoppiamento
- Promuove la coesione
- Ma può complicare troppo una classe

Pattern: Creator

Problema: Chi dovrebbe creare un’istanza di A?

Soluzione: Si assegni alla classe B la responsabilità di creare un’istanza della classe A se è vera una o più tra queste condizioni:

- B contiene o aggrega oggetti A
- B registra oggetti A
- B usa oggetti A
- B ha i dati per inizializzare oggetti A

Chi crea le caselle (square)?

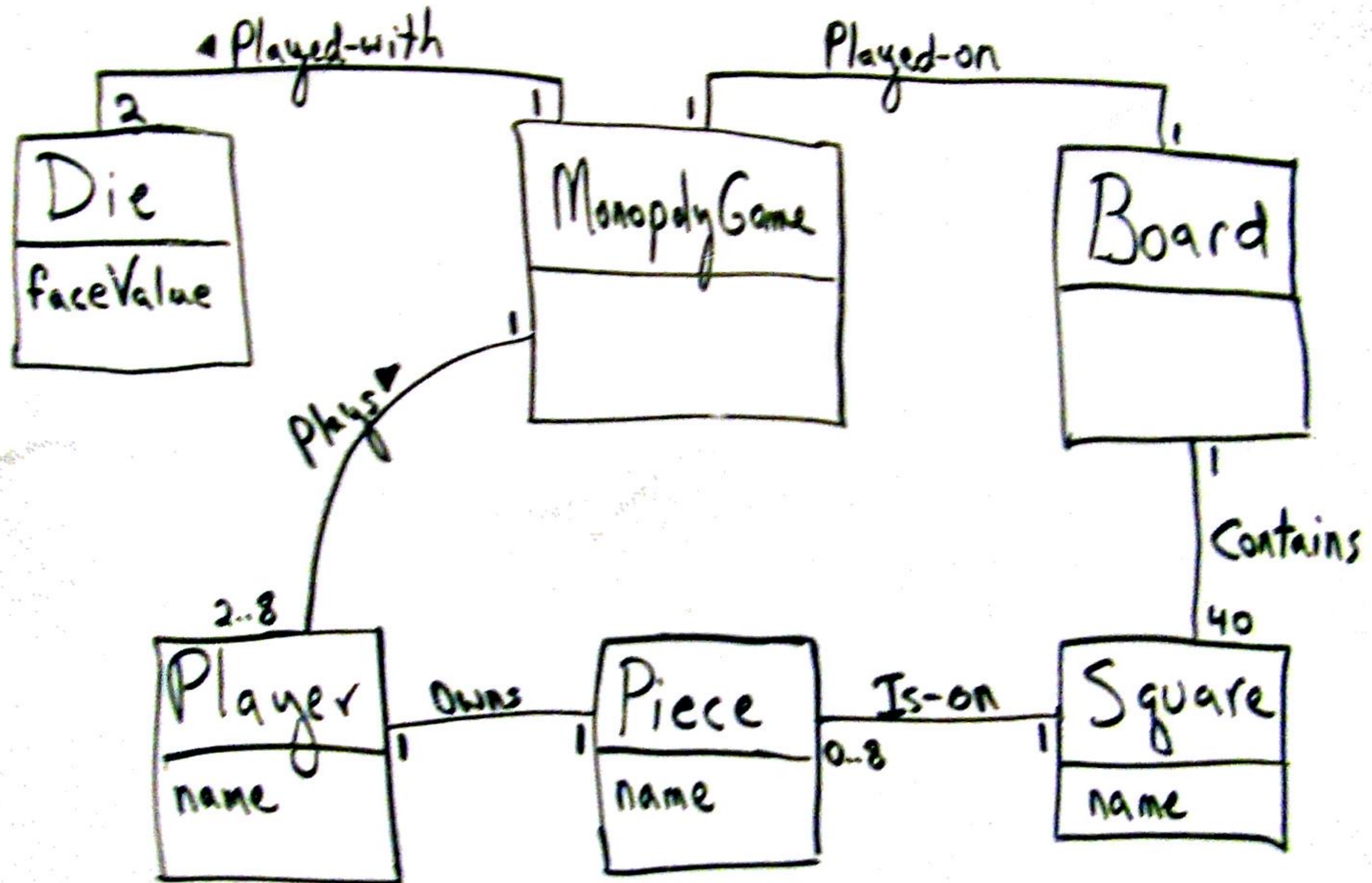


Figure 17.3, page 283

Il pattern Create permette di disegnare questa sequenza

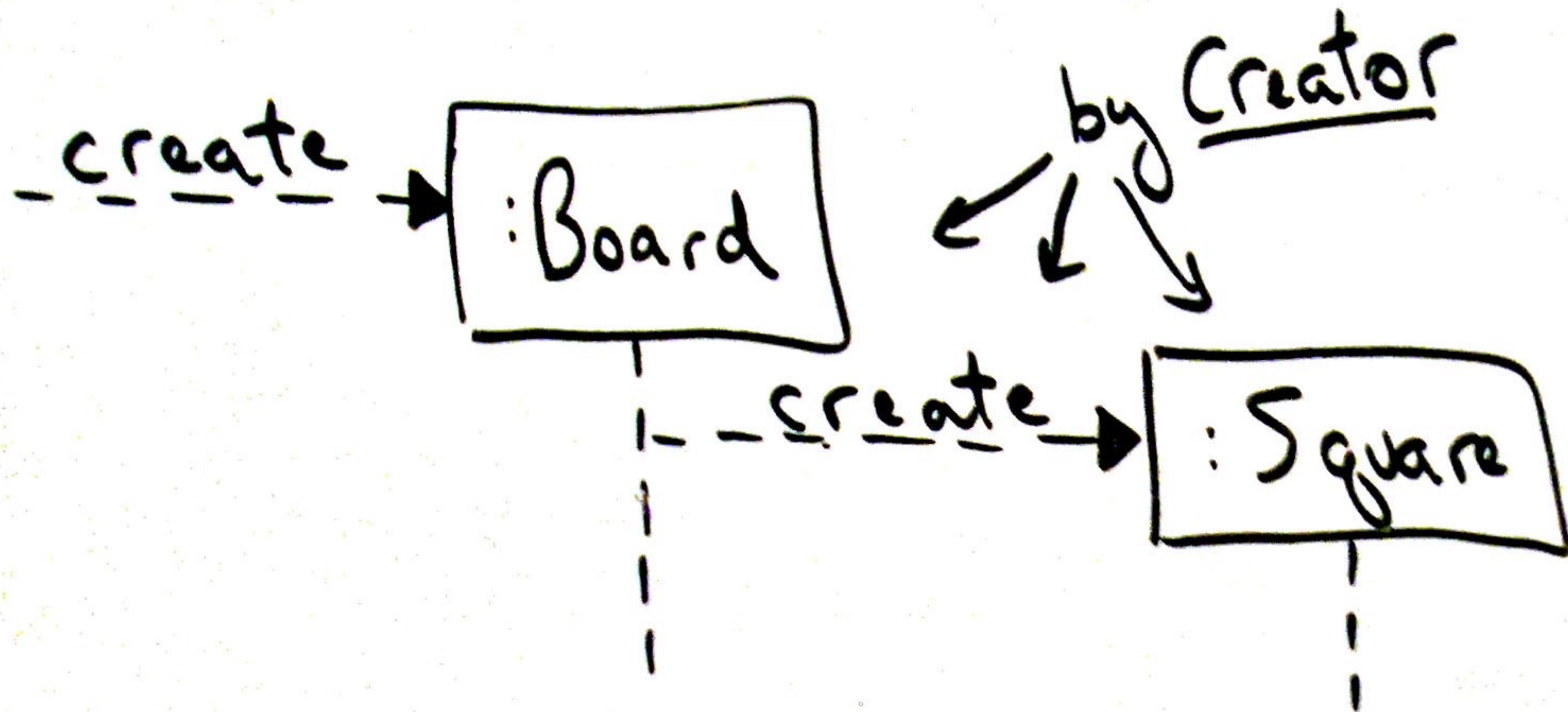


Figure 17.4, page 283

Il pattern Create permette di disegnare questo diagramma di classi

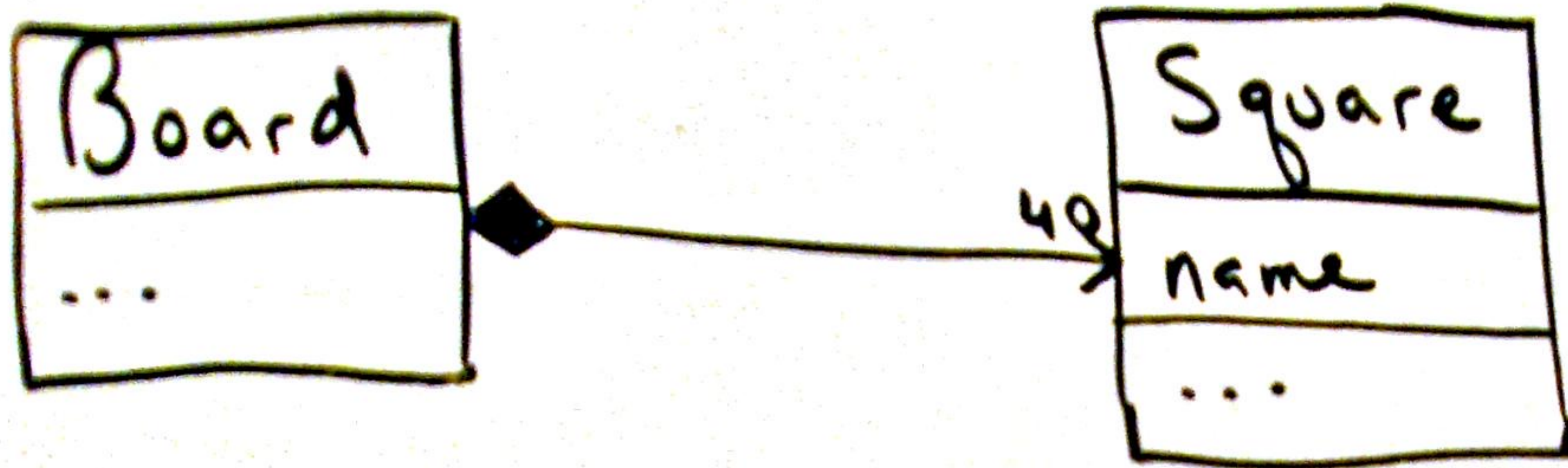


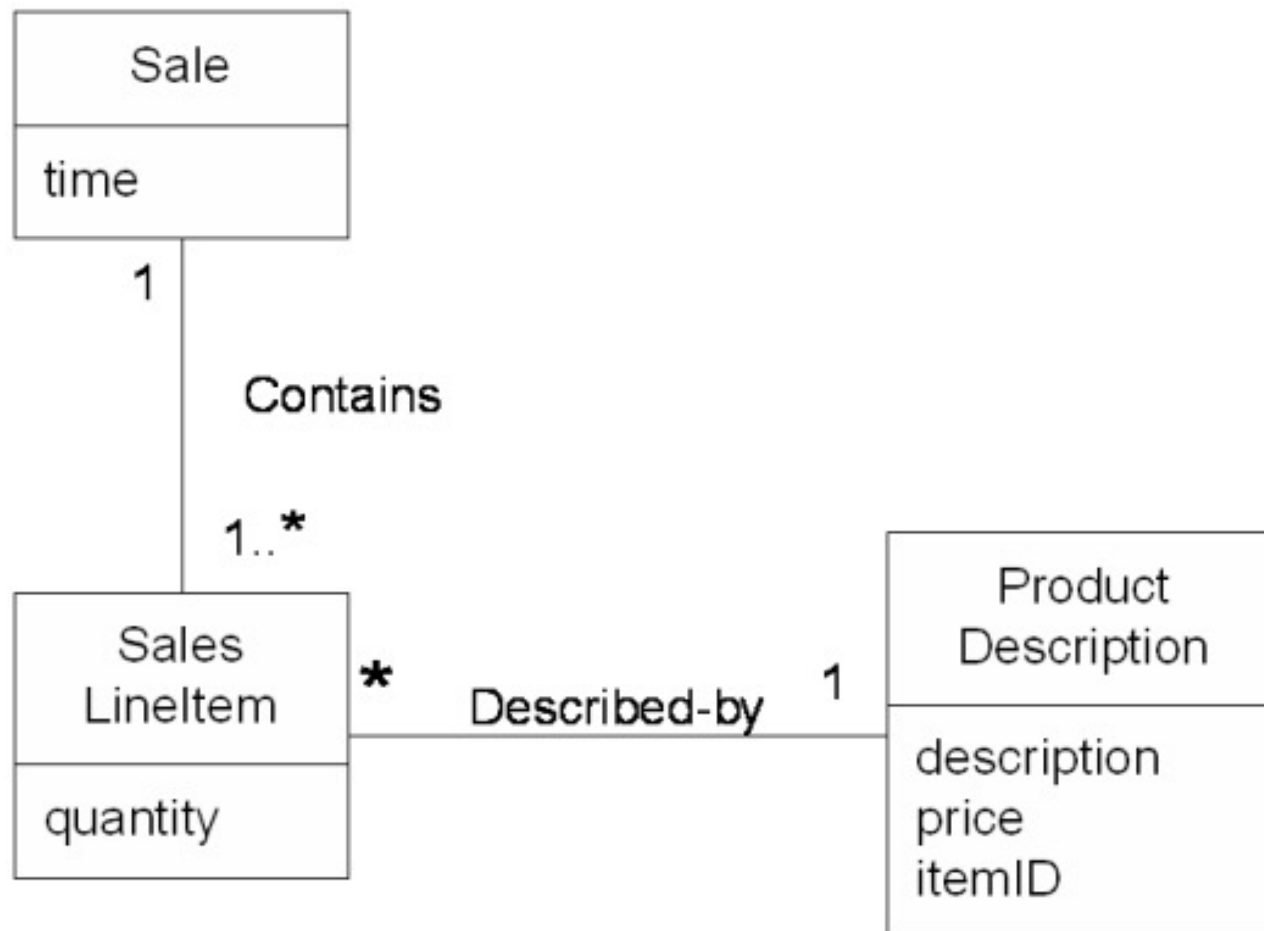
Figure 17.5 , page 283

Board è in relazione di composizione con *Square*
Cioè, Board contiene una collezione di 40 Squares

Creator : esempio

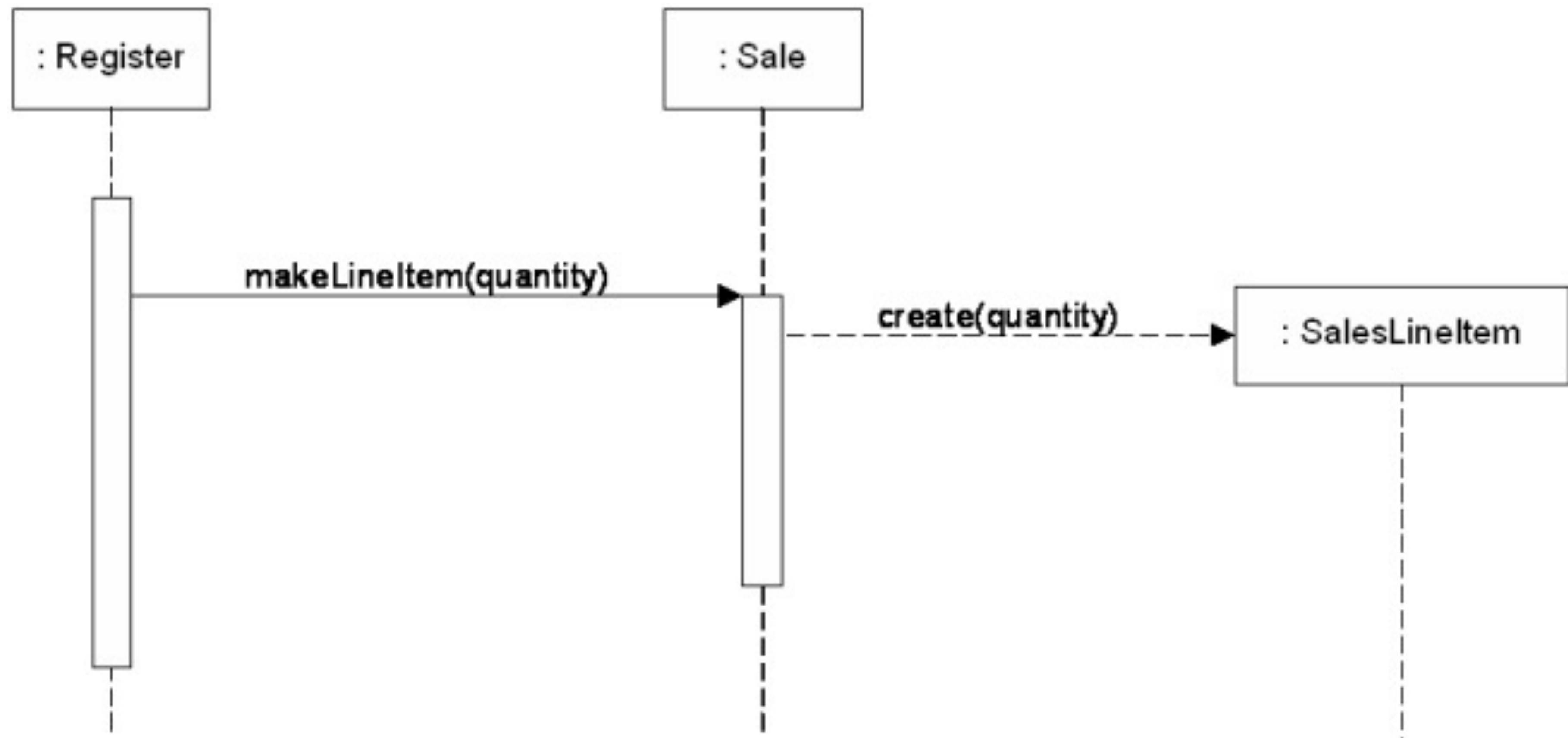
Chi è responsabile per creare oggetti *SalesLineItem*?

Cercate una classe che aggrega oggetti *SalesLineItem*



Creator: esempio

Creator pattern suggerisce come responsabile la classe “*Sale*”.
Un diagramma di sequenza che mostra la creazione:



Discussione di Creator

- Creare oggetti è una responsabilità molto comune
- Un oggetto diventa creatore quando una o più tra queste situazioni sono verificate:
  - È un Aggregatore che aggrega più Parti
  - È un Contenitore che contiene Contenuti
  - È un Recorder registra
  - È in grado di passare i dati di inizializzazione

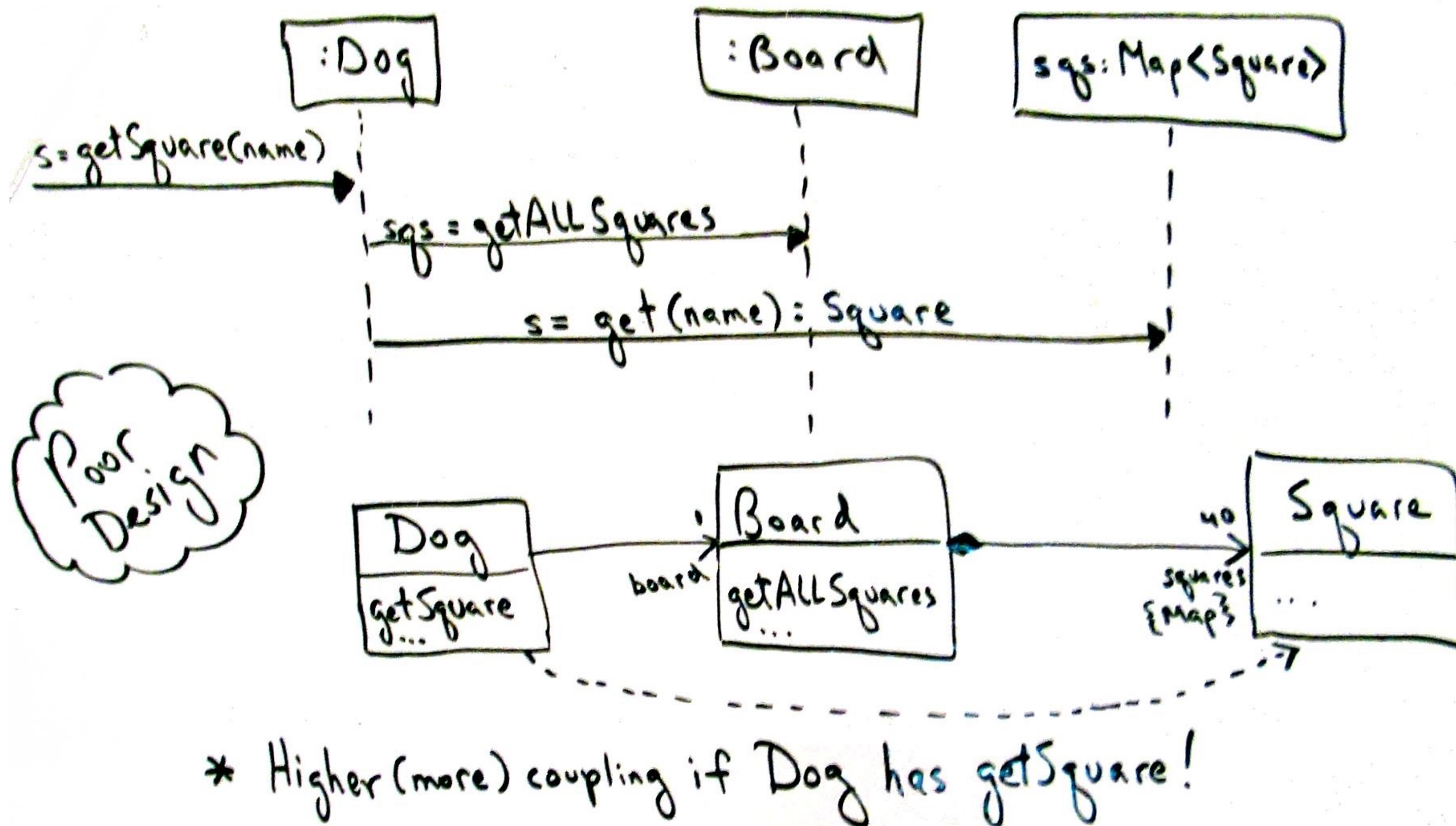
Controindicazioni

- Il pattern Creation può *complicare il progetto*:
  - Può peggiorare le prestazioni
  - È inadatto quando occorre creare condizionatamente istanze da una famiglia di classi simili
- In questi casi si possono usare altri pattern…
  - Più avanti parleremo di Factory e di altri …

Disaccoppiamento

- Nome: **Low Coupling** (disaccoppiamento)
- Problema: Come ridurre le conseguenze delle modifiche ed incoraggiare il riuso?
- Soluzione: Assegnare la responsabilità in modo che l’accoppiamento (dipendenze tra le classi) rimanga basso

Perché questo diagramma viola Low Coupling?



Perché è meglio lasciare la responsabilità getSquare in Board?

Benefici e controindicazioni

- Comprensibilità: le classi sono più semplici da capire da sole
- Manutenibilità: le classi non sono influenzate da modifiche ad altri componenti
- Riusabilità: più facile usare le classi in sistemi diversi

Ma:

- Meglio lasciare accoppiate classi stabili
  (in librerie o se classi molto usate e ben testate)

Pattern: Controller

- Nome: **Controller**
- Problema: a chi si assegna la responsabilità di gestire gli eventi esterni che entrano in una interfaccia utente?
- Soluzione: la responsabilità di ricevere o gestire dall'esterno un evento che attiva un sistema si può assegnare:
  - Mediante un unico rappresentante del sistema (*façade* pattern, non ancora visto)
  - Mediante un *controllore di sessione:* per ogni caso d'uso il pattern definisce il primo oggetto che riceve e coordina un'operazione di sistema

Controller : esempio

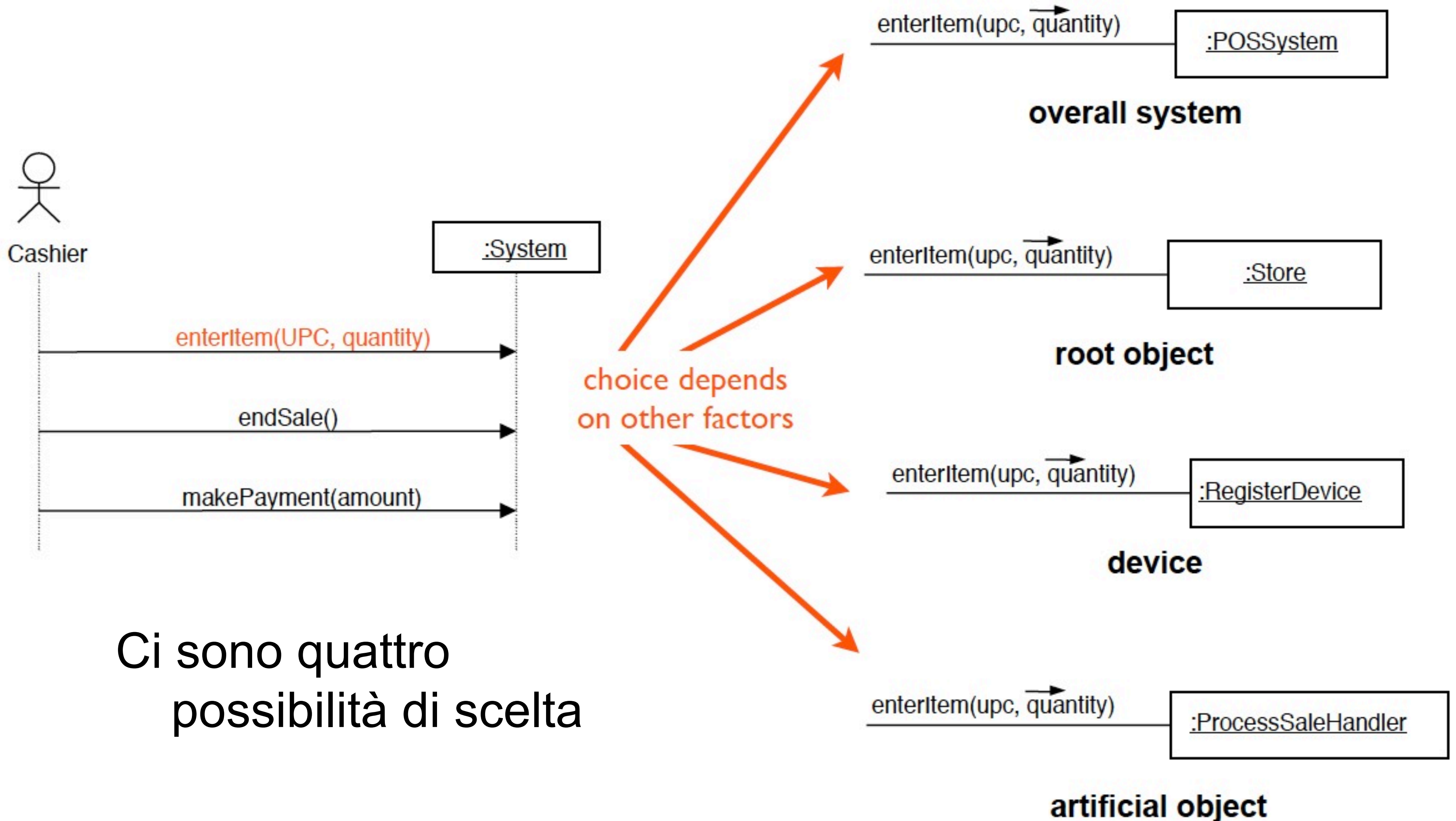
In uno scenario di caso d’uso (nell’esempio del videostore) chiamato “Buy Items” supponiamo che ci sono i seguenti eventi da gestire:

- `enterItem()`
- `endSale()`
- `makePayment()`

Chi ha la responsabilità di *enterItem()*?

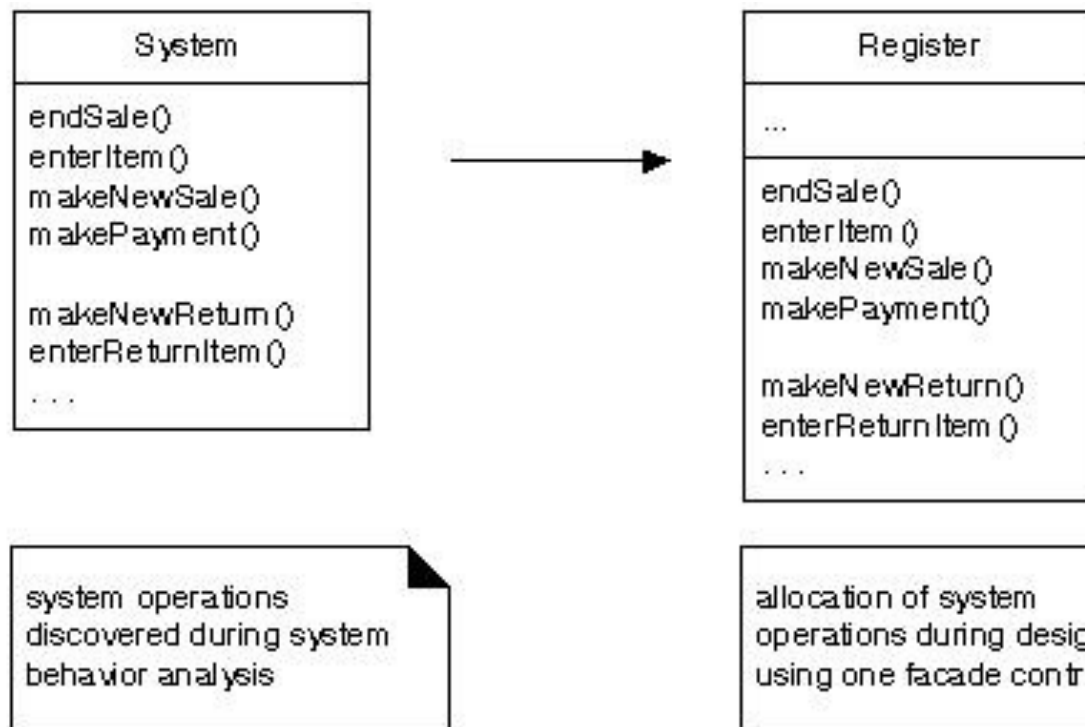
Cioè, chi gestisce l’input da utente?

Controller : esempio

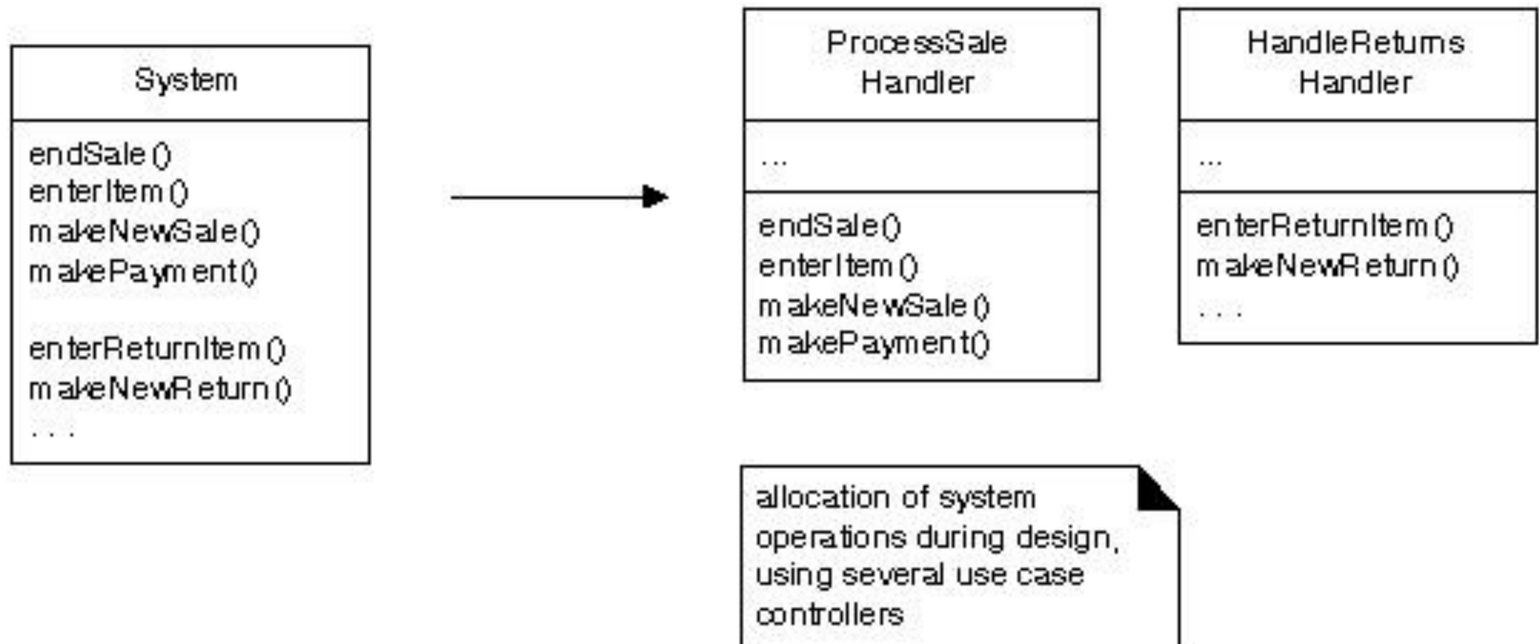


La scelta sarà influenzata da fattori quali coesione e accoppiamento

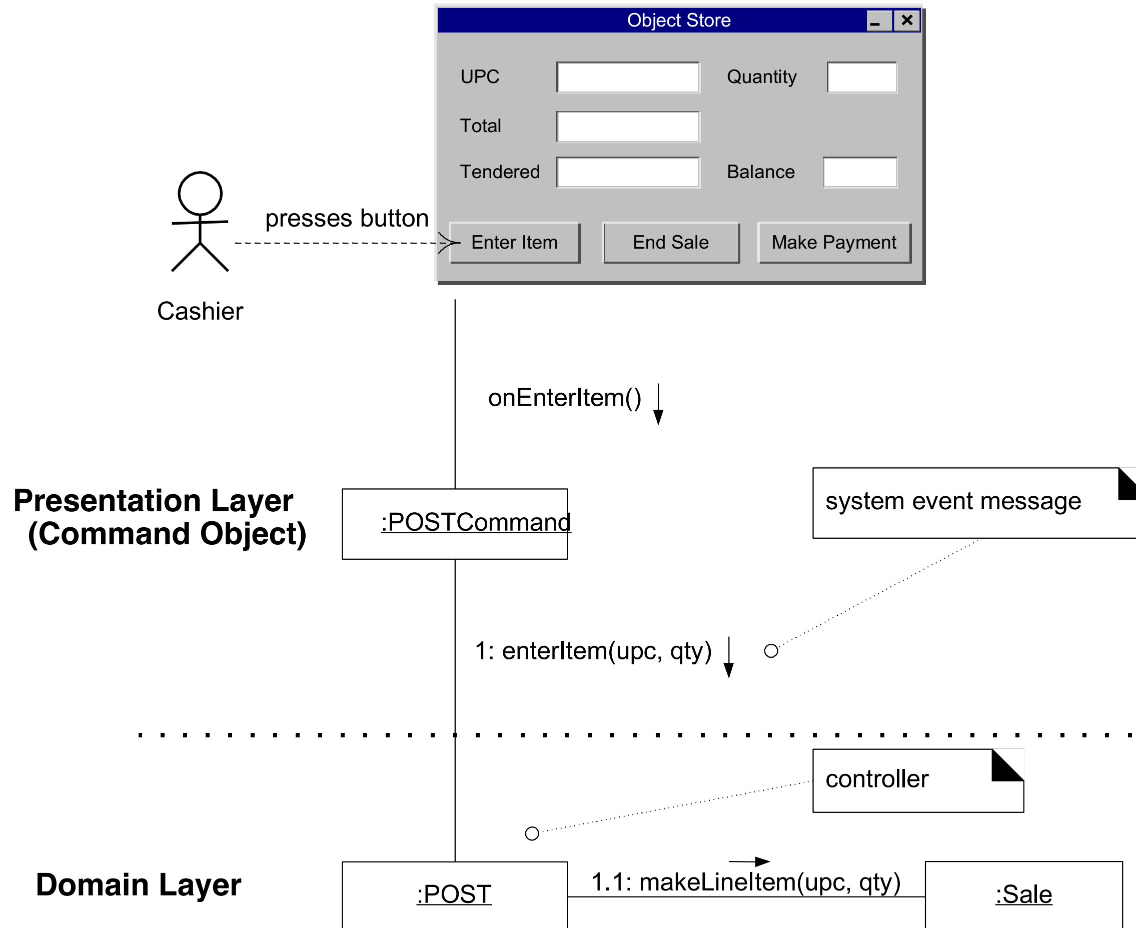
Façade



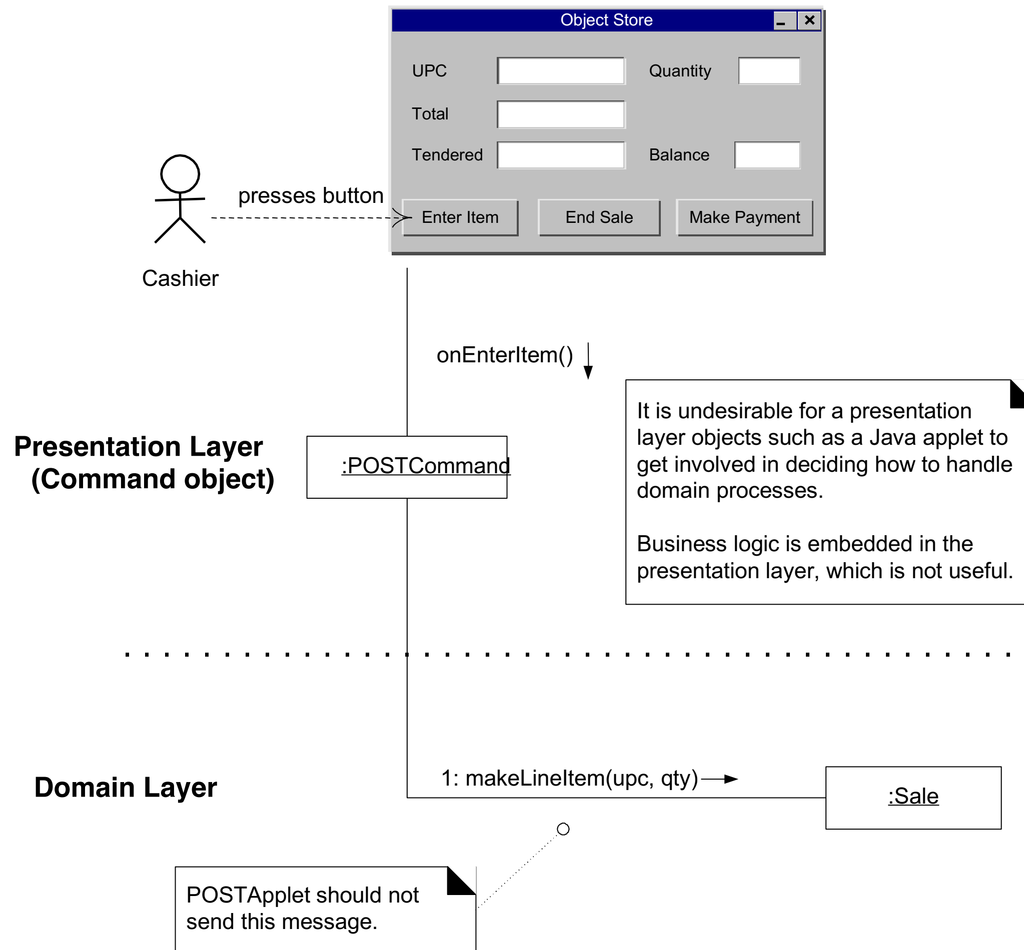
Controller



Buona soluzione: presentazione disaccoppiata dal dominio



Cattiva soluzione: *presentation layer* accoppiato al *domain layer*



Controller: pro e contro

- L'uso di un oggetto *controller* separa la gestione di eventi esterni da quella degli eventi interni ad un sistema
- **Esempio**: usare un controller per gestire gli eventi da mouse separatamente dagli eventi sulle entità puntate
- **Attenzione**: se si assegnano troppe responsabilità gli oggetti controller diverranno fortemente accoppiati e poco coesi

Chi è il controller di playGame?

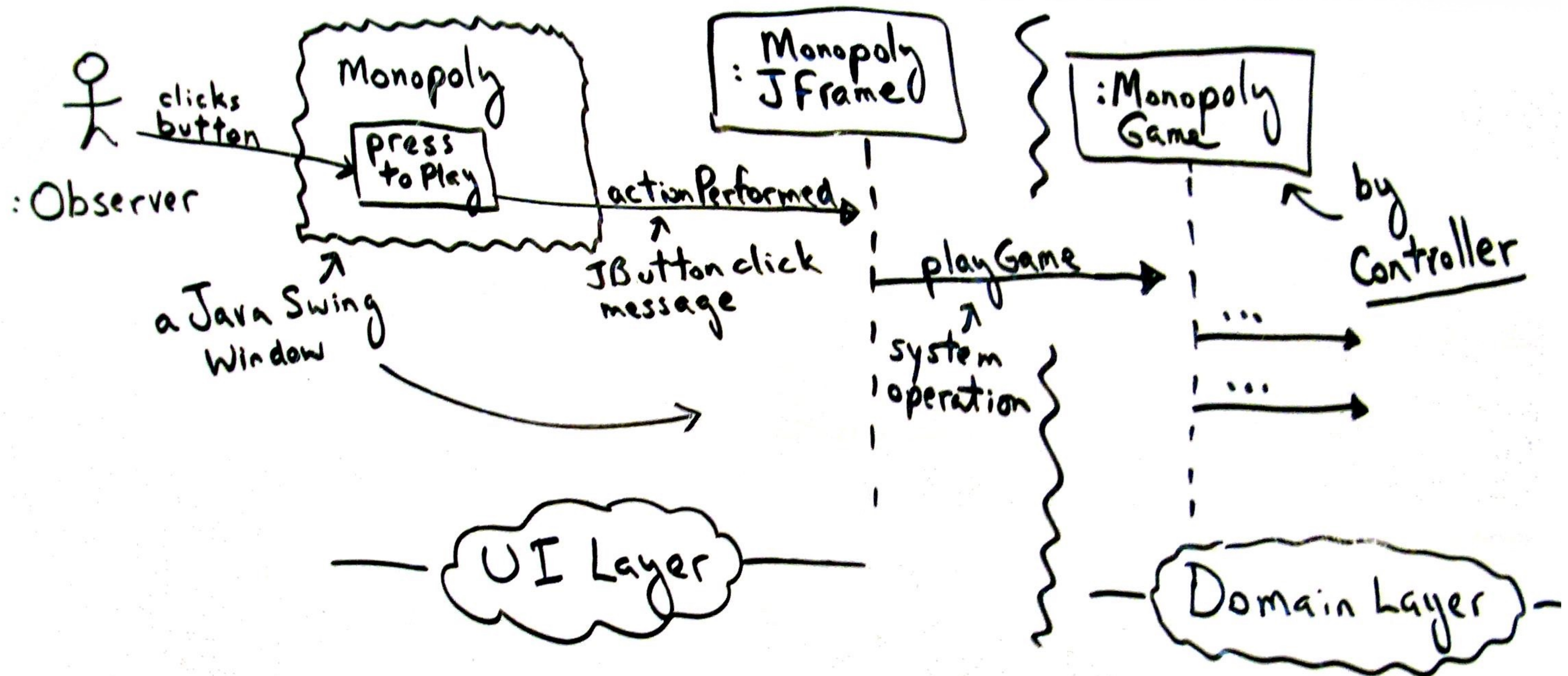


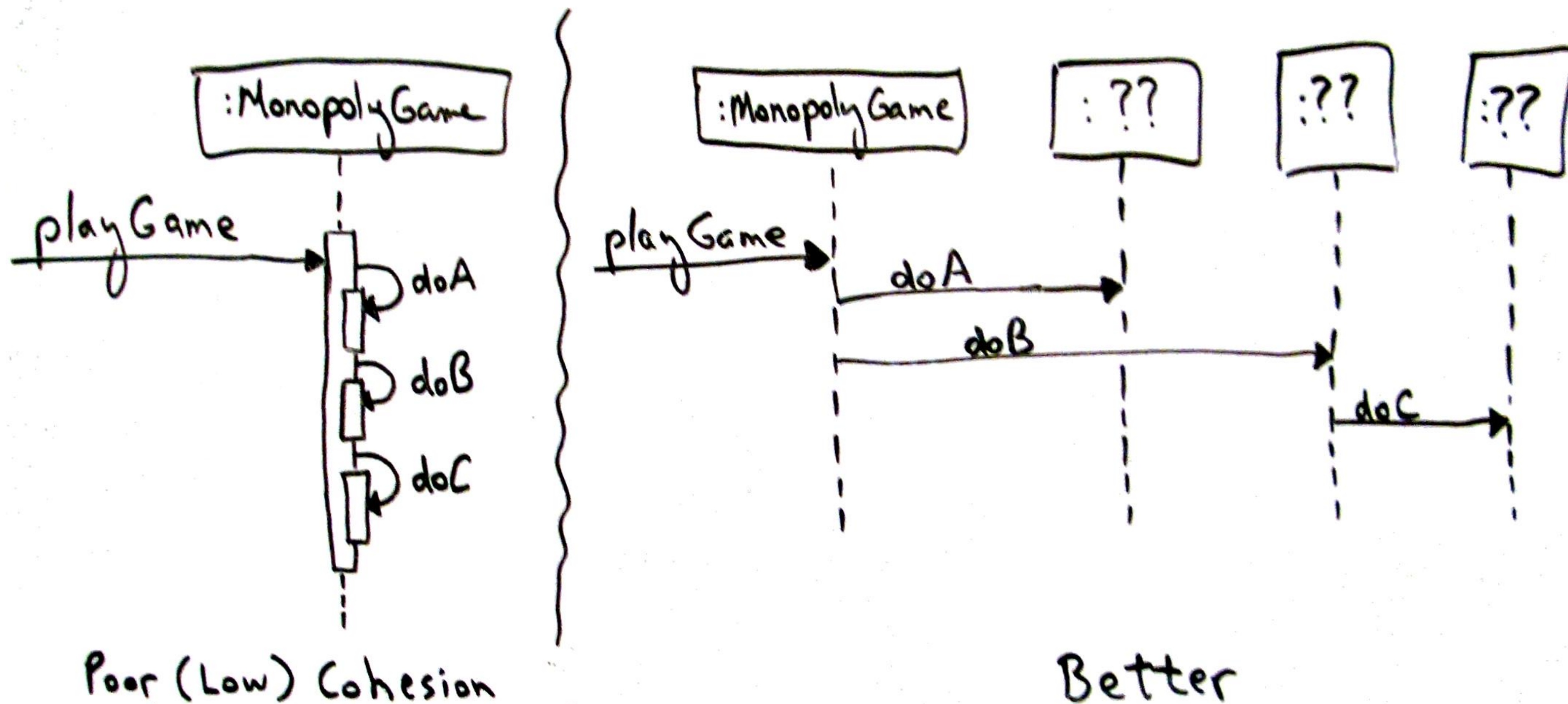
Figure 17.9, p. 288

Quale classe rappresenta il sistema
o un caso d'uso rilevante?

Pattern: Coesione

- La coesione misura la correlazione delle responsabilità di una classe
- Nome: **High Cohesion**
- Problema: come si disegnano classi coese e maneggevoli?
- Soluzione: assegnare le responsabilità in modo da favorire la coesione

Perchè il disegno a destra è più coeso?



Delegazione di responsabilità e coordinamento

Benefici e controindicazioni

- Comprensibilità, riusabilità
- Complementa il disaccoppiamento perché suggerisce di non aumentare le responsabilità (o il codice) in una sola classe il che semplifica la manutenzione

Ma:

- Talvolta è desiderabile un oggetto server meno coeso, se deve offrire interfaccia per molte operazioni, perché oggetti remoti e relative comunicazioni lo renderebbero inefficiente

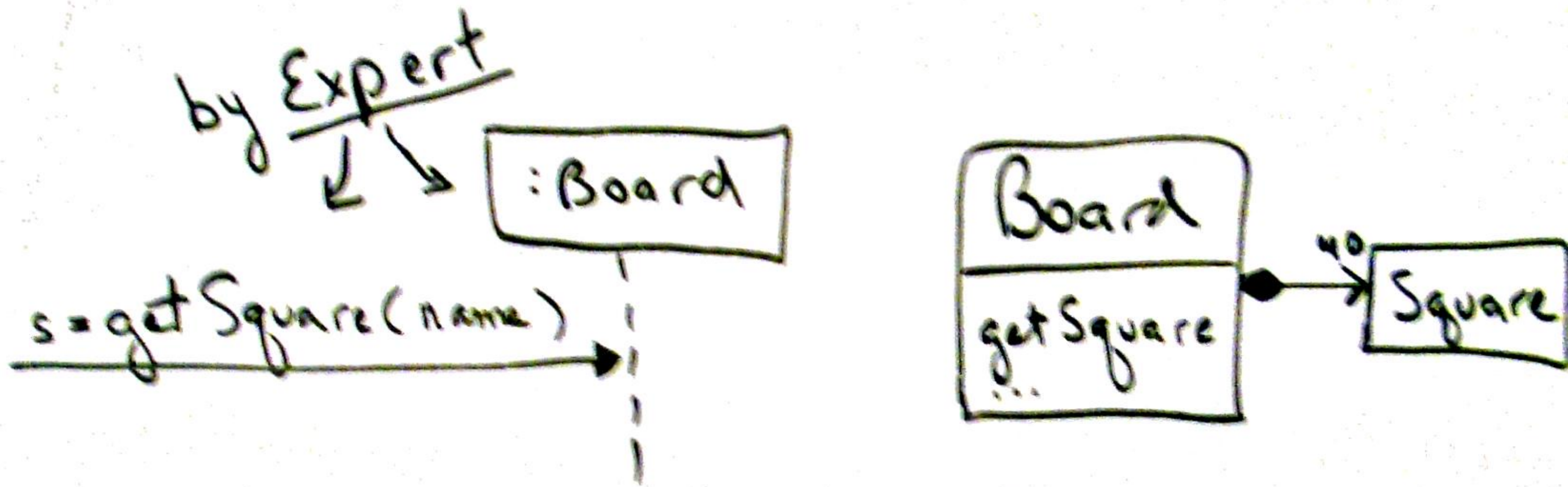
Ancora su Expert

Nome completo: Information Expert

Problema: Come si assegnano le responsabilità?

Soluzione: darle alla classe che ha l'informazione necessaria per gestire la responsabilità

- Es.: Board sa come arrivare a Square



Benefici e controindicazioni

- Promuove l'incaspulamento
  - Le classi usano proprie informazioni per effettuare il proprio compito
- Promuove classi coese e leggere

Ma:

- Information Expert può contraddire i pattern Low Coupling e/o High Cohesion
- I sottosistemi grandi si progettano meglio decomponendoli, ovvero la logica di business va in un componente, la UI in un altro, il DB in un altro, ecc.

Pattern: Polimorfismo

- Problema: come gestire responsabilità alternative basate sul tipo (classe)
- Soluzione: quando le alternative (o i comportamenti) variano col tipo (classe) assegna la responsabilità ai tipi per il quale il comportamento varia, usando operazioni polimorfe

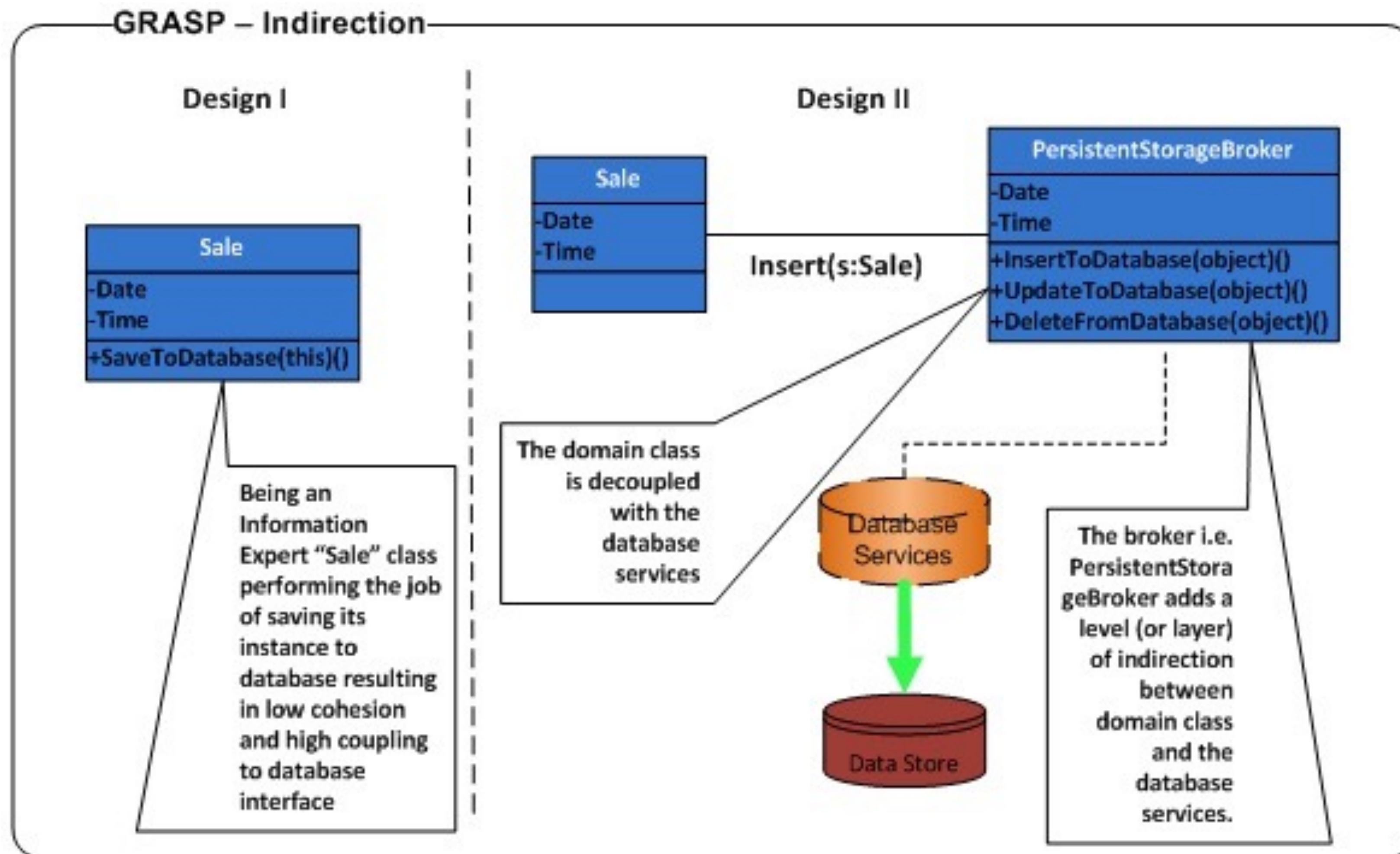
Pattern: Pure Fabrication

- Problema: quando usando Information Expert si violano Cohesion e/o Coupling, come assegnare le responsabilità?
- Soluzione: introdurre una classe artificiosa (di convenienza) altamente coesa e poco accoppiata

Pattern: Indirection

- Problema: dove assegnare una responsabilità, evitando l'accoppiamento diretto? Come disaccoppiare?
- Soluzione: Assegna la responsabilità ad un oggetto intermediario, che dunque crea una indirezione tra gli altri componenti

Indirection: esempio



http://www.dotnetfunda.com/articles/show/1578/software-patterns-grasp-part-ix-indirection

Pattern: Protected Variations

- Problema: come progettare un oggetto le cui responsabilità non sono ancora fissate, senza che ci sia un impatto indesiderato su altri oggetti?
- Soluzione: identifica i punti in cui sono prevedibili variazioni, e crea attorno ad essi un’interfaccia stabile

Nota: è uno dei pattern più importanti nella progettazione software, generalizza la “Legge di Demetra”

Principio: Legge di Demetra

La "*Legge di Demetra*" (o "Non parlare agli sconosciuti") è un principio di progettazione che suggerisce di disaccoppiare le classi:

-Ciascuna classe deve avere la minima informazione necessaria sulle altre classi (incluse le sue classi componenti)

-Ogni classe parla solo con le classi amiche (non parla con le classi "sconosciute")

www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/navigation-control/apply-LOD.html

Caratteristiche di un buon pattern

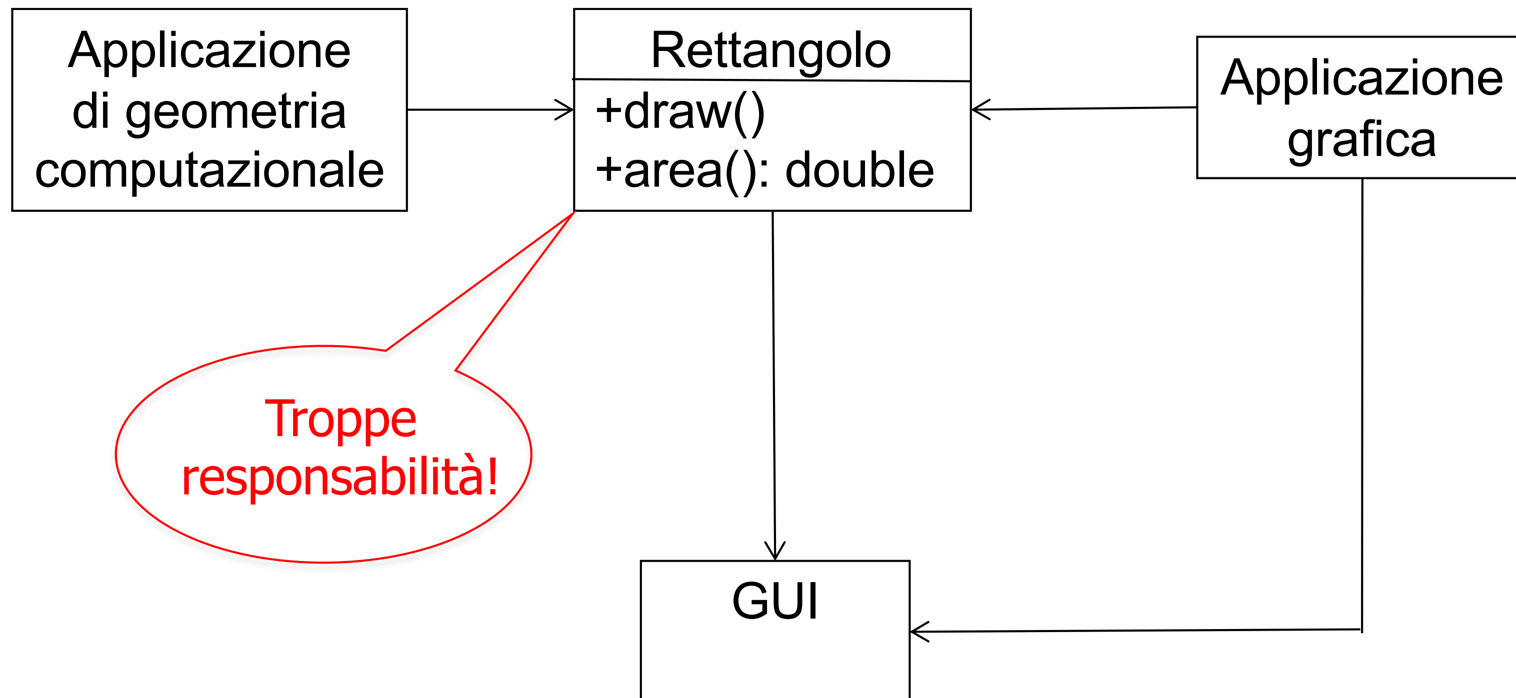
- Risolve un problema
- Descrive di solito una relazione problematica e sempre una relazione risolutiva
- La soluzione non è ovvia
- Il concetto è comprovato (cioè esiste consenso sul problema e sulla soluzione)
- È utile e possibilmente elegante
- Segue principi generali di progettazione

I principi SOLID

- **Single responsibility**: ogni classe dovrebbe avere una sola responsabilità (alta coesione)
- **Open-closed**: una classe dovrebbe essere aperta all'estensione, ma chiusa alle modifiche
- **Liskov substitution**: ogni superclasse dovrebbe essere rimpiazzabile da una sua sottoclasse, senza modificare il significato del programma
- **Interface segregation**: meglio tante interfacce, una per ciascun cliente, che un'unica interfaccia buona per tutti
- **Dependency inversion**: meglio dipendere dalle astrazioni che dalle concretizzazioni

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

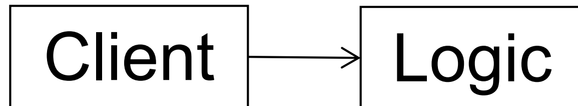
Single responsibility



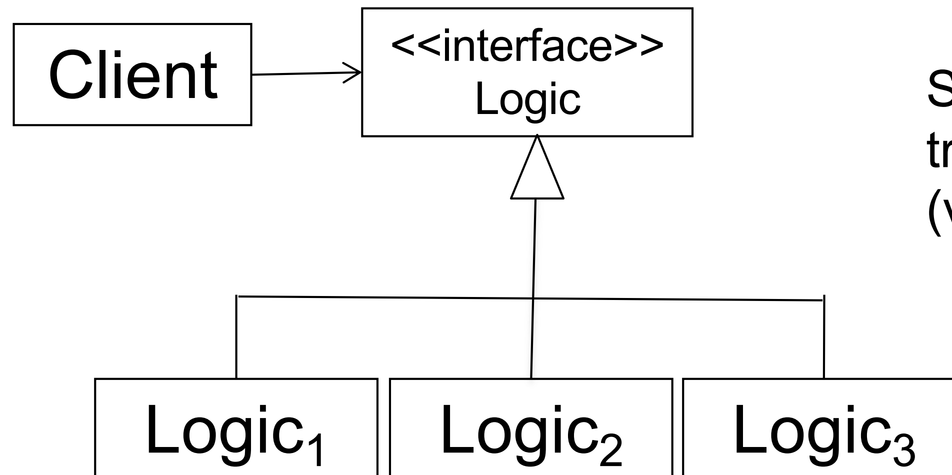
SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

Il principio Open-Closed



Questa relazione viola il principio Open-Closed perché rende Client direttamente dipendente da Logic



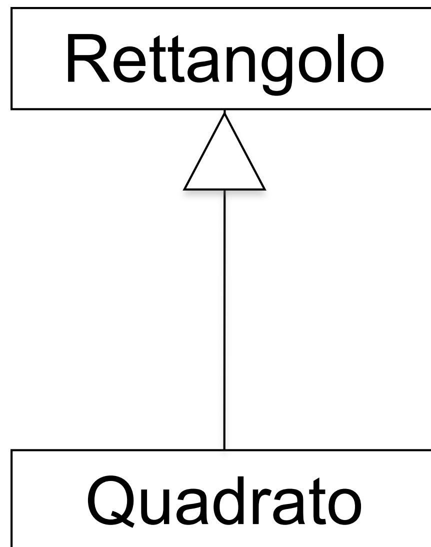
Si può evitare il problema trasformando Logic in un'interfaccia (vedi il pattern GoF di nome Strategy)

Il principio di sostituzione di Liskov-1

```
void DrawShape(const Shape& s)
{
    if (typeid(s) == typeid(Square))
    DrawSquare(static_cast<Square&>(s));
    else if (typeid(s) == typeid(Circle))
    DrawCircle(static_cast<Circle&>(s));
}
```

Questa funzione va cambiata ogni volta che si aggiunge una sottoclasse di Shape (violando il principio Open-Closed)

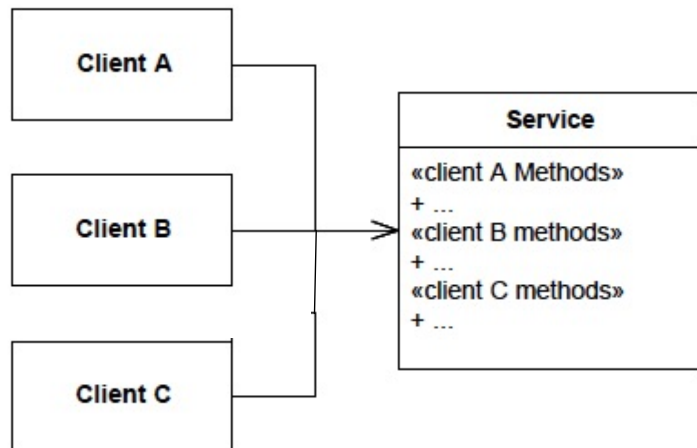
Il principio di sostituzione di Liskov - 2

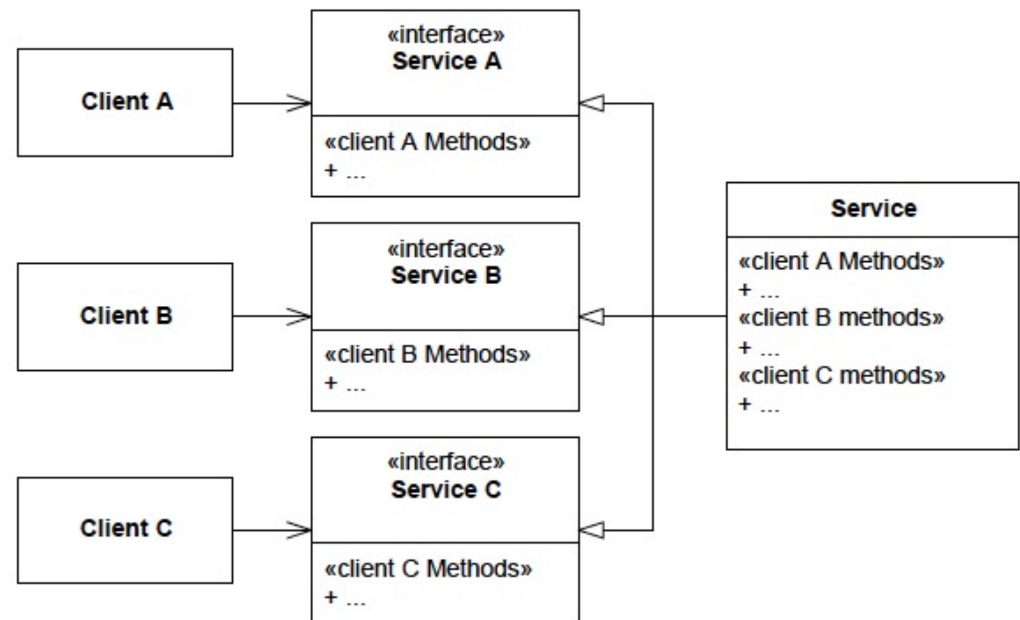



```
class Rettangolo {
 public:
   void SetBase(double w) {base=w;}
   void SetAltezza(double h) {altezza=w;}
   double GetAltezza() const {return altezza;}
   double GetBase() const {return base;}
 private:
   double base;
   double altezza;
};
```

Quadrato eredita i due lati (spreco di memoria) e le due funzioni separate che lavorano con tali lati (pericolo di inconsistenze)

Segregazione delle interfacce

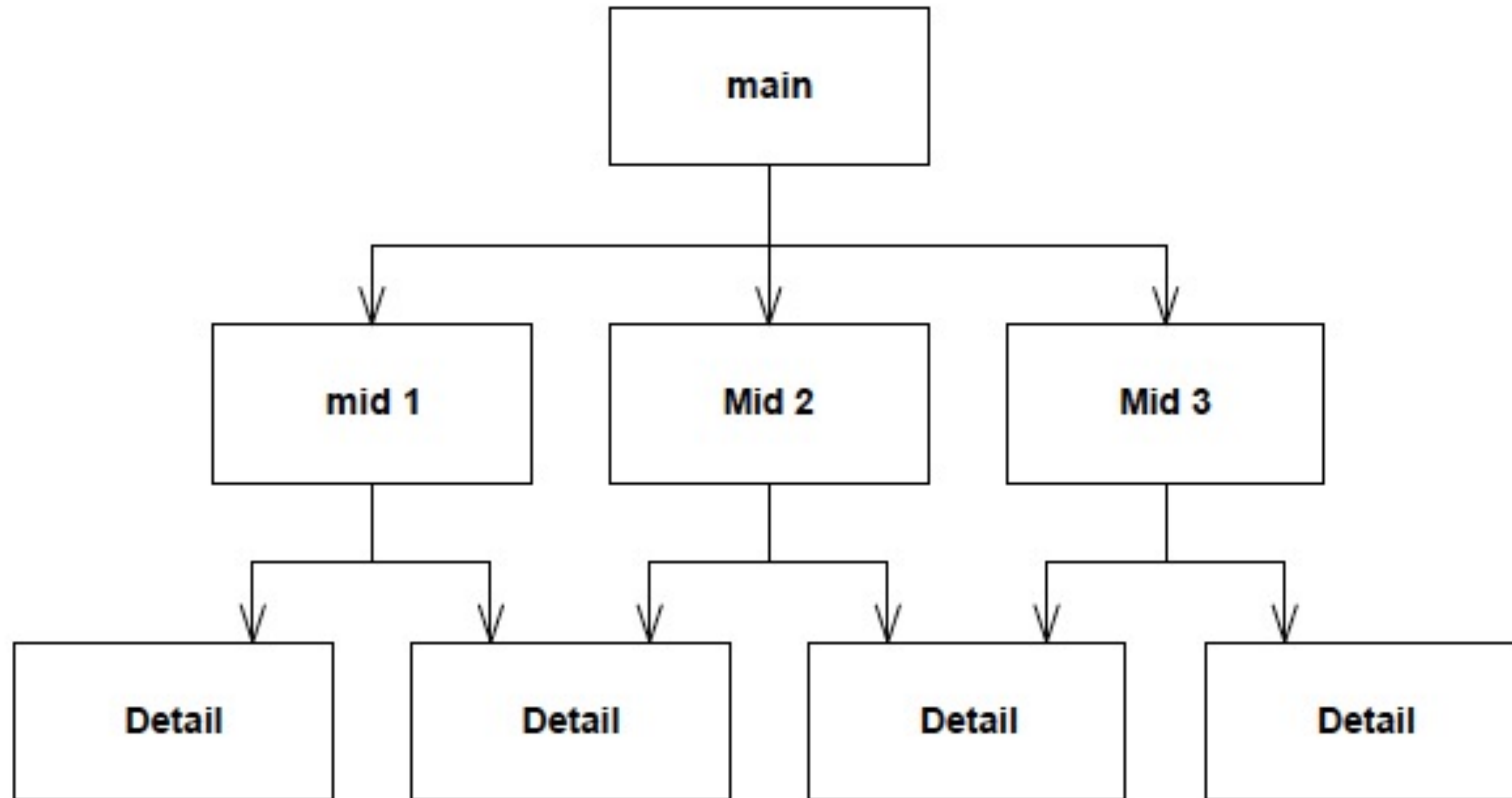




Server senza interfacce segregate: la modifica di una può impattare le altre

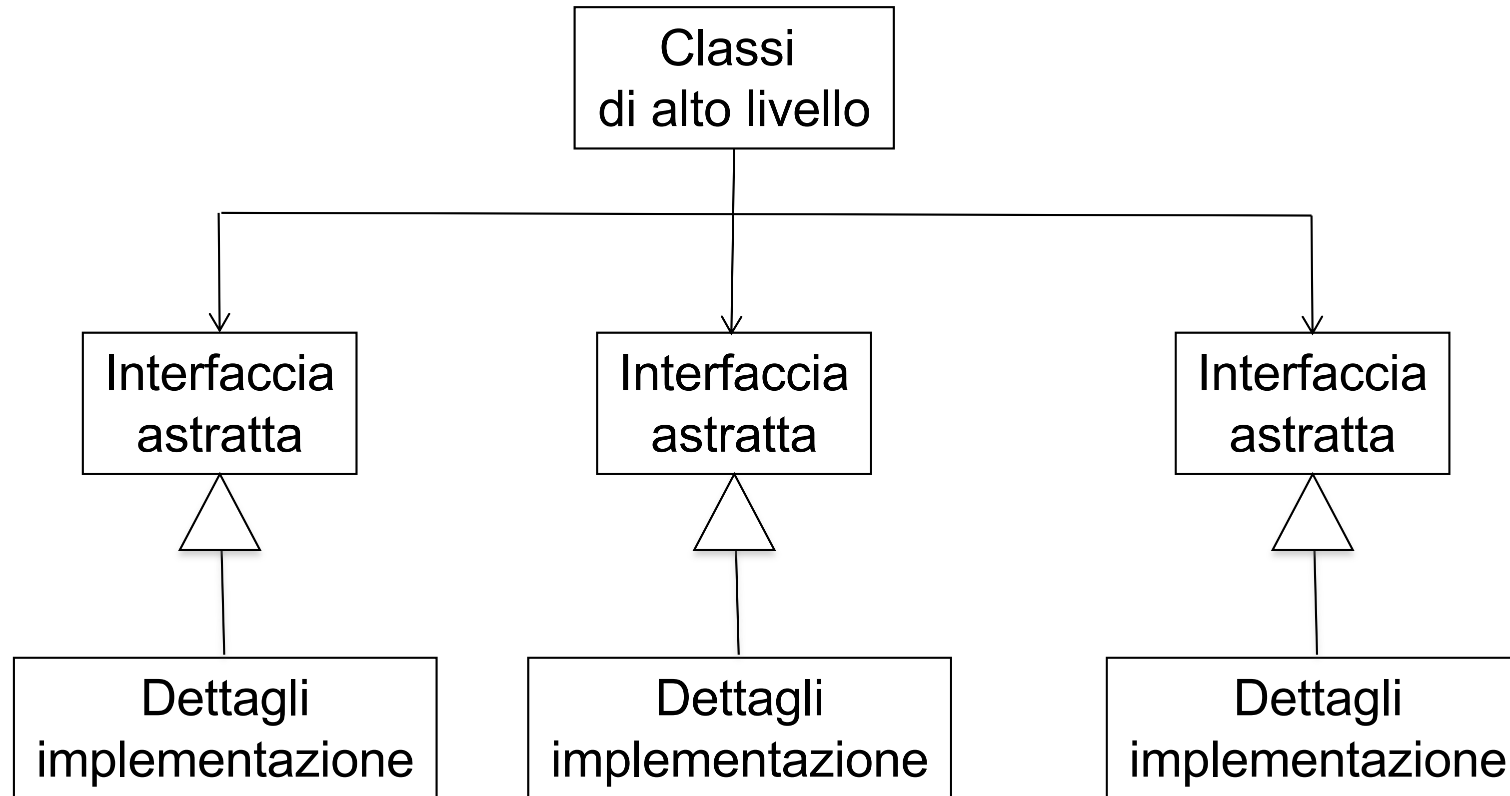
Server con interfacce segregate

Inversione delle dipendenze-1



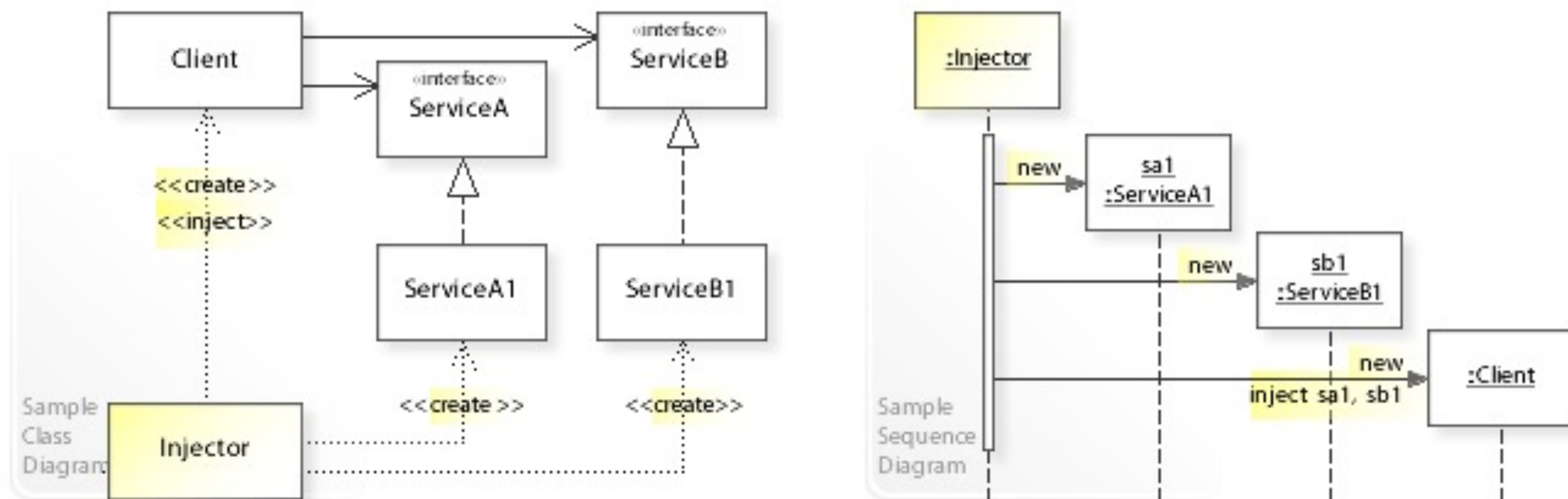
Architettura procedurale

Inversione delle dipendenze-2



Architettura orientata agli oggetti

Dependency injection
(pattern basato su dependency inversion)



Separa la creazione delle dipendenze di un cliente dal suo comportamento

Sommario

- L'assegnazione di responsabilità è importante nella fase iniziale del design
- I pattern GRASP sono coppie **problema**/**soluzione** utili per assegnare le responsabilità all' inizio della progettazione
- Abbiamo visto cinque pattern GRASP:
  - Creator, Information Expert, Controller, Low Coupling e High Cohesion
  - Esistono altri quattro pattern GRASP, cercateli da soli
- L'assegnazione di responsabilità dovrebbe seguire i principi SOLID e la legge di Demetra
- I principi SOLID servono a diminuire le dipendenze usando i meccanismi OO, per es. l'ereditarietà e le interface astratte

Domande di autotest

- Cos’è un design pattern?
- Cos’è la progettazione guidata dalle responsabilità?
- Quali sono i pattern GRASP?
- Cosa suggerisce il pattern Expert?
- Cosa suggerisce il pattern Controller?
- Cosa dicono i principi SOLID?

Riferimenti

Larman, *Applying UML and patterns* (cap 17), Pearson 2005

Martin, *Agile software development: principles,* ***patterns****, and practices*, Pearson, 2002

`www.agilemodeling.com/artifacts/crcModel.htm`

`www.ccs.neu.edu/research/demeter`

`en.wikipedia.org/wiki/Anti-pattern`

Siti utili

- www.oodesign.com
- www.objectmentor.com
- butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- tomdalling.com/blog/software-design
- sourcemaking.com

Pubblicazioni di ricerca

- PLOP: Conference on Pattern Languages of Programs
- ECOOP: European Conference on Object Oriented Programming
- SPLASH: Systems, Programming, Languages, and Applications

Domande?