

La progettazione orientata agli oggetti

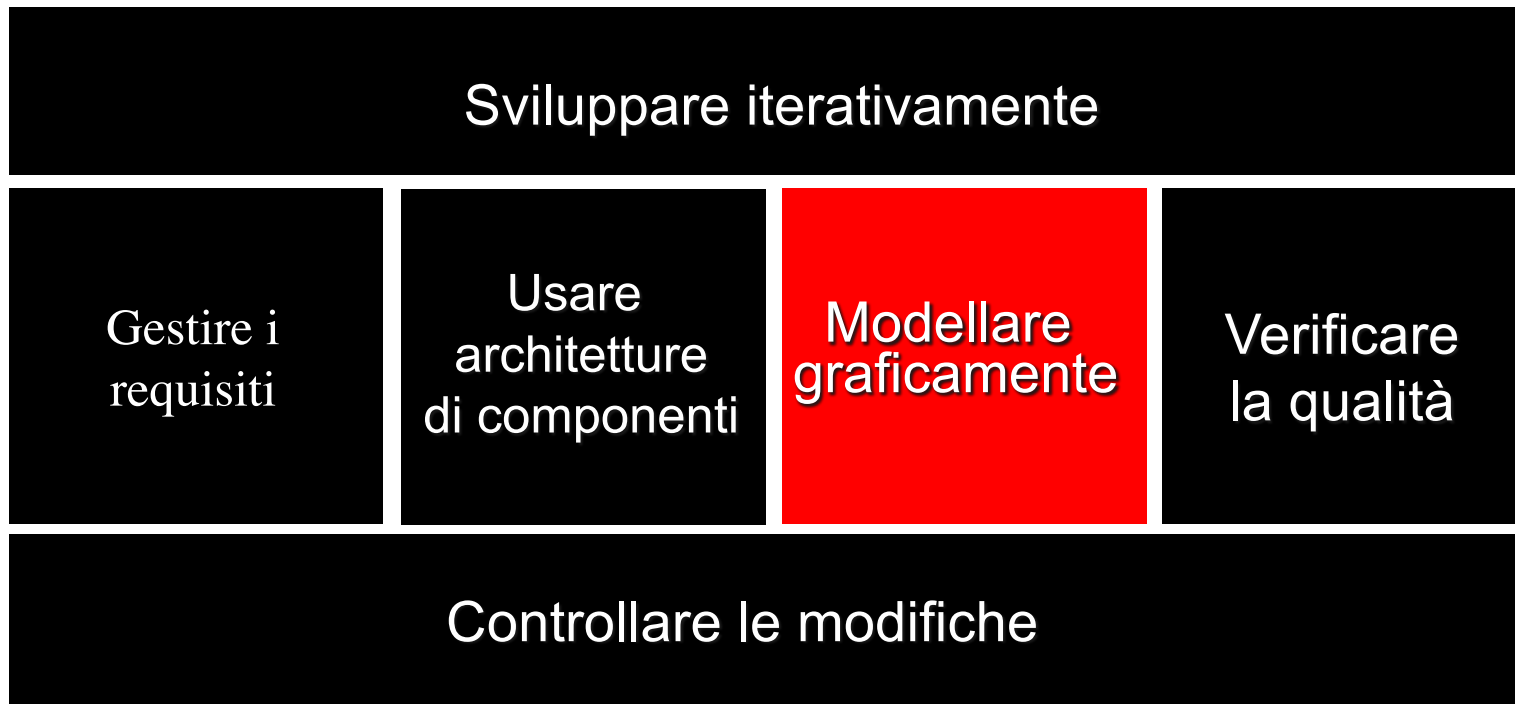


Prof. Paolo Ciancarini
Corso di Ingegneria del Software
CdL Informatica
Università di Bologna

Obiettivi della lezione

- Che cos'è la **progettazione orientata agli oggetti**?
- Come si **inizia** a progettare un sistema object oriented?
- Come si **descrive** un sistema object oriented?

Principi guida dello sviluppo software

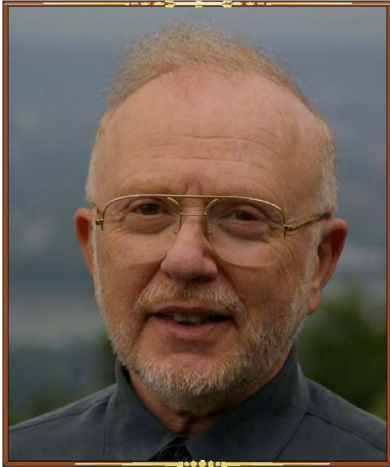


Agenda

- Linguaggi di programmazione ad oggetti
- Analisi e design orientati agli oggetti
- La progettazione guidata dalle responsabilità
- Il processo di progettazione secondo Larman

Chi sono costoro?

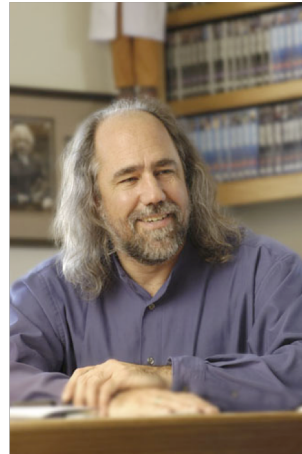
(e cosa hanno fatto?)



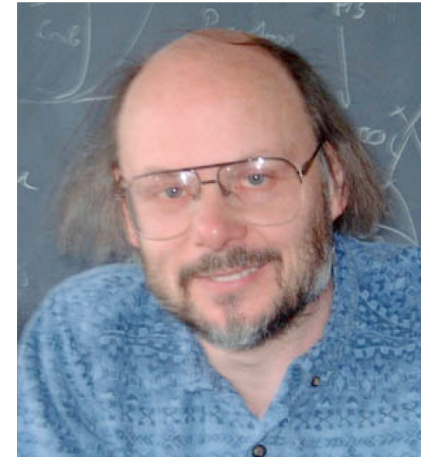
David Parnas



Alan Kay



Grady Booch



Bjarne Stroustrup



Dahl e Nygaard



James Gosling



Anders Hejlsberg



Rebecca Wirfs-Brock

Cosa hanno fatto?



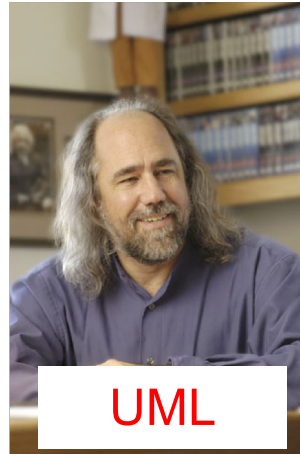
incapsulamento

David Parnas



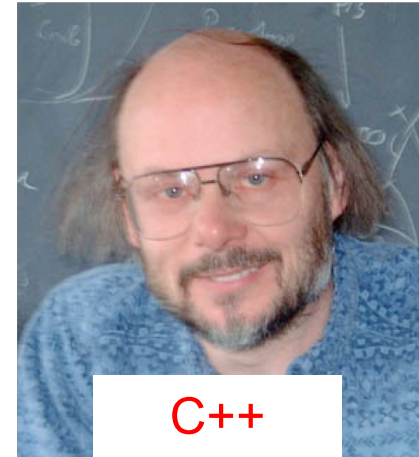
Smalltalk

Alan Kay



UML

Grady Booch



C++

Bjarne Stroustrup



Simula

Dahl e Nygaard



Java

James Gosling



C#

Anders Hejlsberg



Responsibility
driven design

Rebecca Wirfs-Brock

Linguaggi OO: Storia breve

- 1967. Simula 67 - linguaggio per simulazione
- 1980. Smalltalk e C++: primi linguaggi basati su oggetti
- 1984. Object Pascal viene usato da Apple per MacOS
- 1986. Eiffel: design by contract
- 1987. Objective-C viene usate per scrivere Next-OS
- 1989. nasce Object Management Group (OMG)
- 1990. Primi metodi di analisi e progettazione a oggetti
- 1991. Python (scripting language) e CORBA (middleware per oggetti distribuiti)
- 1995. Java e JavaScript: gli oggetti vanno in rete
- 1997. UML 1.1 diventa uno standard OMG
- 2000. C# risposta di Microsoft a Java
- 2003. Scala: combinazione dei paradigmi OO e funzionale
- 2005. UML 2.0 viene definito da OMG
- 2009. Go, introdotto da Google
- 2014. Swift, introdotto da Apple

Principali linguaggi OO

- Simula (anni '60)
- Smalltalk (fine anni '70)
- C++ (inizio anni '80)
- Objective C (fine anni '80)
- Eiffel (fine anni '80)
- Visual Basic
- Python (inizio anni '90)
- Delphi (Object Pascal, TurboPascal, 1995)
- Java (1995)
- C# (2000)
- Swift (2014)

Linguaggi più popolari

Oct 2017	Oct 2016	Change	Programming Language	Ratings	Change
1	1		Java	12.431%	-6.37%
2	2		C	8.374%	-1.46%
3	3		C++	5.007%	-0.79%
4	4		C#	3.858%	-0.51%
5	5		Python	3.803%	+0.03%
6	6		JavaScript	3.010%	+0.26%
7	7		PHP	2.790%	+0.05%
8	8		Visual Basic .NET	2.735%	+0.08%
9	11	▲	Assembly language	2.374%	+0.14%
10	13	▲	Ruby	2.324%	+0.32%
11	15	▲▲	Delphi/Object Pascal	2.180%	+0.31%
12	9	▼	Perl	1.963%	-0.53%
13	19	▲▲	MATLAB	1.880%	+0.26%
14	23	▲▲	Scratch	1.819%	+0.69%
15	18	▲	R	1.684%	-0.06%
16	12	▼▼	Swift	1.668%	-0.34%
17	10	▼▼	Objective-C	1.513%	-0.75%
18	14	▼▼	Visual Basic	1.420%	-0.57%
19	20	▲	PL/SQL	1.408%	-0.12%
20	16	▼▼	Go	1.357%	-0.45%

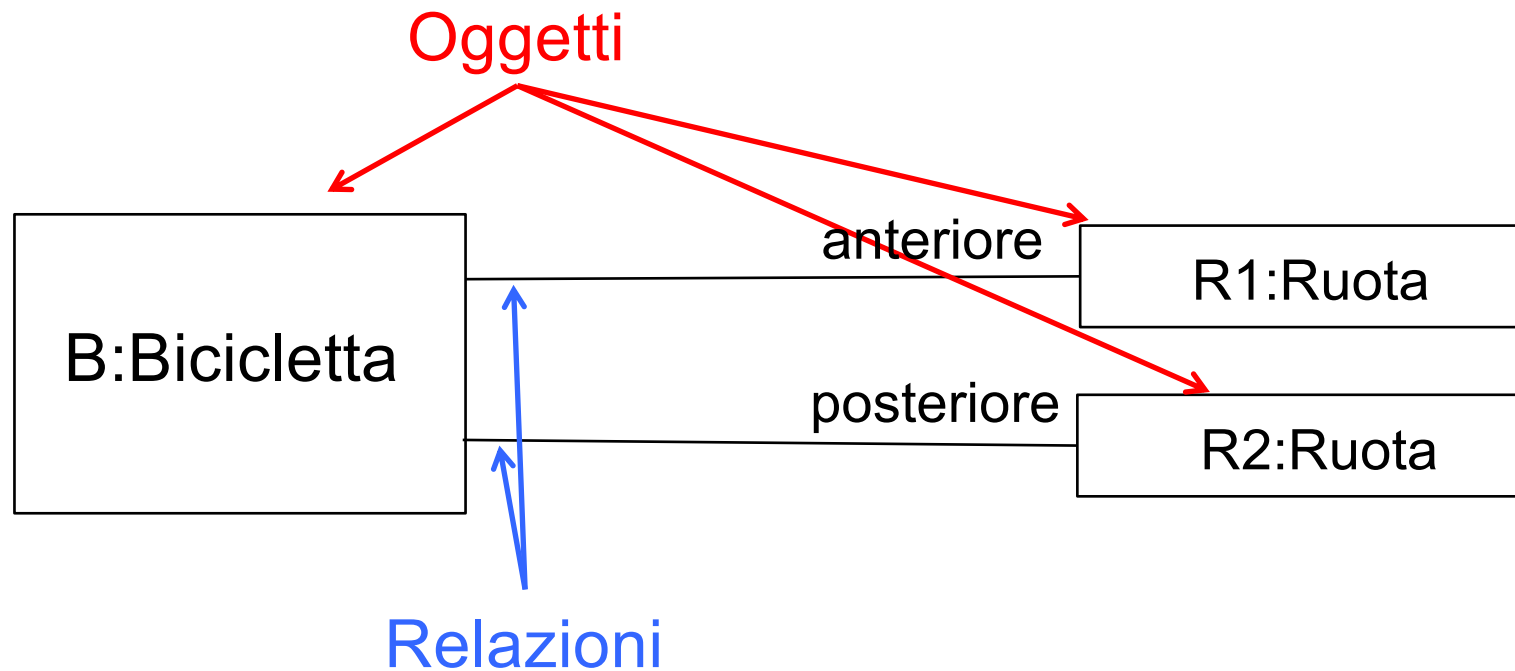
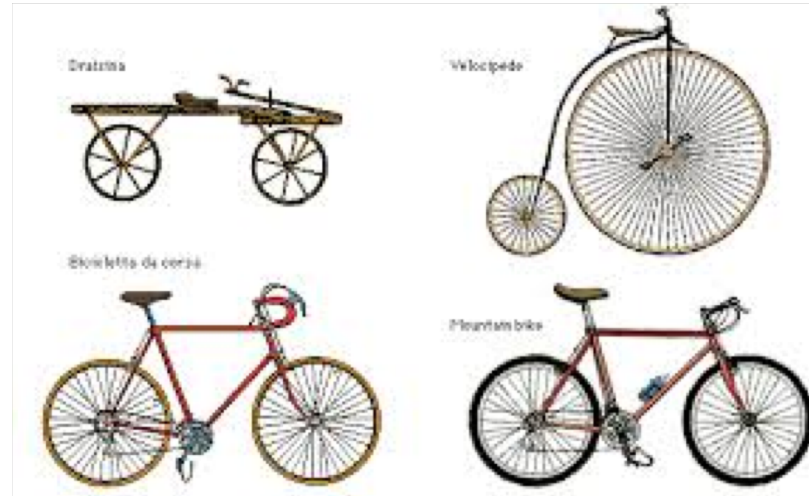
Mamma,
come nascono gli oggetti?



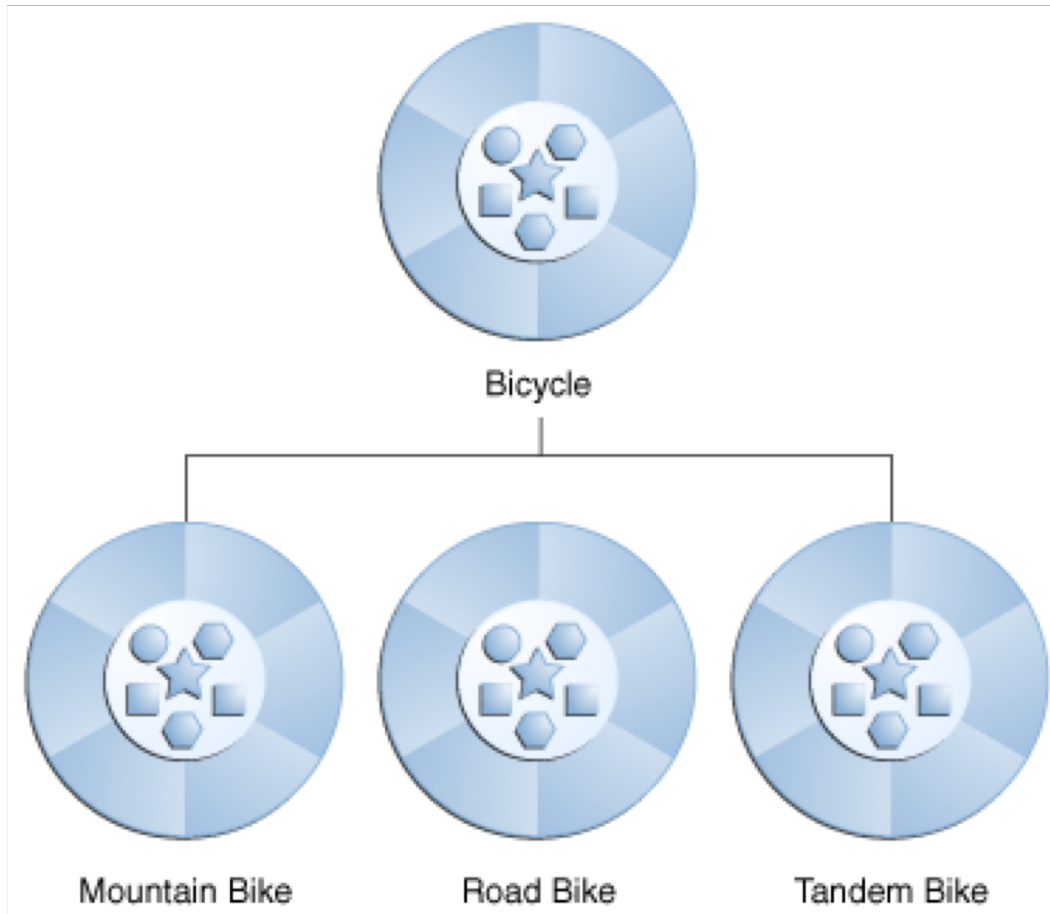
Oggetto

- Un sistema è un insieme di oggetti interagenti, creati da un oggetto iniziale (di solito *main*)
- Ogni oggetto si assume alcune **responsabilità**:
 - offre servizi (ad oggetti che in genere NON conosce)
 - chiede servizi agli oggetti che conosce
 - interagisce in accordo con i termini di un contratto (detto *interfaccia*)

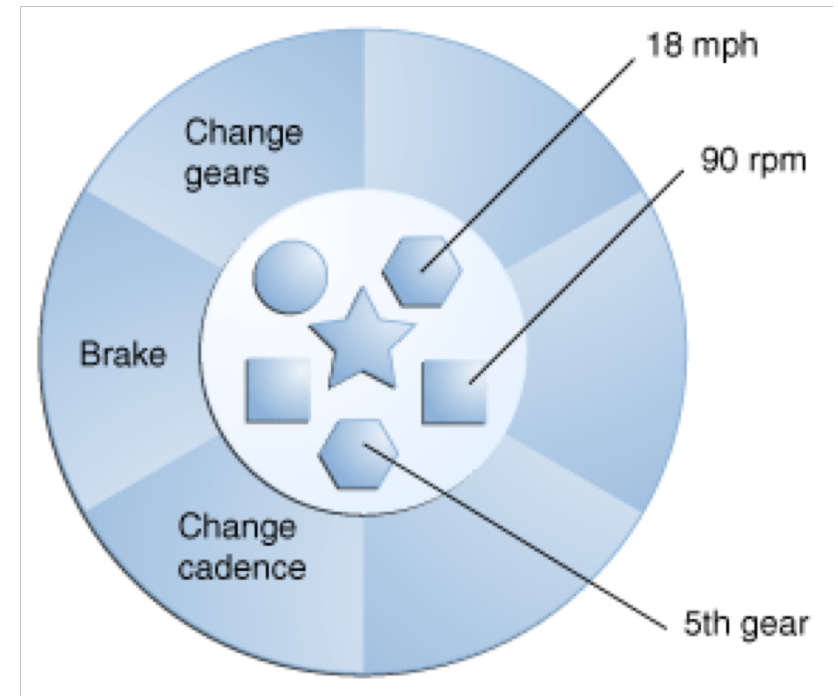
Esempio



Gli oggetti sono descritti da modelli



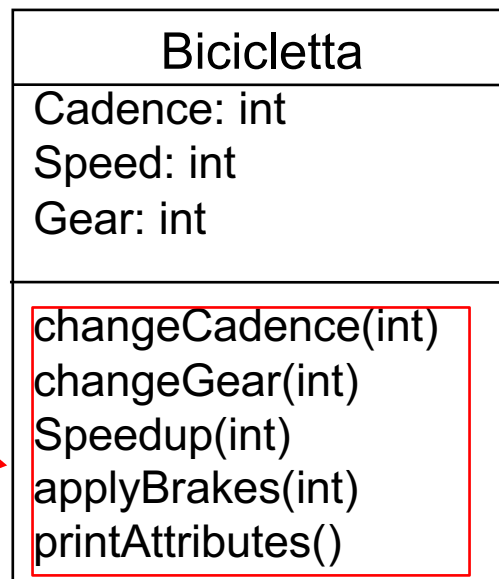
Classi e superclasse



Oggetto

Modello (classe UML)

Interfaccia



```

class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
        cadence = newValue;}

    void changeGear(int newValue) {
        gear = newValue;}

    void speedUp(int increment) {
        speed = speed + increment; }

    void applyBrakes(int decrement) {
        speed = speed – decrement;}

    void printStates() {
        System.out.println("cadence:" +
            cadence + " speed:" +
            speed + " gear:" + gear);}
}

```

Modello (codice Java)

```

class BicycleDemo {
    public static void main(String[] args) {

        // Crea due oggetti Bicycle
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoca metodi dei due oggetti
        bike1.changeCadence(90);
        bike1.speedUp(18);
        bike1.changeGear(5);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}

```

UMPLE



Draw on the right, write (Umple) model code on the left. Generate Java, C++, PHP or Ruby code
Visit [the User Manual](#) or [the Umple Home Page](#) for help. [Download Umple](#) [Report an Issue](#)

Line=

[Create Bookmarkable URL](#)

```
1 class Bicycle {
2   int cadence = 0;
3   int speed = 0;
4   int gear = 1;
5
6   void changeCadence(int newValue) {
7     cadence = newValue;}
8
9   void changeGear(int newValue) {
10    gear = newValue;}
11
12   void speedUp(int increment) {
13     speed = speed + increment; }
14
15   void applyBrakes(int decrement) {
16     speed = speed - decrement;}
17
18   void printStates() {
19     System.out.println("cadence:" +
20       cadence + " speed:" +
21       speed + " gear:" + gear);}
22 }
23
```

SAVE & RESET

TOOLS

LOAD

Class Diagrams

Select Example

DRAW

- Class
- Association
- Generalization
- Delete
- Undo
- Redo
- Sync-Diagram

GENERATE

Java Code

Generate Code

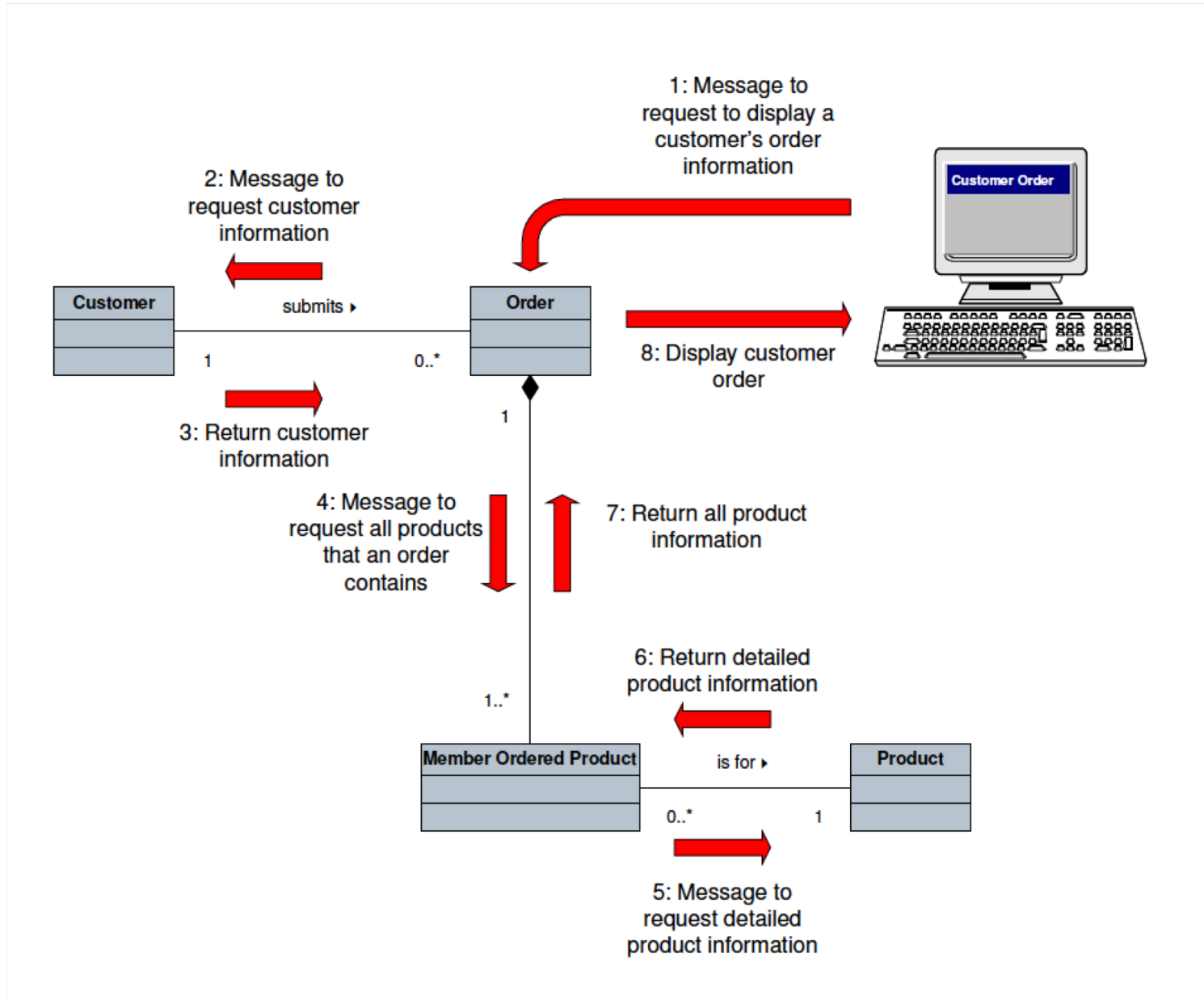
Bicycle

cadence : int ×
speed : int ×
gear : int ×
-- Add More --

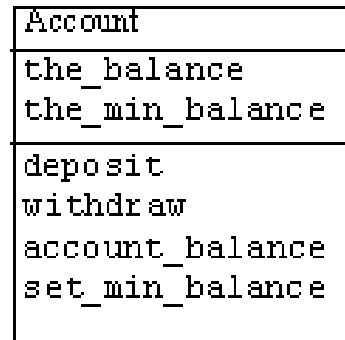
Classi, oggetti, messaggi

- Un linguaggio ad oggetti include sempre un *tipo di modulo* chiamato **classe**, che è uno **schema dichiarativo** che definisce *tipi di oggetti*
- La dichiarazione di classe *incapsula* (cioè nasconde) la definizione dei campi e dei *metodi* degli oggetti creabili come *istanza* della classe
- A tempo di esecuzione, un programma crea *istanze* delle classi chiamate **oggetti**
- Gli oggetti si scambiano **messaggi**; ogni messaggio invoca un “metodo” dell’oggetto ricevente

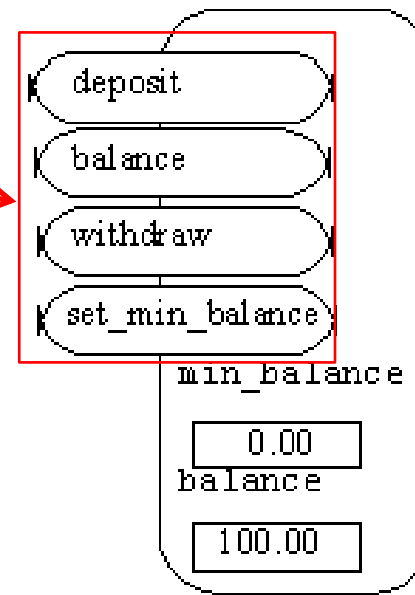
Responsabilità: la dinamica degli oggetti



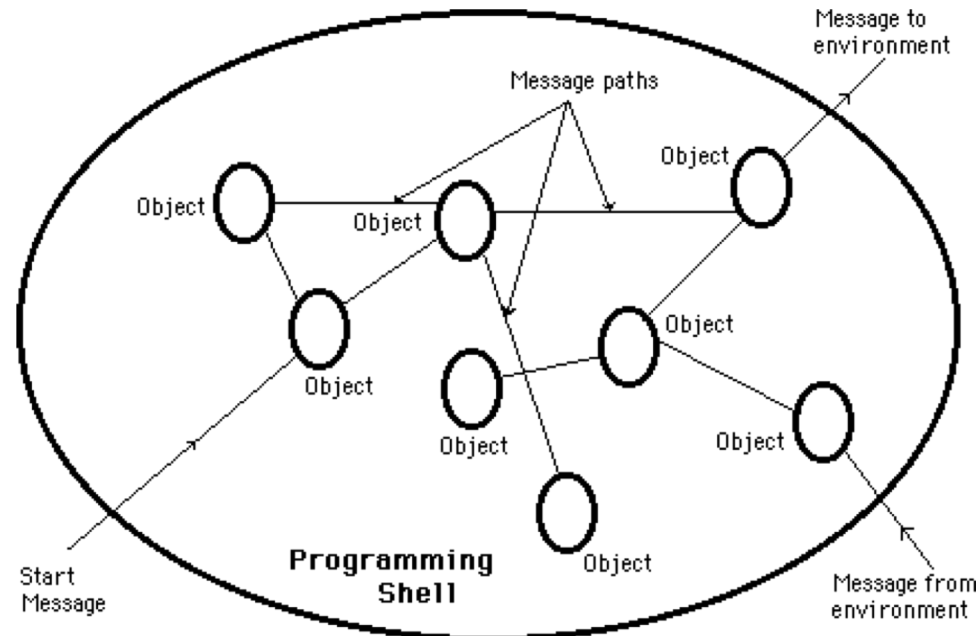
interfaccia



classe



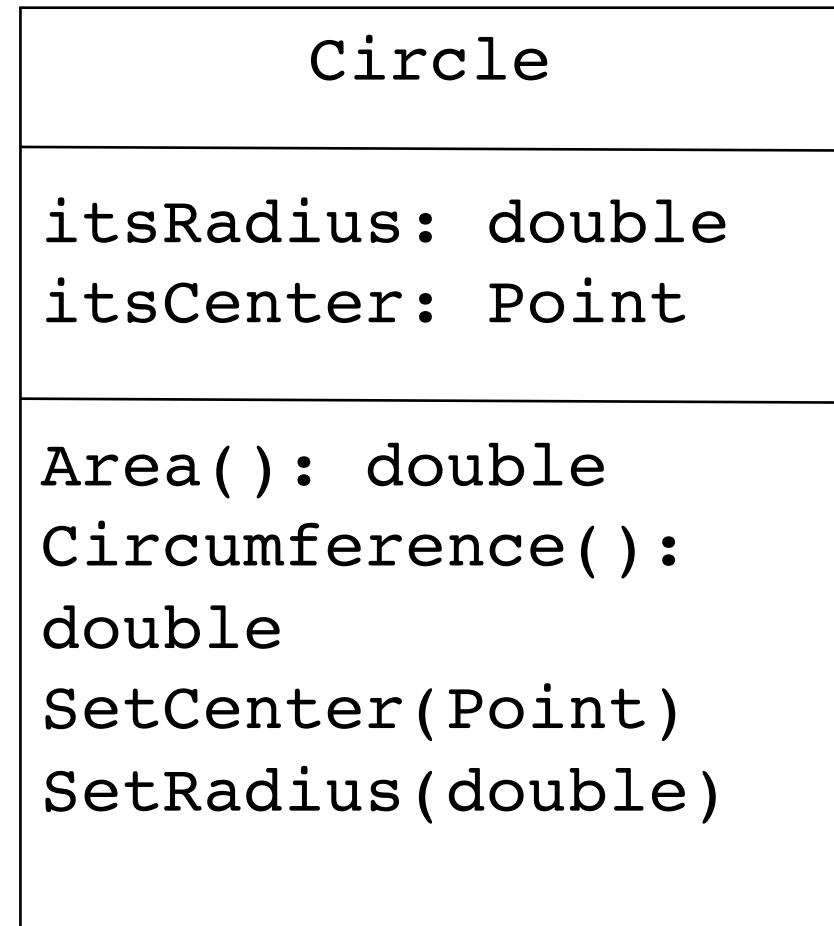
oggetto



“snapshot” di un sistema ad oggetti (in esecuzione)

L'icona di classe in UML

- Definisce
 - Uno stato persistente
 - Un comportamento
- La classe ha
 - Nome
 - Attributi
 - Operazioni
- Ha una rappresentazione grafica in forma di un rettangolo diviso in tre parti



Terminologia

- **Classe.** Tipo di tutti gli oggetti che hanno gli stessi metodi e attributi di istanza
- **Oggetto (o istanza):** **Entità strutturata** (codice, dati) e con stato la cui struttura e stato è **invisibile** all'esterno dell'oggetto
- **Attributi (o variabili) di istanza:** Variabili nascoste nell'oggetto (**incapsulate**) che rappresentano il suo stato interno
- **Stato:** Lo stato di un oggetto (=le sue variabili) si accede e manipola solamente mediante messaggi che invocano i metodi dell'oggetto stesso
- **Messaggio** Richiesta da un oggetto A ad un oggetto B che invoca uno dei metodi di B; A si blocca in attesa della risposta
- **Metodo** Operazione che accede o manipola lo stato interno dell'oggetto (di solito le variabili di istanza). L'implementazione di tale operazione è nascosta a chi la invoca

Metodi di analisi OO

- Analisi OO: inizio di un processo di sviluppo OO
- Metodi di analisi OO (inizio anni 90):
 - Booch: “modello a oggetti” astratto
 - Rumbaugh: OMT (Object Modeling Technique) notazione per modellazione strutturale, dinamica e funzionale
 - Jacobson: OOSE (Object Oriented Software Engineering) metodo di sviluppo basato sui casi d’uso
 - Coad and Yourdon: enfasi sui livelli della modellazione
 - Wirfs-Brock: analisi e progetto sono un continuum
- Metodi simili, con differenze noiose
- Booch, Rumbaugh e Jacobson si allearono nel 1994 per creare UML (ed il RUP)

Che cos'è la progettazione OO?

Orientato agli oggetti:

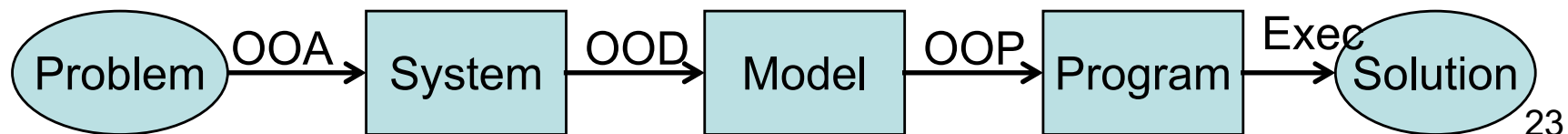
- Decomposizione di un sistema mediante astrazioni di oggetto
- Diversa dalla decomposizione funzionale/procedurale

OO Design, Analysis e Programming:

OOA: Esaminare e decomporre il problema... *analysis*

OOD: Disegnare un modello del problema... *design*

OOP: Realizzare il modello... *programming*



Discussione

- Da dove si comincia a progettare?

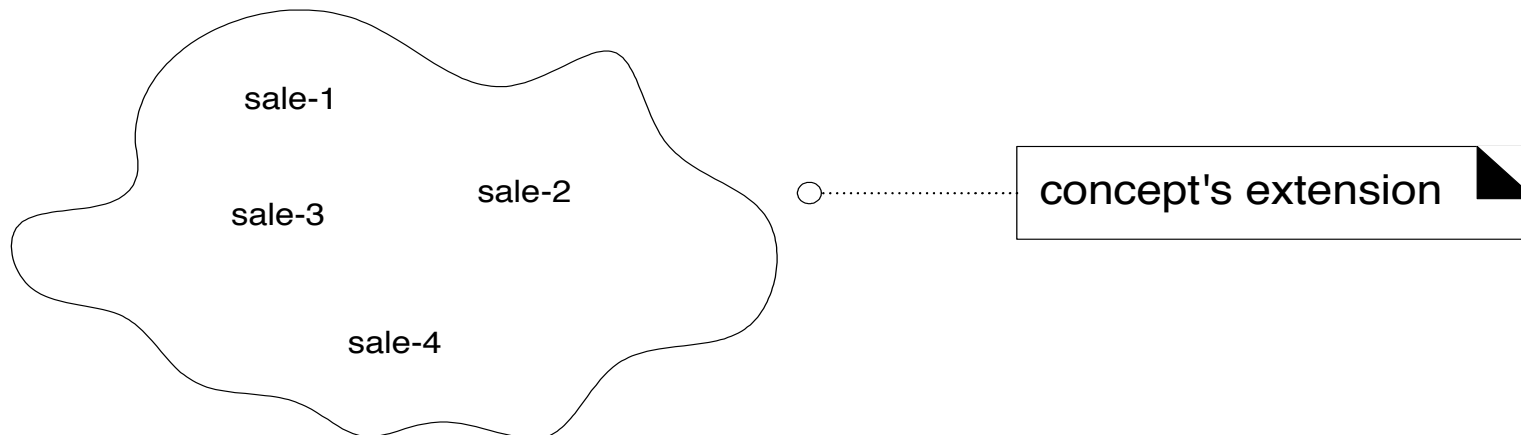
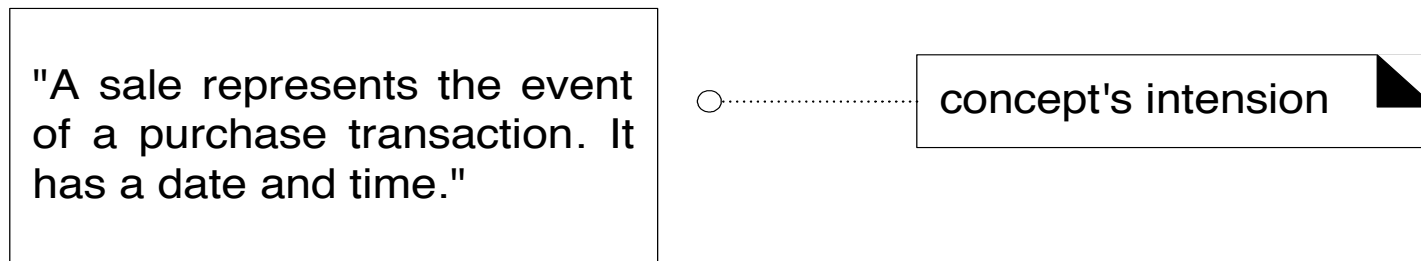
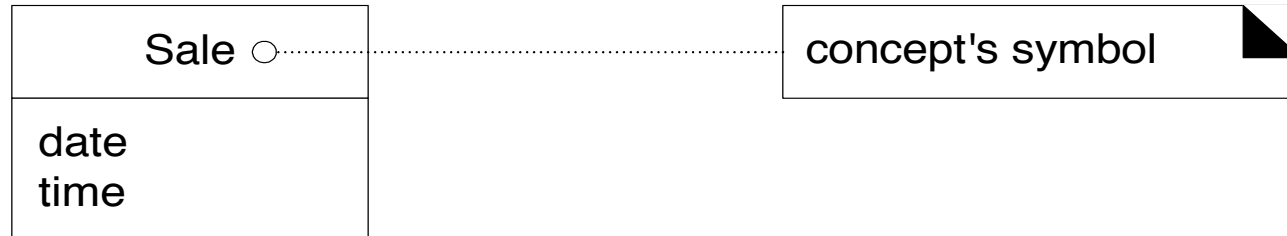


Il paradigma OO

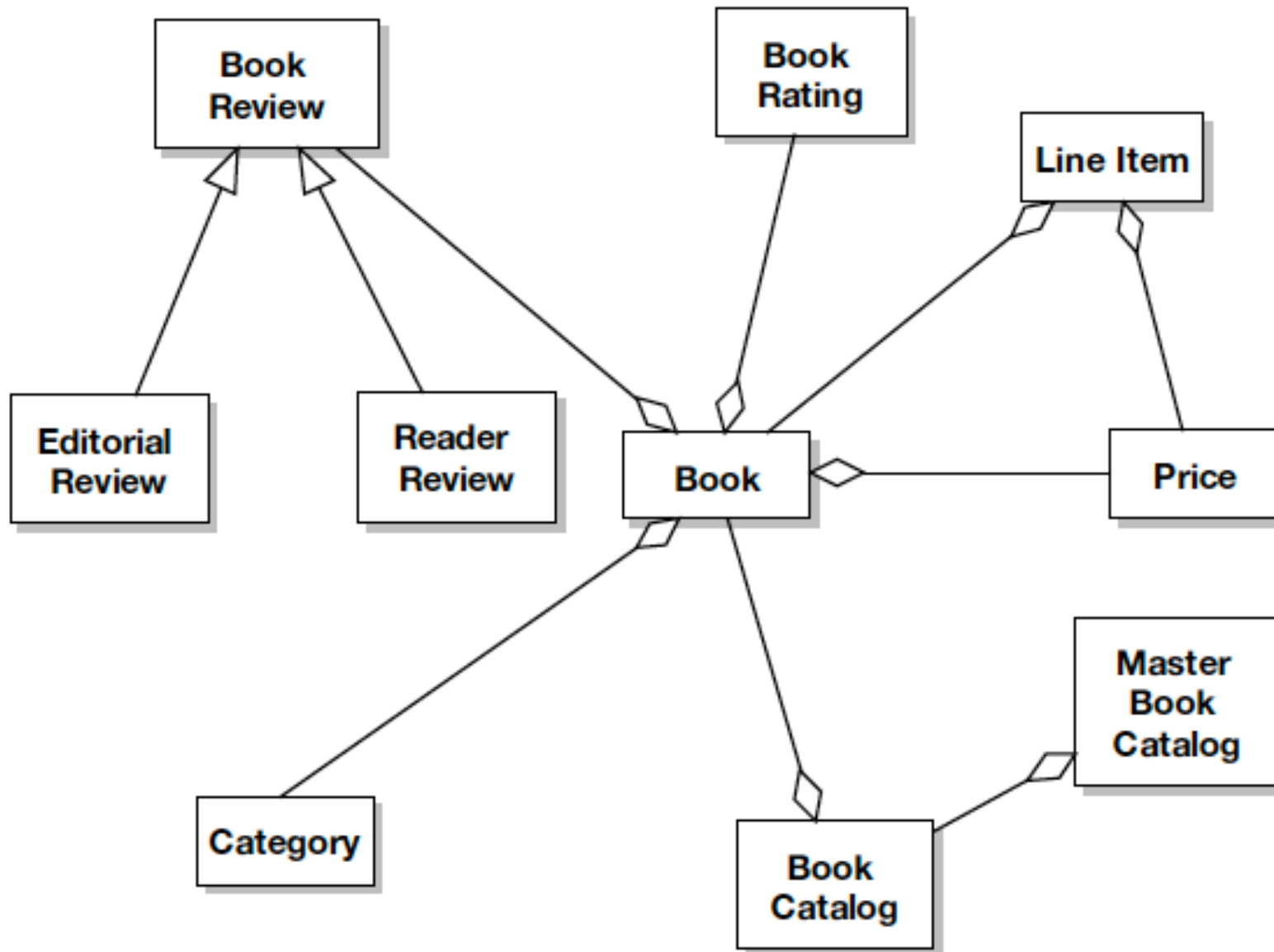
I metodi di sviluppo OO includono le seguenti attività:

1. Analisi del **dominio** del problema e requisiti
2. Identificazione di **scenari** e casi d'uso
3. Disegno delle classi concettuali (**modello di dominio**)
4. Identificazione di attributi e metodi
5. Disegno di una **gerarchia** di classi
6. Costruzione di un modello **statico** di oggetti e relazioni
7. Costruzione di un modello **dinamico** degli oggetti
8. Revisione dei due modelli rispetto ai casi d'uso
9. Iterare se necessario

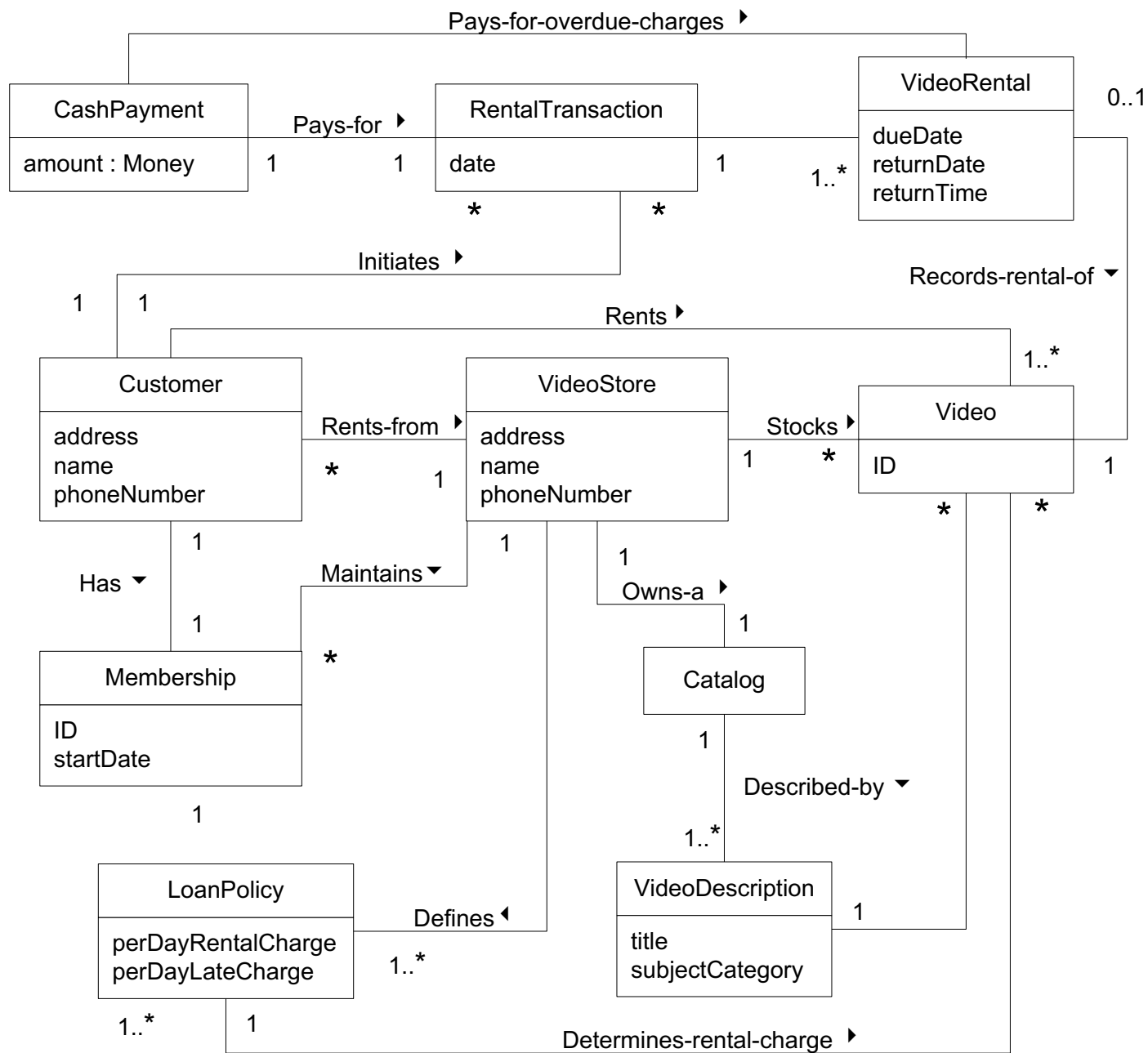
Classi concettuali



Un modello di dominio

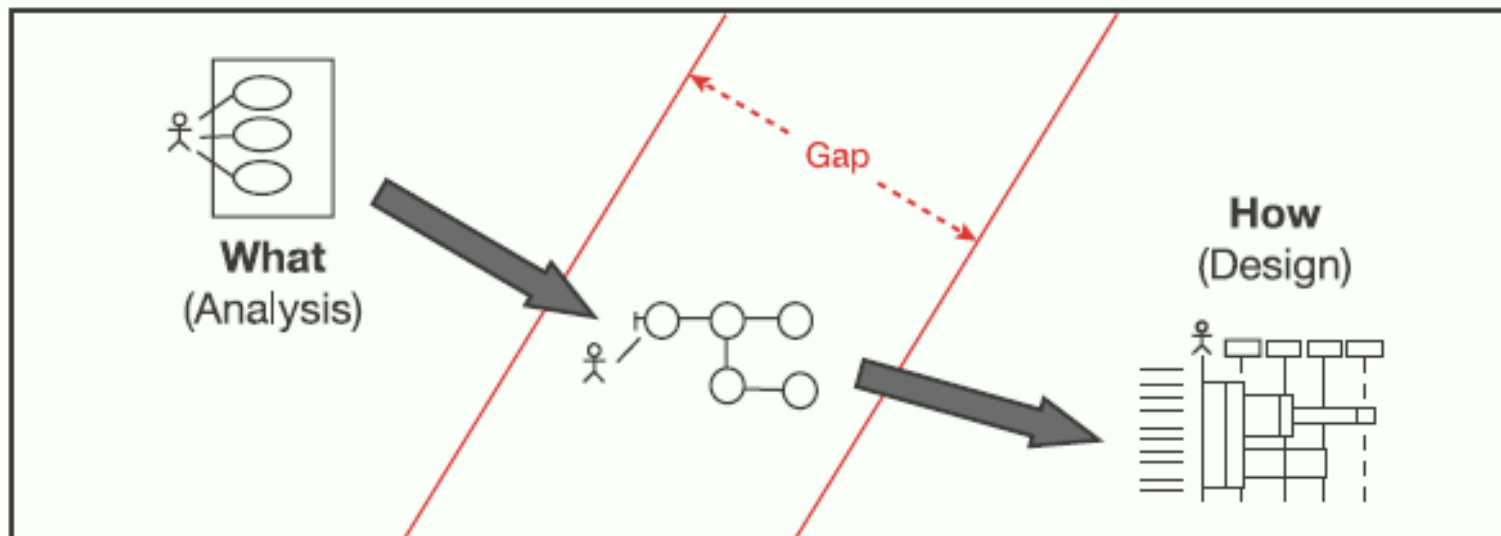


Altro esempio di Modello di Dominio



Modello del dominio (Domain Model)

- Un **modello del dominio** visualizza solo le entità importanti di un dominio
 - E' una sorta di “dizionario visuale”
 - *Non* rappresenta le classi del programma finale
- Aiuta a visualizzare le informazioni più critiche sul sistema, ed a trasformarle ove necessario
- E' fonte d'ispirazione per disegnare poi le classi del sistema, riducendo il gap, ovvero il “salto di rappresentazione” dai requisiti all'architettura

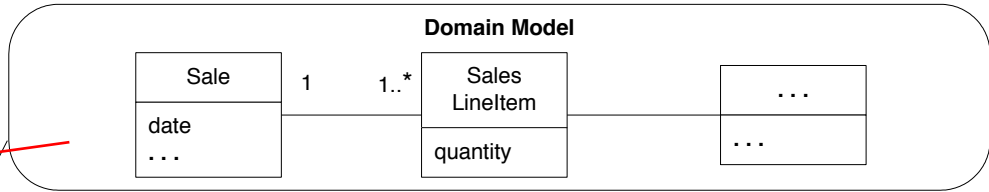


Sample UP Artifact Relationships

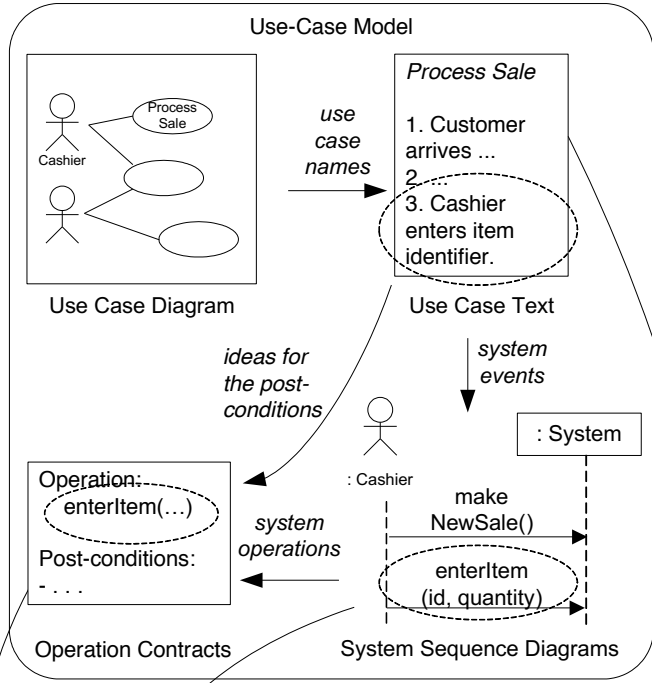
Modello di dominio

Modello di progetto

Business Modeling



Requirements



inspiration for names of some software domain objects

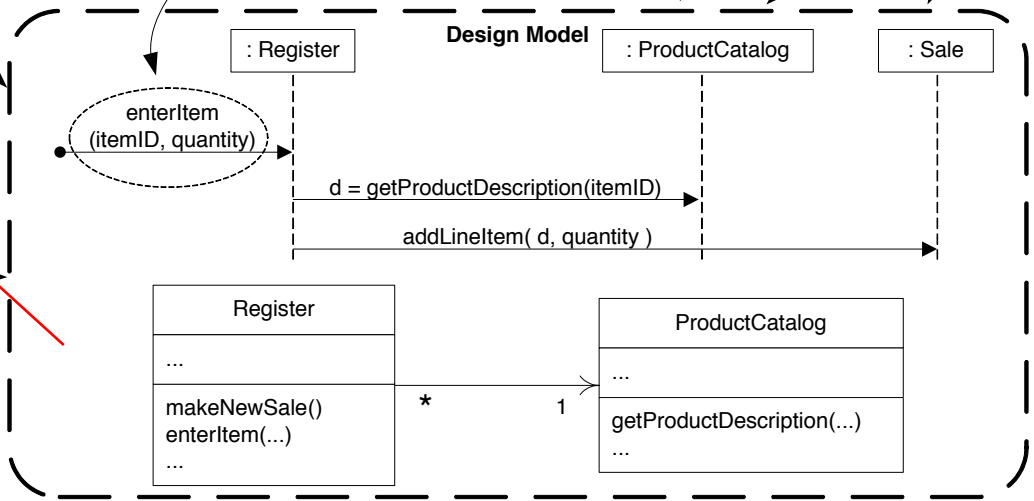
starting events to design for, and detailed post-condition to satisfy

Supplementary Specification
non-functional requirements
domain rules

functional requirements that must be realized by the objects

Glossary
item details, formats, validation

Design



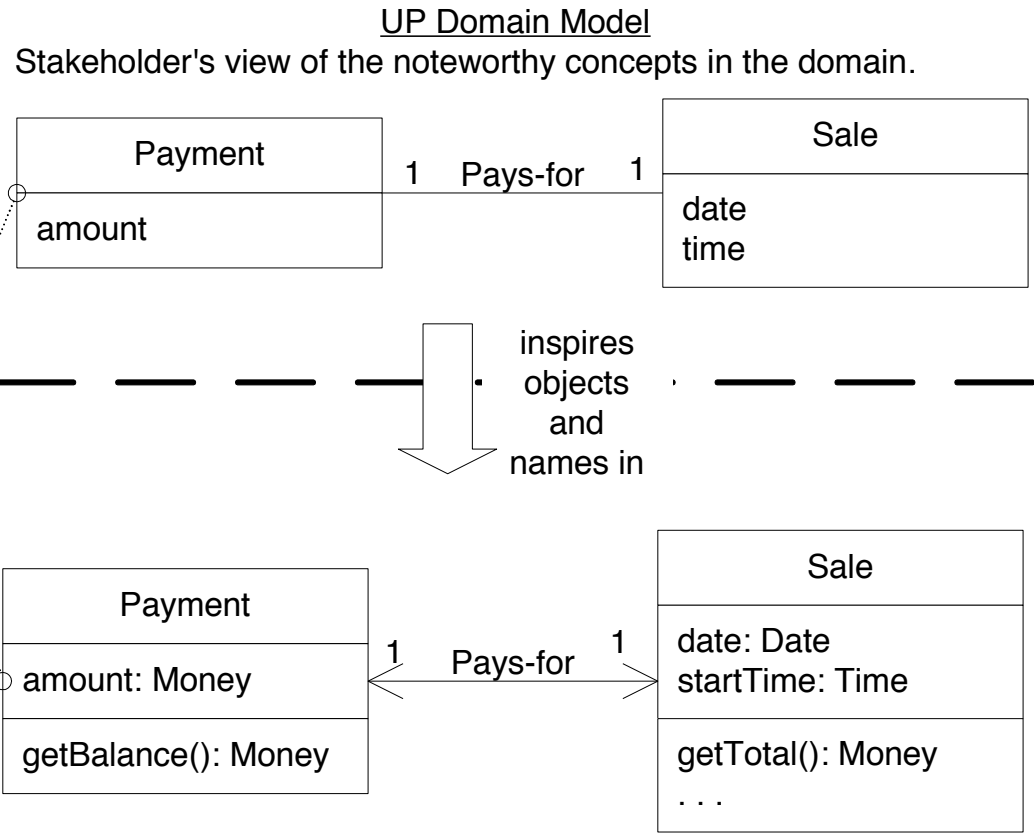
Il processo secondo Larman

Raffinamento: dal modello di dominio a quello di progetto

A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former *inspired* the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.



Domain layer of the architecture in the UP Design Model

The object-oriented developer has taken inspiration from the real world domain in creating software classes.

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

Responsibility-Driven Design (RDD)

- La progettazione guidata dalle responsabilità è una tecnica in cui le “responsabilità” di un oggetto guidano il suo design
- Si concentra sul ruolo di un oggetto in un sistema e su come il suo comportamento influenza gli altri oggetti
- Se si comincia a elencare le responsabilità di un oggetto poi è più semplice
 - Progettare il suo funzionamento interno
 - Capire quali altri oggetti collaborano con esso
 - Creare la sua interfaccia pubblica (= come il mondo esterno accede le sue funzioni)
 - Tener conto degli eventi da esso riconosciuti

Prospettive di modellazione

Esempio: Prospettive di modellazione di un cavallo

- *Vista strutturale:*
ha un corpo, una coda, quattro zampe (*componenti*)
- *Vista comportamentale:*
cammina, mangia, emette versi (*attività*)
- *Vista delle responsabilità:*
trasporta persone/merci, corre in ippodromo (*scopo entro un sistema*)

(Rebecca Wirfs-Brock)

Esempio: programma capace di giocare a scacchi

- **Struttura:**
 - La scacchiera (matrice di pezzi) e la partita (lista di mosse)
 - L'albero di gioco (per la strategia)
- **Comportamenti (o funzioni):**
 - Interfaccia utente grafica
 - Generazione delle mosse possibili
 - Scelta della mossa (funzione di valutazione)
- **Responsabilità:**
 - giocare “bene”, impegnando un giocatore a livello di “maestro”
 - giocare “sufficientemente bene” da impegnare un novizio
 - insegnare a giocare a scacchi ad un bambino
 - agire da interfaccia per giocare in rete

Si noti che queste sono **responsabilità diverse** che tuttavia usano tutte le stesse funzioni di base (interfaccia, modulo generatore di mosse, ecc.)

RDD: Glossario

- Applicazione = insieme di oggetti interagenti
- Oggetto = implementazione di uno o più ruoli
- Ruolo = insieme coeso di responsabilità
- Responsabilità = obbligo di eseguire un compito o disporre di un'informazione
- Collaborazione = interazione di oggetti e/o ruoli
- Contratto = accordo che definisce i termini di una collaborazione

alistair.cockburn.us/Responsibility-based+modeling

Progettazione guidata dalle responsabilità

Le responsabilità di un oggetto si classificano come:

- **Conoscere**
 - A quali domande deve rispondere?
- **Fare**
 - Quali operazioni deve eseguire?
- **Applicare**
 - Quali regole deve applicare e/o imporre?

Esempio: una festa

- Qualcuno vuole organizzare una festa
- Cosa è necessario? Occorre:
 - sapere chi invita: dove? quando?
 - invitare le persone,
 - ricevere le conferme di partecipazione,
 - decidere la musica da suonare,
 - acquistare bevande e cibo da servire
 - predisporre le pulizie al termine della festa

Quali sono gli oggetti?

- Padroni di casa
 - Ospiti
 - Invito
- } Persona (superclasse)
- Indirizzo
 - » della festa
 - » del padrone di casa
 - » dell'ospite
 - Data
 - Tema
- Cibo e bevande
 - Musica: canzoni, playlist, ecc.
 - Strumenti per pulire la casa: secchio, scopa, paletta, ecc.

Esempio

- Quali sono le responsabilità?
 - **Invitare**: occorre la lista degli (oggetti) invitati
 - **Comprare**: occorre lista della spesa e denaro
 - **Cucinare**: occorre il cibo
 - **Pulire la casa**: occorrono gli strumenti
 - **RSVP**: occorre l'invito
 - **Andare alla festa**: occorre l'indirizzo e la data
- Come assegniamo queste responsabilità?

Approccio

- Definire il problema
- Creare gli scenari d'uso
 - Mediante interviste
 - Concentrarsi sulle operazioni principali
- Usare le schede CRC [Beck&Cunningham 1989]
 - Elencare gli oggetti (il dominio)
 - Iniziare con quelli più importanti
 - Elencare le responsabilità
 - Le cose principali che fanno gli oggetti più importanti
 - Elencare le relazioni con altri oggetti

Schede CRC

- Classe, Responsabilità, Collaborazione
 - responsabilità: compiti da eseguire
 - collaborazioni: altri oggetti che cooperano con questo

Nome della classe:	Superclasse:	Sottoclassi:
Responsabilità:	Collaborazioni:	

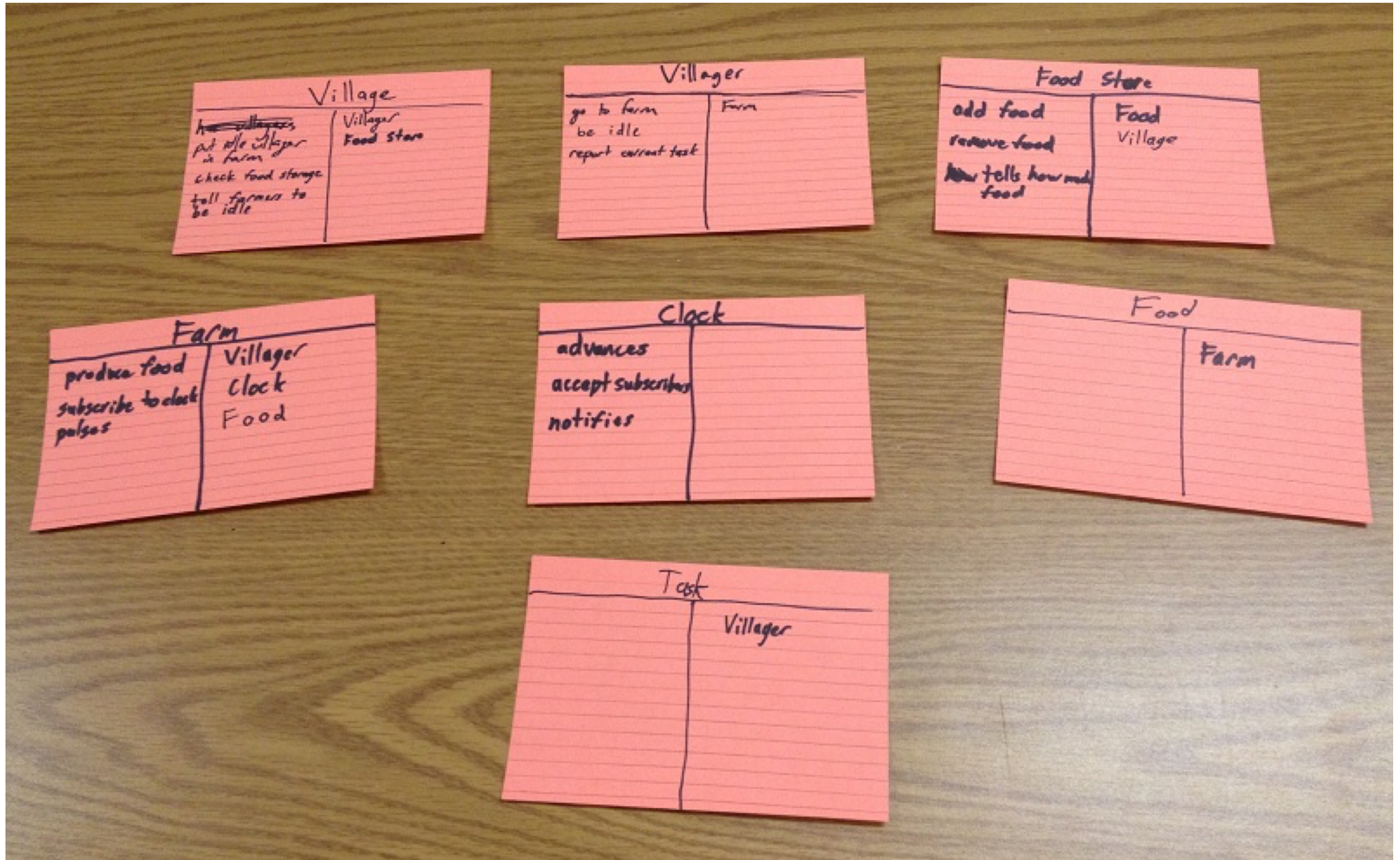
CRC Card: padrone di casa

Class name: Padrone	Superclass: Persona	Subclasses:
Responsibilities:	Collaborations:	
Invita	Invito, Persona, Lista	
Compra	Denaro, Negozio, Cibo, ...	
Pulisce_casa	Spugna, Straccio, ...	

CRC Card: ospite

Class name: Ospite	Superclass: Persona	Subclasses:
Responsibilities:	Collaborations:	
RSVP	Telefono, Padrone	

Un insieme di CRC cards è molto simile ad un Domain Model



Strumenti sw per CRC

The screenshot shows the Eclipse IDE interface with a CRC Card Diagram for a 'Shipment' class. The diagram is displayed in the main editor area, and the left sidebar shows the project structure. The diagram includes the following sections:

- Shipment** (Class Name)
- Super Classes:** (Empty)
- Sub Classes:** (Empty)
- Description:** Hold shipment information
- Attributes:**

Name	Description
ID	The id of shipment
Customer ID	The id of customer
Size	The length, width, height of shipment
Weight	The weight of shipment
Delivery address	The delivery address

- Responsibilities:**

Name	Collaborator
get the size	
get the weight	
get delivery address	

The left sidebar shows the project structure for 'CourierSystem (Courier)*', including 'UML Diagrams', 'Requirements Capturing', 'Textual Analysis (1)', 'Requirement Diagram (2)', 'Basic Diagram (2)', 'CRC Card Diagram (1)', and 'Shipment'. The bottom left shows the 'CRC Card Diagram1 - CRC Car...' properties, including 'Name', 'Parent model', and 'Background'.

Da CRC a UML

- Le schede CRC definiscono le classi principali e le loro associazioni
 - Strumento di brainstorming
 - Se ne scrivono tante, se ne buttano tante
- UML
 - Per raffinare e documentare il progetto
 - Per descrivere il progetto ad altri

Discussione

- Che si fa dopo l'analisi delle classi e delle responsabilità?



Le viste principali

**Modello di dominio,
Diagrammi di classe**

Vista logica

**Codice,
Diagrammi di package**

Vista sviluppo

Diagrammi di collaborazione

Vista casi d'uso
(scenari, SSD)

Diagrammi di deployment

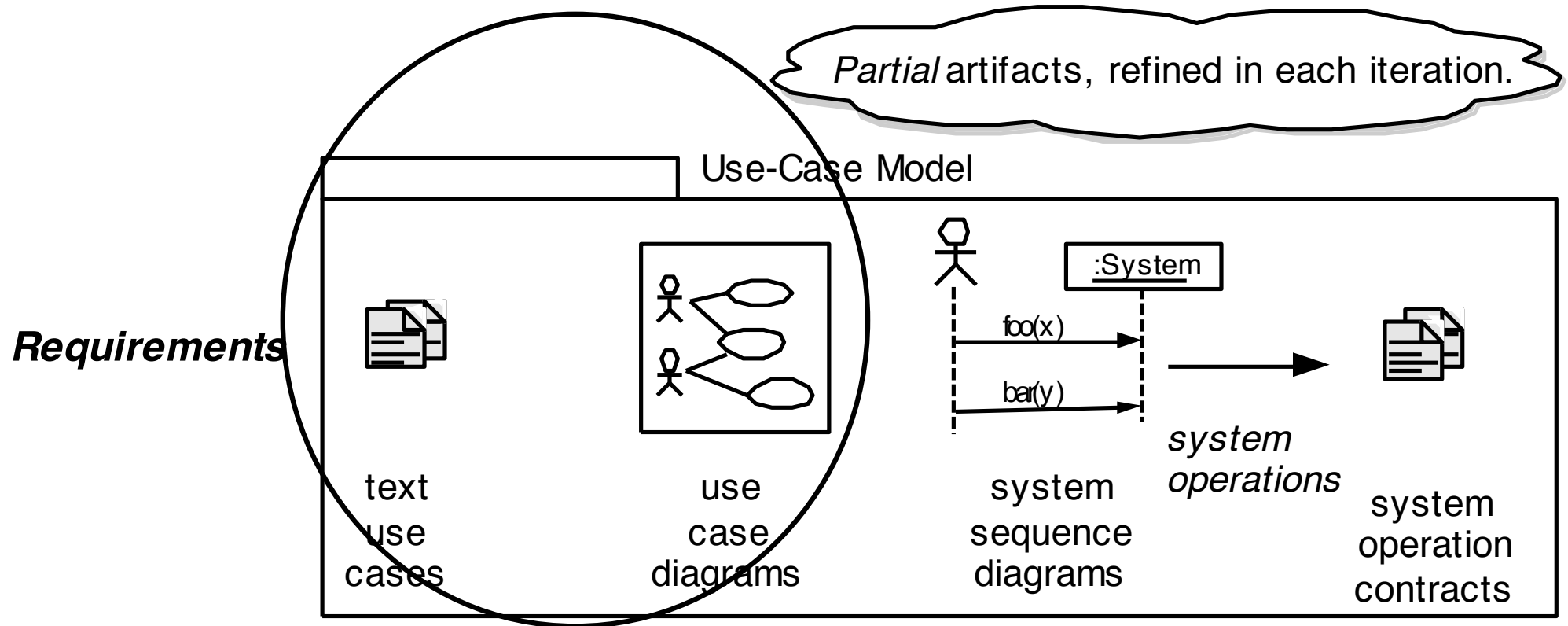
Vista processi

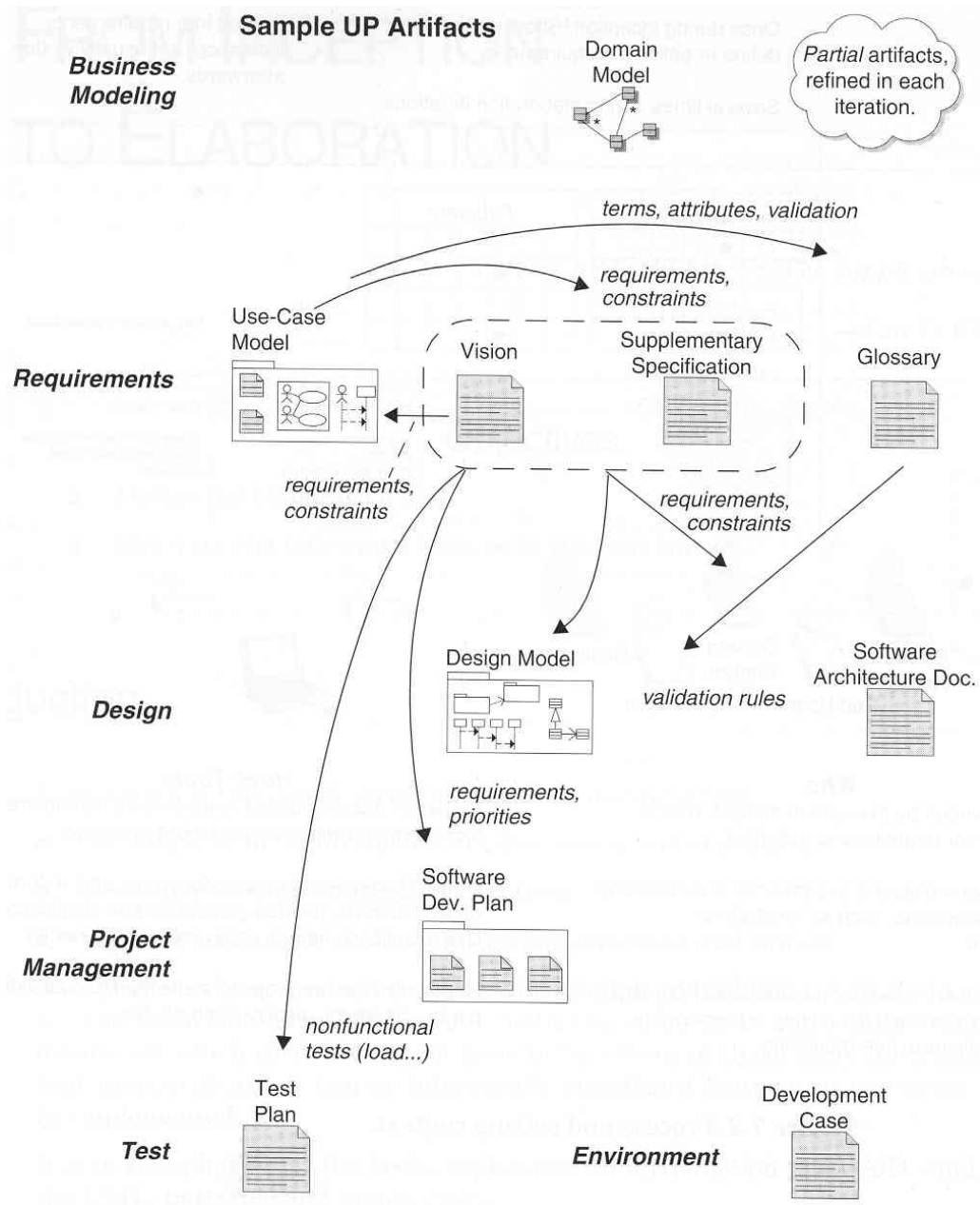
Vista fisica

Il processo secondo Larman (objects by design)

- i. Casi d'uso: definire gli scenari delle interazioni degli utenti col sistema, possibilmente evidenziano il contesto del sistema stesso
- ii. Modello del dominio: usare i nomi nei casi d'uso, stabilire le associazioni
- iii. System sequence diagram (SSD): crearne uno per ciascuno scenario
- iv. "Contratti di sistema": specificare le post-condizioni per gli eventi di sistema negli SSD
- v. Diagramma di collaborazione: assegnare le responsabilità alle classi nel modello di dominio in modo che i contratti siano soddisfatti
- vi. Diagramma delle classi: aggiungere a ciascuna classe i metodi trovati disegnando il diagramma di collaborazione
- vii. Codice: ricavarlo dal diagramma delle classi e da quello di collaborazione

II design secondo Larman

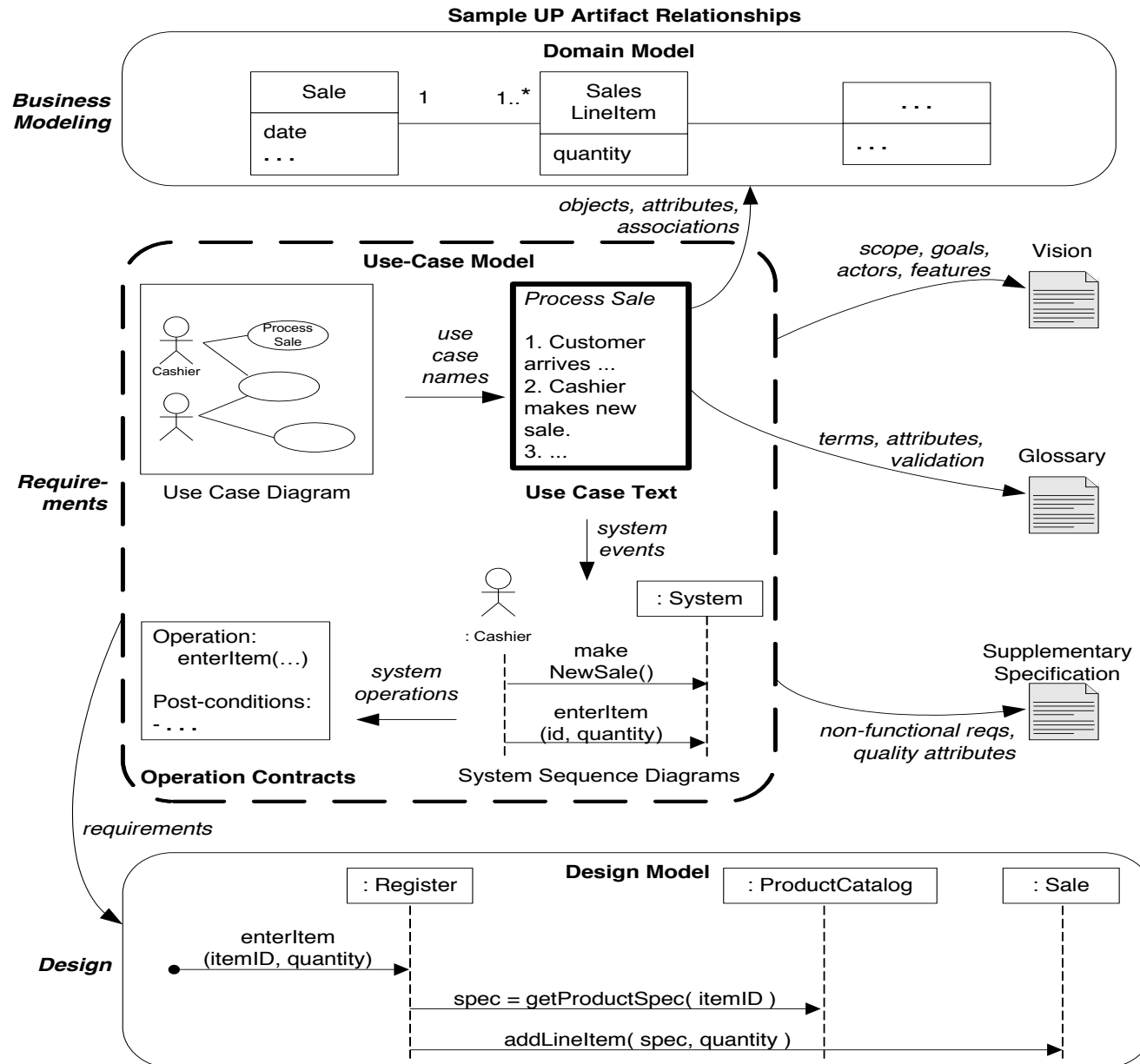




Il design secondo Larman

La figura mostra le dipendenze tra vari artefatti del design secondo Larman

Modello dei casi d'uso



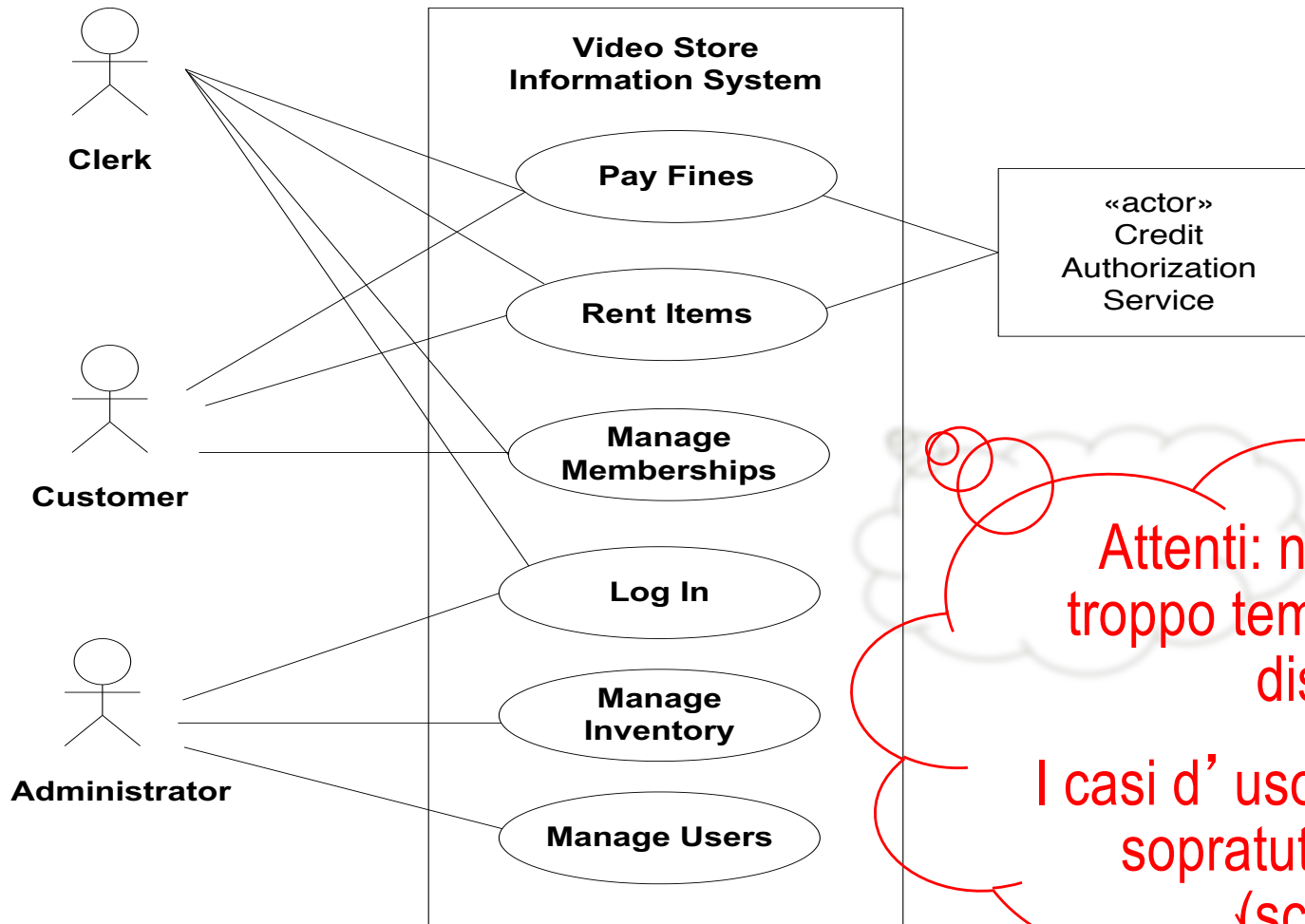
Caso d'uso

- Un caso d'uso è una “storia” (racconto) su un sistema che deve raggiungere uno scopo
 - *Rent Videos*
- Usato da attori primari
 - *Commesso*
 - Sistemi esterni
 - Qualcosa o qualcuno che ha un comportamento
- Usa *attori di servizio*.
 - *CreditAuthorizationSystem*

Scenario

- Uno *scenario* è una specifica sequenza di azioni e interazioni pertinenti in un caso d'uso
 - Una sola sequenza principale
 - Es.: lo scenario di noleggio di un film solo dopo aver pagato more di noleggi precedenti
- Un *caso d'uso* è una collezione di scenario di successo o fallimento che descrivono come un attore principale usa un sistema per conseguire uno scopo

Diagrammi dei casi d'uso



Attenti: non dedicate
troppo tempo a curare i
disegni

I casi d'uso sono descritti
soprattutto da testo
(scenari)

Diagramma di contesto

Un **diagramma di contesto** è la rappresentazione sintetica delle relazioni di un sistema con l'ambiente esterno

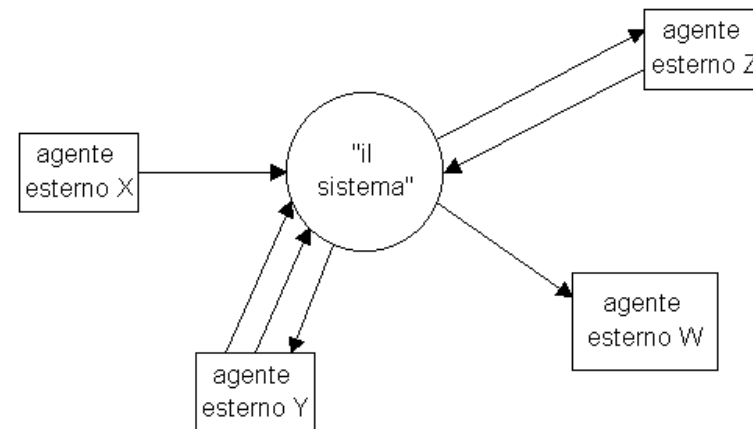


Diagramma di contesto

Summarized from the use case diagram.

Context diagrams come in different formats with varying detail, but all show the major external actors related to a system.

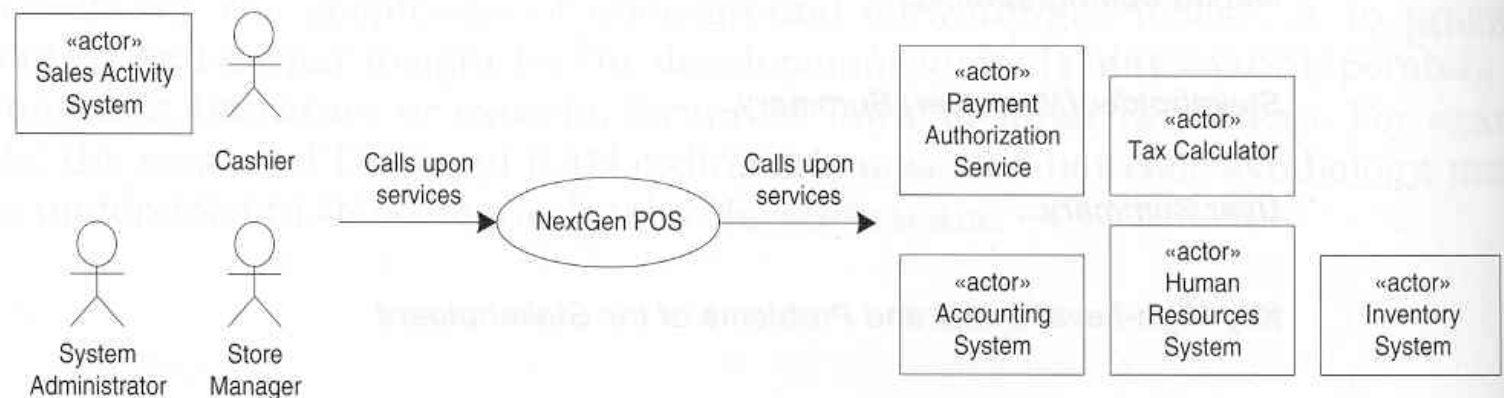
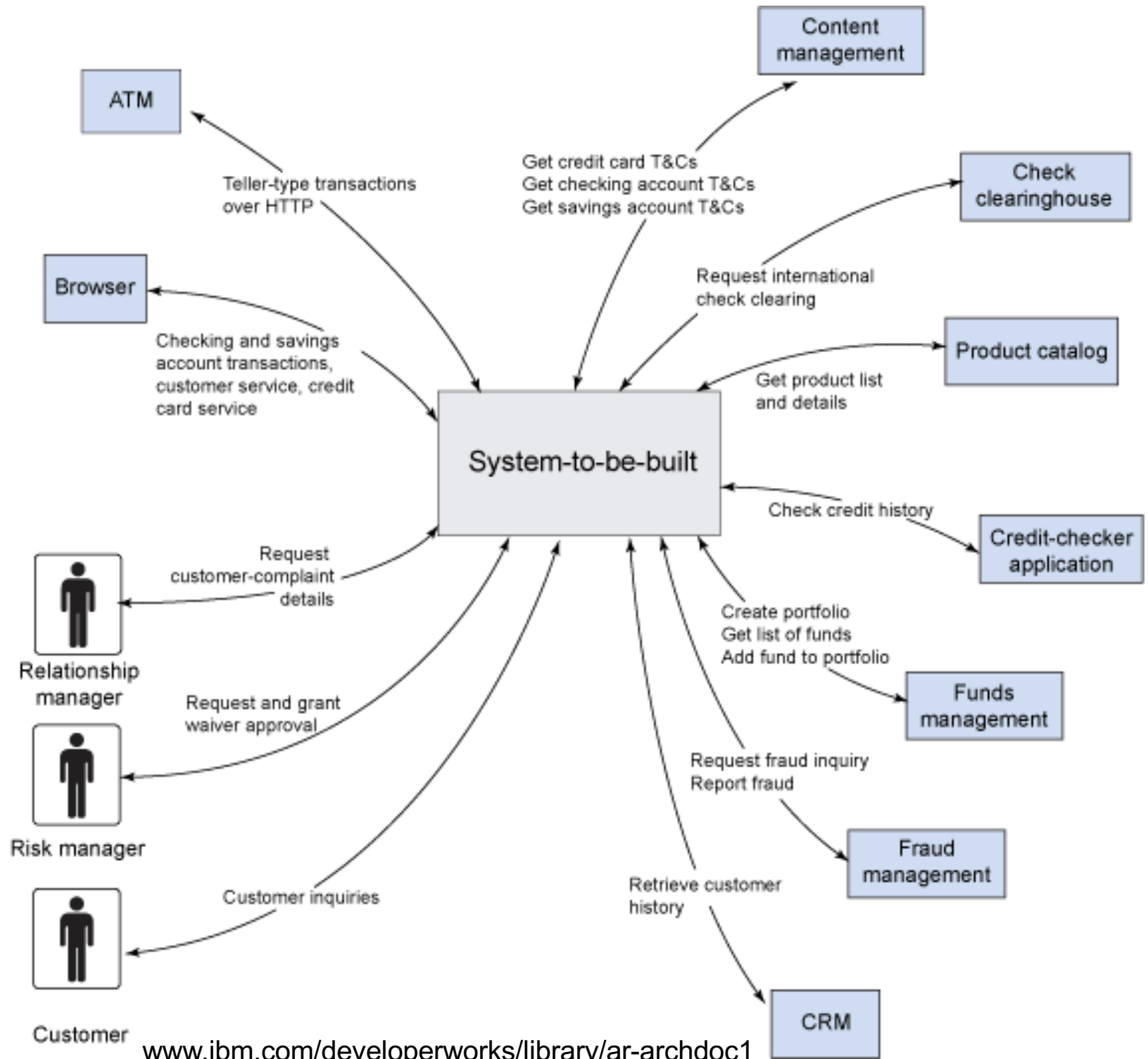
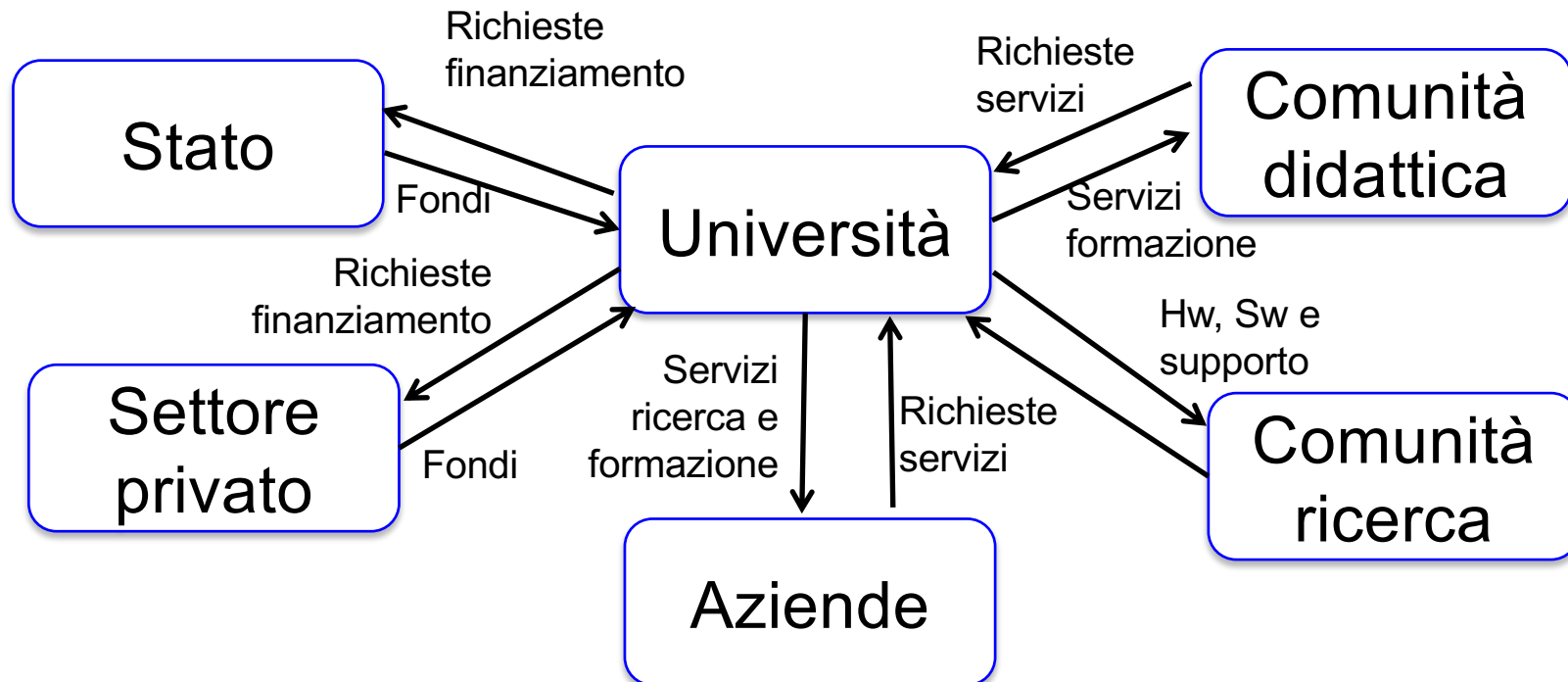


Figure Vision-1. NextGen POS system context diagram

Esempio:
Diagramma
di contesto



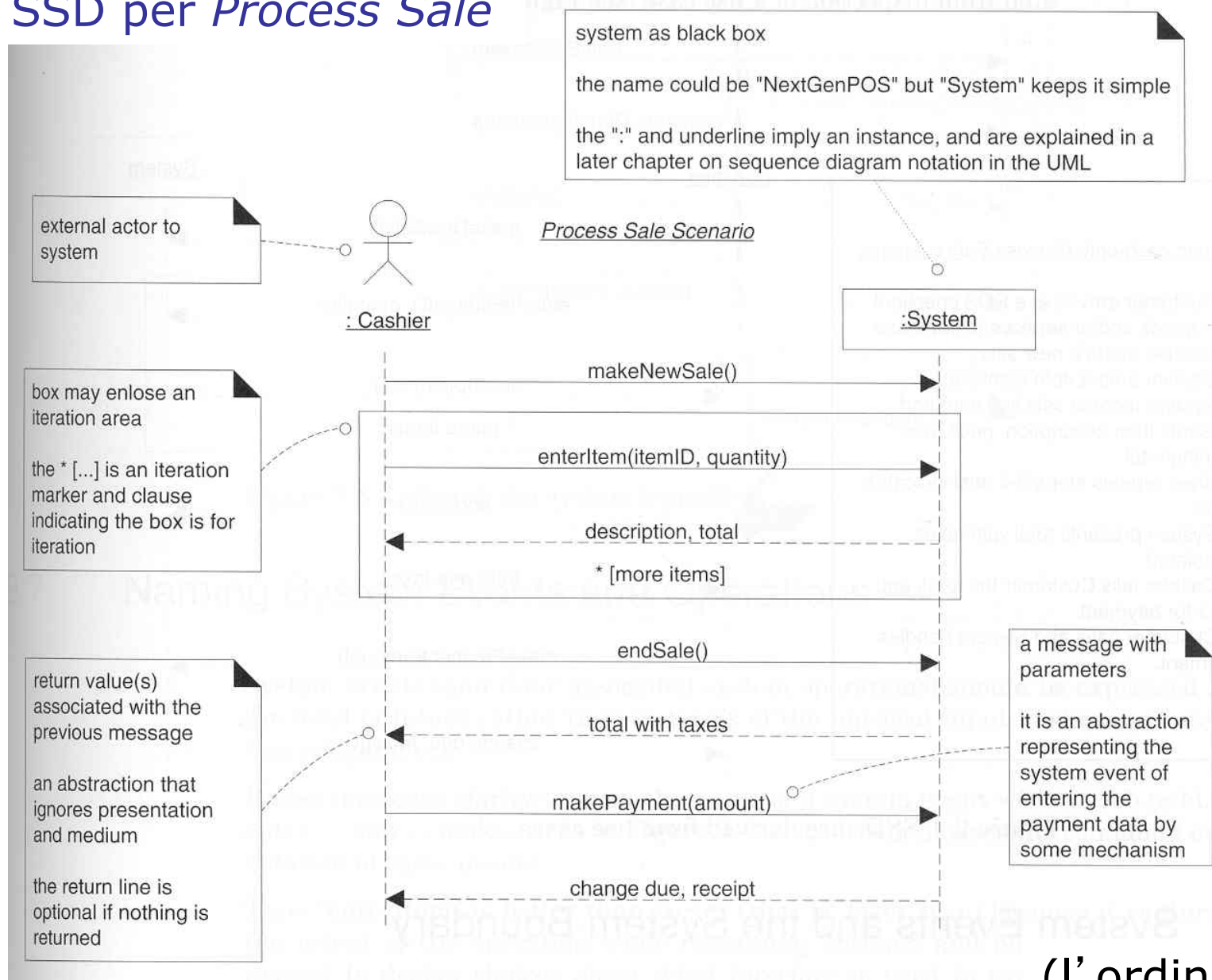
Esempio: diagramma di contesto



Comportamento del sistema

- Descrivere “cosa” fa un sistema senza spiegare “come”
- Use cases, *sequence diagrams*, contracts
- System Sequence Diagram (SSD): describe per uno specifico scenario gli eventi generati da attori esterni, il loro ordinamento, le possibili interazioni con altri sistemi

SSD per *Process Sale*

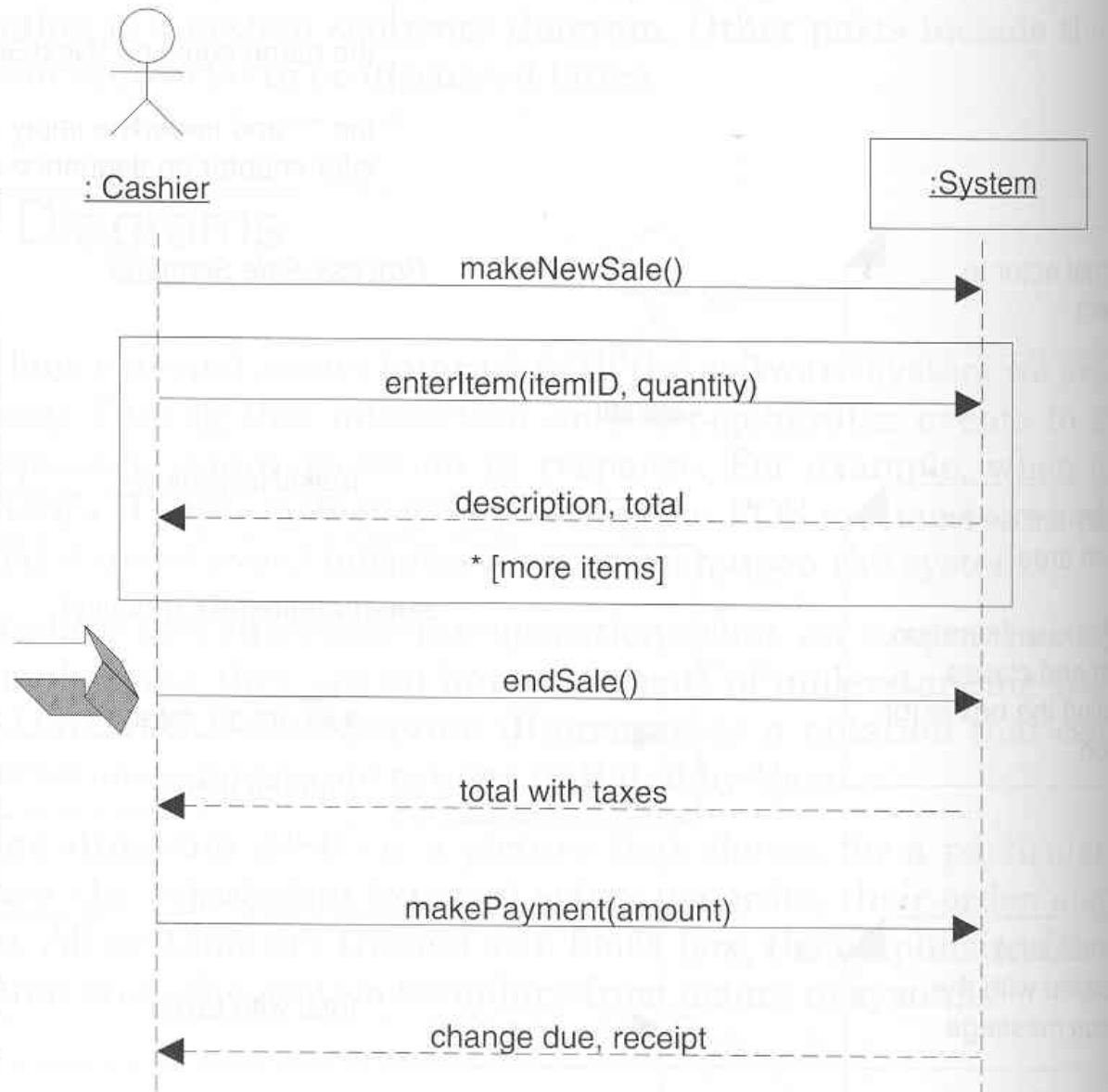
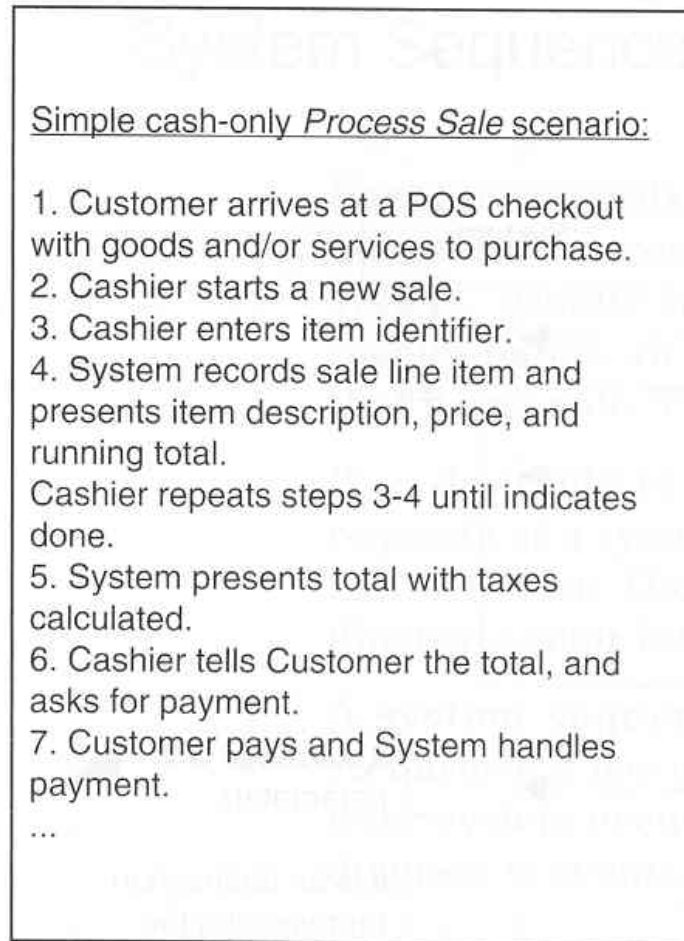


TIME

(l'ordine rispetta i passi dello scenario)

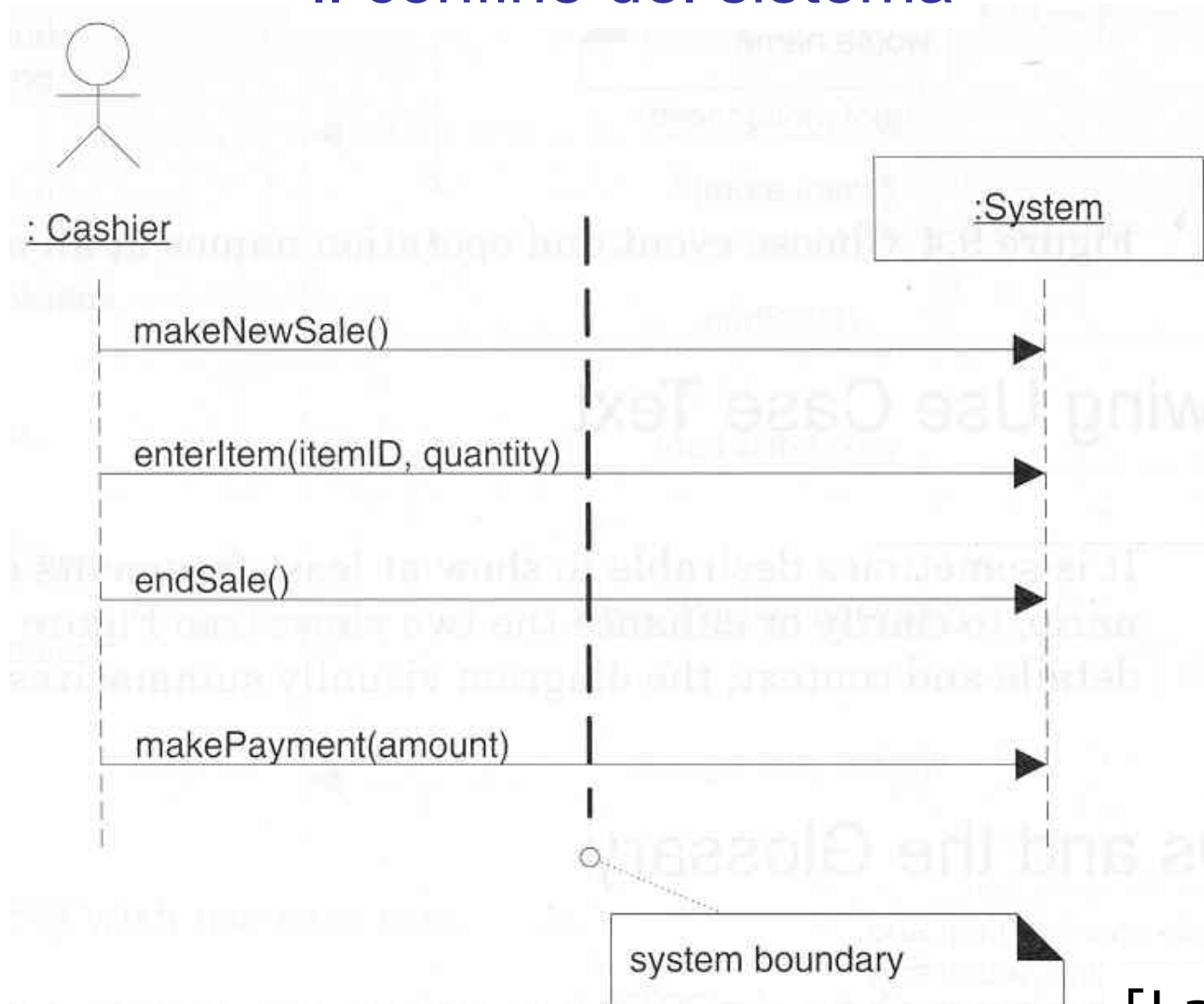
[Larman, 2002]

Da casi d'uso a SSD



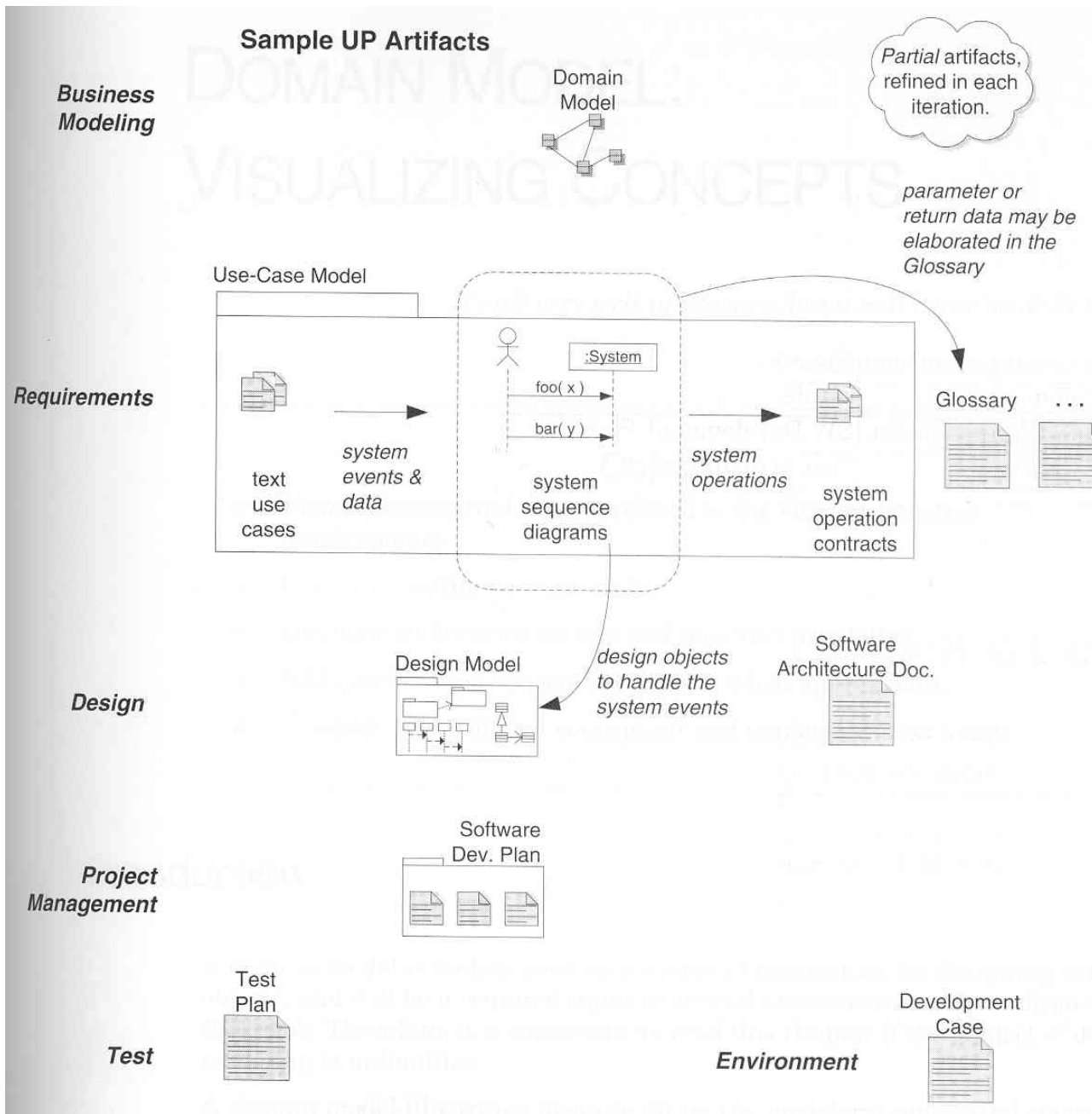
[Larman, 2002]

Il confine del sistema



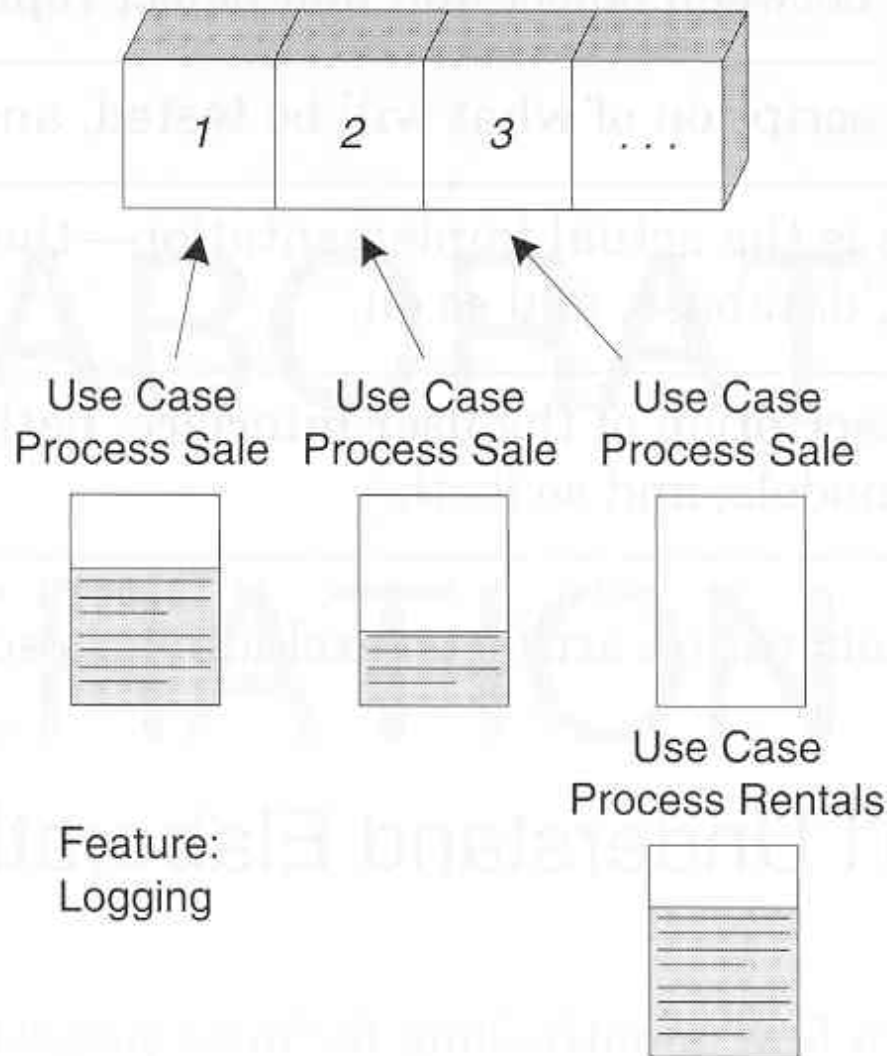
[Larman, 2002]

Uso del SSD



[Larman, 2002]

Iterare sui casi d'uso



A use case or feature is often too complex to complete in one short iteration.

Therefore, different parts or scenarios must be allocated to different iterations.

[Larman, 2002]

Analisi = Attività + Modelli

Attività di processo	Modello prodotto
1. Elicitazione dei requisiti d'utente e identificazione dei casi d'uso	Modello di dominio, Diagrammi e scenari dei casi d'uso, SSD
2. Estrazione delle classi candidate, identificazione degli attributi e dei metodi, definizione della gerarchia delle classi	Schede Classe-Responsabilità-Collaboratori (CRC)
3. Costruzione di un modello a oggetti e relazioni	Diagramma delle classi
4. Costruzione di un modello operativo degli oggetti	Diagramma delle interazioni

Riassumendo...

- Passi principali della progettazione (iterabili)
 1. Modellazione degli attori e dei requisiti funzionali
 - Determinare gli attori principali e quelli esterni
 - Determinare gli scenari e le varianti
 - Tecnica: **diagrammi Use Case**
 2. Modellazione del dominio
 - Determinare le entità principali
 - Assegnando le responsabilità
 - Tecnica: **schede CRC**
 3. Modellazione supplementare e requisiti non funzionali
 - Determinare l'ordine degli eventi
 - Tecnica: **diagrammi di contesto e SSD**

Sommario

- I requisiti vanno analizzati per scoprire scenari e casi d'uso rilevanti nel dominio del problema
- Gli oggetti vanno progettati a partire dalle responsabilità che ricoprono nel dominio: il metodo delle CRC permette di analizzare le responsabilità
- L'approccio di Larman parte dal modello dei requisiti, definisce il modello del dominio ed arriva a costruire il modello di design
- Grazie a UML la modellazione è un'attività fortemente grafica

Domande di autotest

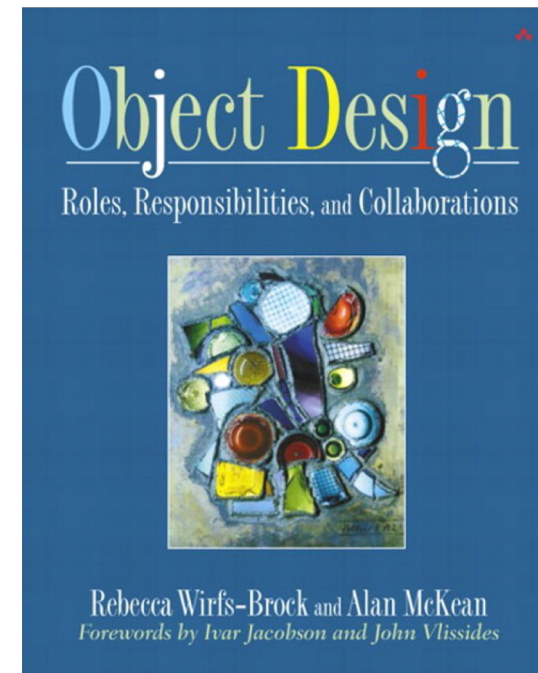
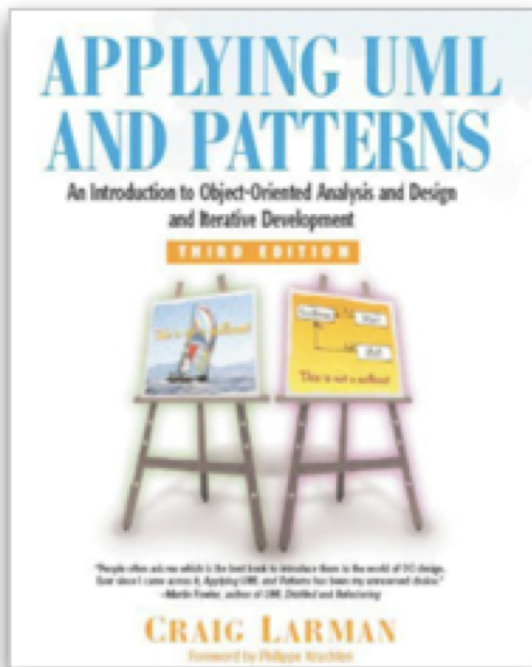
- Cosa caratterizza un linguaggio ad oggetti?
- Cos'è un diagramma del contesto?
- Cos'è il modello del dominio?
- Cos'è una responsabilità?
- Cos'è la progettazione guidata dalle responsabilità?
- Che relazione c'è tra schede CRC e diagramma di classi?
- A che serve un System Sequence Diagram?

Lettura raccomandata

- Blair, Watt & Cull, Responsibility-Driven Architecture, *IEEE Software*, 27:2, 2010

Riferimenti

- Larman, *Applicare UML e i Patterns*, Pearson 2005
- Wirfs-Brock and Mckean, *Object Design: Roles, Responsibilities and Collaborations*, AW 2002



Siti

- www.sei.cmu.edu/str/descriptions/oodesign.html
- www.objectsbydesign.com/
- hci.stanford.edu/bds/
- alistair.cockburn.us/Responsibility-based+modeling
- alistair.cockburn.us/Using+CRC+cards
- www.wirfs-brock.com/rebeccasblog.html

Strumenti

- `cruise.eecs.uottawa.ca/umpleonline`
- `www.excelsoftware.com/quickcrcmacosx`
- `www.math-cs.gordon.edu/courses/cps211/ATMExample/`

Pubblicazioni di ricerca

- SPLASH: Int. SIGPLAN conf. on Programming, Languages, Applications and Systems (era OOPSLA)
- European Conf. On OO Programming (ECOOP)
- Journal on Object Technology (open: www.jot.fm)

Domande?

