Object-Oriented Software Engineering

Using UML, Patterns, and Java

Chapter 2, Modeling with UML, Part 2

Unified Modeling Language

- un linguaggio (e notazione) universale, per la creazione di modelli software
- nel novembre ‘97 è diventato uno **standard** approvato dall’**OMG** (Object Management Group)
- numerosi i co-proponenti: Microsoft, IBM, Oracle, HP, Platinum, Sterling, Unysis …..

What is UML? Unified Modeling Language

- Convergence of different notations used in object-oriented methods, mainly
  - OMT (James Rumbaugh and collegues), OOSE (Ivar Jacobson), Booch (Grady Booch)
- They also developed the Rational Unified Process, which became the Unified Process in 1999

25 year at GE Research, where he developed OMT, joined (IBM) Rational in 1994, CASE tool OMTool

At Ericsson until 1994, developed use cases and the CASE tool Objectory, at IBM Rational since 1995, http://www.ivarjacobson.com

Developed the Booch method ("clouds"), ACM Fellow 1995, and IBM Fellow 2003 http://www.booch.com/

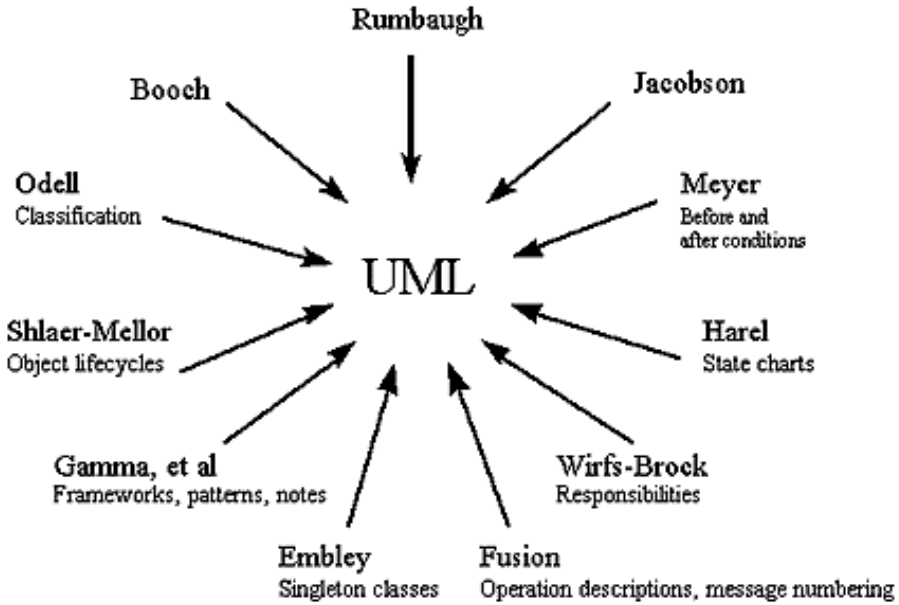
Grady Booch Reflects on UML 1.1

- https://www.youtube.com/watch?v=wy9pEIX7paQ

Grady Booch Reflects on UML 1.1 20th Anniversary

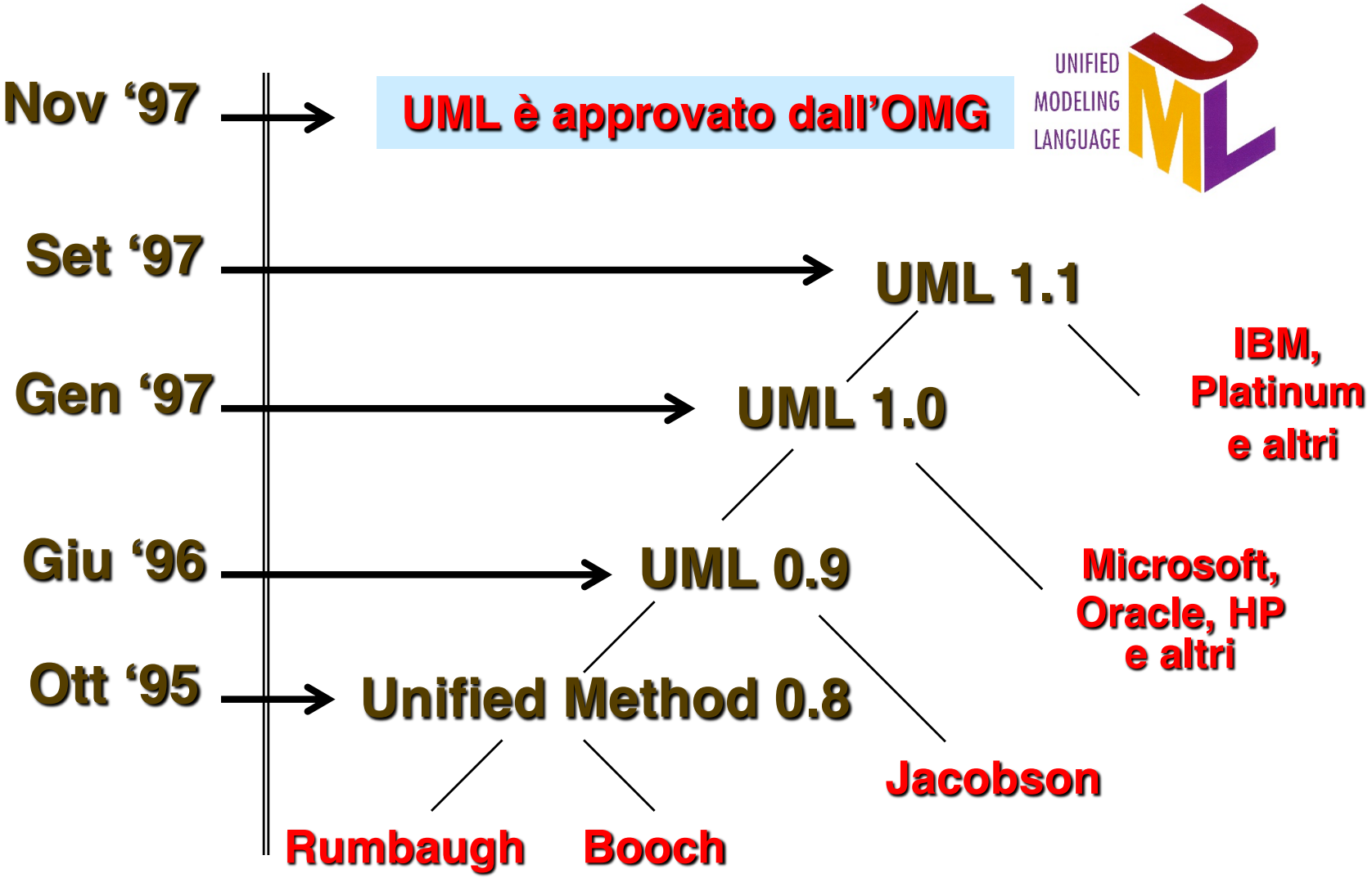
Unified Modeling Language

- è l'unificazione dei metodi:
  - Booch-93 di **Grady Booch**
  - OMT di **Jim Rumbaugh**
  - OOSE di **Ivar Jacobson**

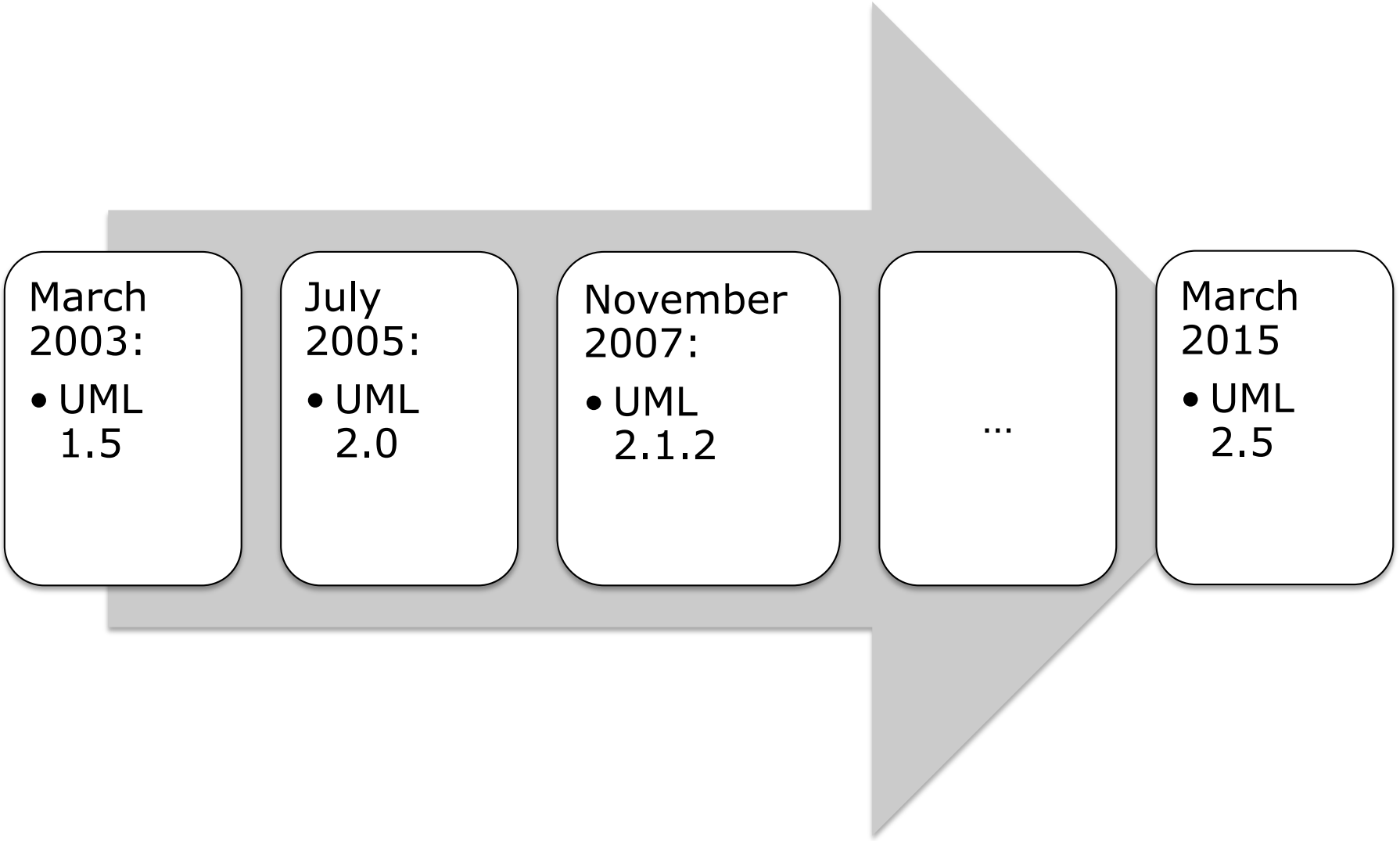


- ha accolto inoltre le idee di numerosi altri metodologi
- è tuttavia indipendente dai metodi, dalle tecnologie, dai produttori

Storia di UML…



Recent History of UML

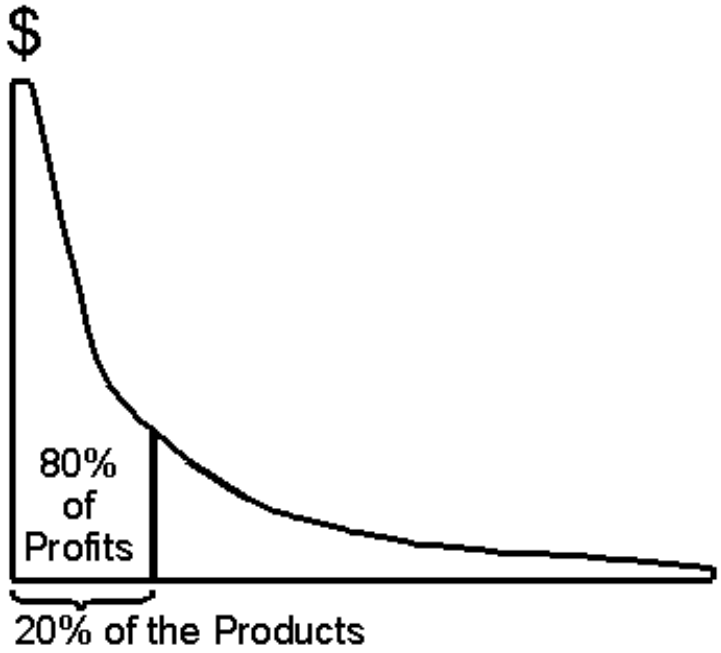


UML

- Nonproprietary standard for modeling systems
- Current Version: UML 2.5.1
  - Information at the OMG portal http://www.uml.org/
- Commercial tools:
  - Rational (IBM),Together (Borland), Visual Architect (Visual Paradigm), Enterprise Architect (Sparx Systems)
- Open Source tools http://www.sourceforge.net/
  - ArgoUML, StarUML, Umbrello (for KDE), PoseidonUML
- Example of research tools: Unicase, Sysiphus
  - Based on a unified project model for modeling, collaboration and project organization
  - http://unicase.org
  - http://sysiphus.in.tum.de/

UML: First Pass

- You can solve 80% of the modeling problems by using 20 % UML
- We teach you those 20%
- 80-20 rule: Pareto principle



Vilfredo Pareto, 1848-1923
Introduced the concept of Pareto
Efficiency,
Founder of the field of microeconomics.

UML - linguaggio universale

- **linguaggio** per specificare, costruire, visualizzare e documentare gli artefatti di un sistema

- **universale**: può rappresentare sistemi molto diversi, da quelli web ai legacy, dalle tradizionali applicazioni Cobol a quelle object oriented e a componenti

UML non è un metodo

- è un **linguaggio di modellazione**, non un metodo, né una metodologia

- definisce una notazione standard, basata su un meta-modello integrato degli “elementi” che compongono un sistema software

- non prescrive una sequenza di processo, cioè non dice “prima bisogna fare questa attività, poi quest’altra”

UML e Processo Software

"un unico processo universale buono per tutti gli stili dello sviluppo non sembra possibile e tanto meno desiderabile"

- in realtà UML assume un processo:
  - **basato sui Casi d'Uso (*use case driven*)**
  - **incentrato sull'architettura**
  - **iterativo e incrementale**
- i dettagli di questo processo di tipo generale vanno adattati alle peculiarità della cultura dello sviluppo o del dominio applicativo di ciascuna organizzazione

Obiettivi dell'UML

- Fornire all'utente un **linguaggio di specifica** espressivo, visuale e pronto all'uso
- Offrire meccanismi di **estensibilità** e **specializzazione** del linguaggio (tramite stereotipi)
- Essere *indipendente* dagli specifici linguaggi di programmazione e dai processi di sviluppo
- Incoraggiare la crescita dei tool OO commerciali
- Supportare concetti di sviluppo ad alto livello come frameworks, pattern ed i componenti
- Integrare i migliori approcci.

Cosa non è UML!

- **Non** è un linguaggio di *programmazione* visuale (è un linguaggio di **specifica** visuale)
- UML non è un modello per la definizione di interfacce
- UML *non è dipendente* dal paradigma di sviluppo nel quale può essere utilizzato

We use Models to describe Software Systems

- System model: functional model + object model + dynamic model

- Functional model: What are the functions of the system?
  - UML Notation: Use case diagrams
- Object model: What is the structure of the system?
  - UML Notation: Class diagrams
- Dynamic model: How does the system react to external events?
  - UML Notation: Sequence, State chart and Activity diagrams

Diagrams

- Use case diagrams
  - Describe the functional behavior of the system *as seen by the user*
- Class diagrams
  - Describe the *static structure* of the system: Objects, attributes, associations
- Sequence diagrams
  - Describe the *dynamic behavior* between *objects* of the system
- Statechart diagrams
  - Describe the *dynamic behavior* of an *individual object*
- Activity diagrams
  - Describe the *dynamic behavior* of a *system*, in particular the *workflow*.

Use case diagrams

Diagramma dei casi d'uso

Mostra:

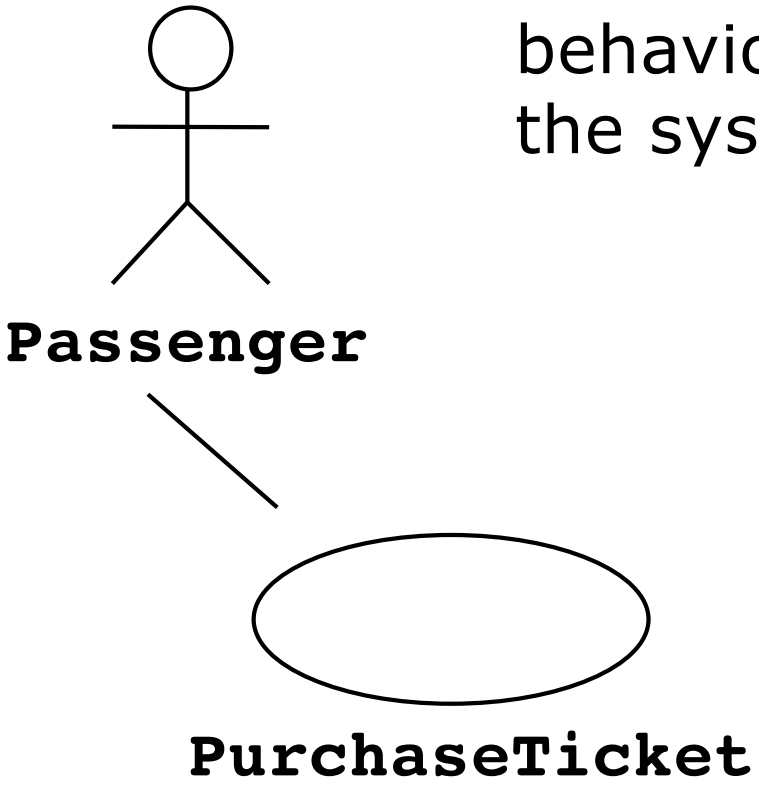
- le modalità di utilizzo del sistema (casi d'uso)
- gli utilizzatori e coloro che interagiscono con il sistema (attori)
- le relazioni tra attori e casi d'uso

Un caso d'uso

- rappresenta un possibile "modo" di utilizzo del sistema
- descrive l'interazione tra attori e sistema, non la "logica interna" della funzione

una funzionalità dal punto di vista di chi la utilizza

UML Use Case Diagrams



Used during requirements elicitation and analysis to represent external behavior ("visible from the outside of the system")

An ***Actor*** represents a role, that is, a type of user of the system

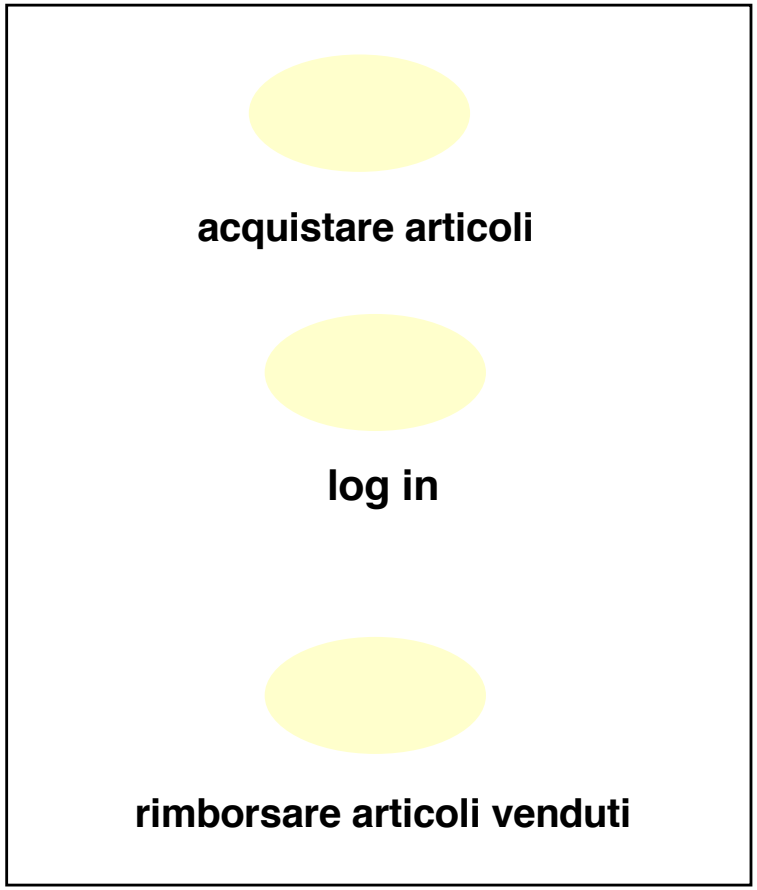
A ***use case*** represents a class of functionality provided by the system

Use case model:

The set of all use cases that completely describe the functionality of the system.

Diagramma dei casi d'uso

attore: un utilizzatore del sistema

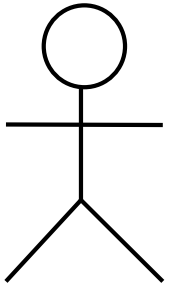


cliente

cassiere

caso d'uso: un "modo" di utilizzare il sistema

Actors



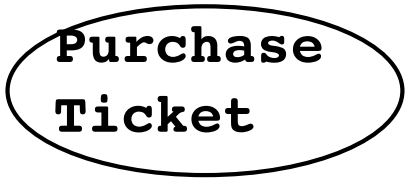
Passenger

- An actor is a model for an **external entity** which *interacts (communicates)* with the system:
  - User
  - External system (Another system)
  - Physical environment (e.g. Weather)
  - time
- An actor has a unique name and an optional description
- Examples:
  - **Passenger**: A person in the train
  - **GPS satellite**: An external system that provides the system with GPS coordinates.

Description

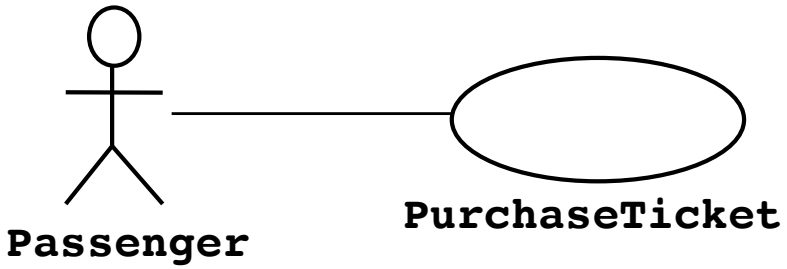
Name

Use Case



- A use case represents *a class of functionality* provided by the system
- Use cases can be described textually, with a focus on the event flow between actor and system
- The textual use case description consists of several parts:
  1. Unique name
  2. Participating actors
  3. Entry conditions
  4. Exit conditions
  5. Flow of events
  6. Special requirements.

Textual Use Case Description Example



1. Name: `Purchase ticket`

2. Participating actor:
`Passenger`

3. Entry condition:

- `Passenger` stands in front of ticket distributor
- `Passenger` has sufficient money to purchase ticket

4. Exit condition:

- `Passenger` has ticket

5. Flow of events:

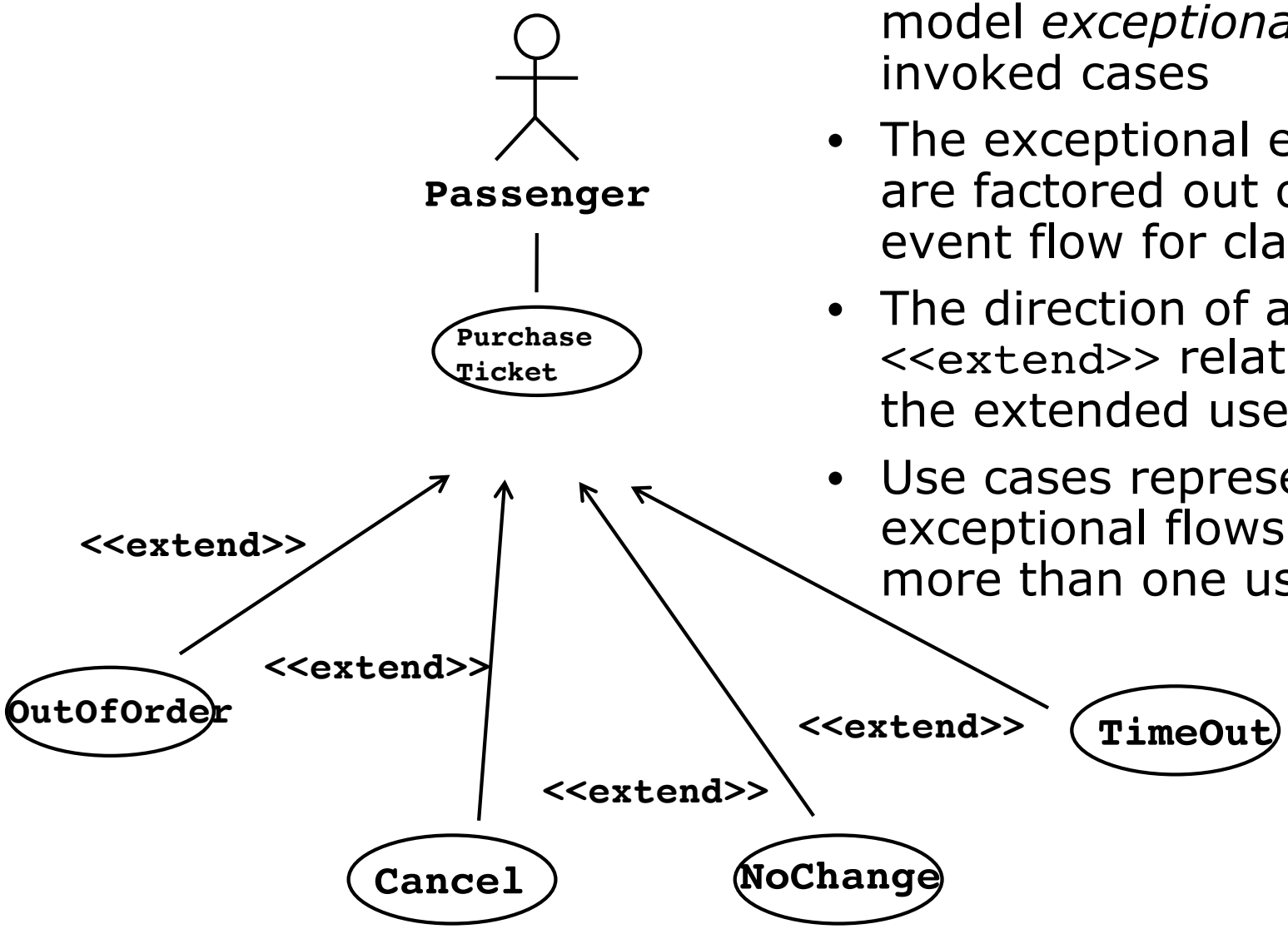
1. `Passenger` selects the number of zones to be traveled
2. Ticket Distributor displays the amount due
3. `Passenger` inserts money, at least the amount due
4. Ticket Distributor returns change
5. Ticket Distributor issues ticket

6. Special requirements:
None.

Uses Cases can be related

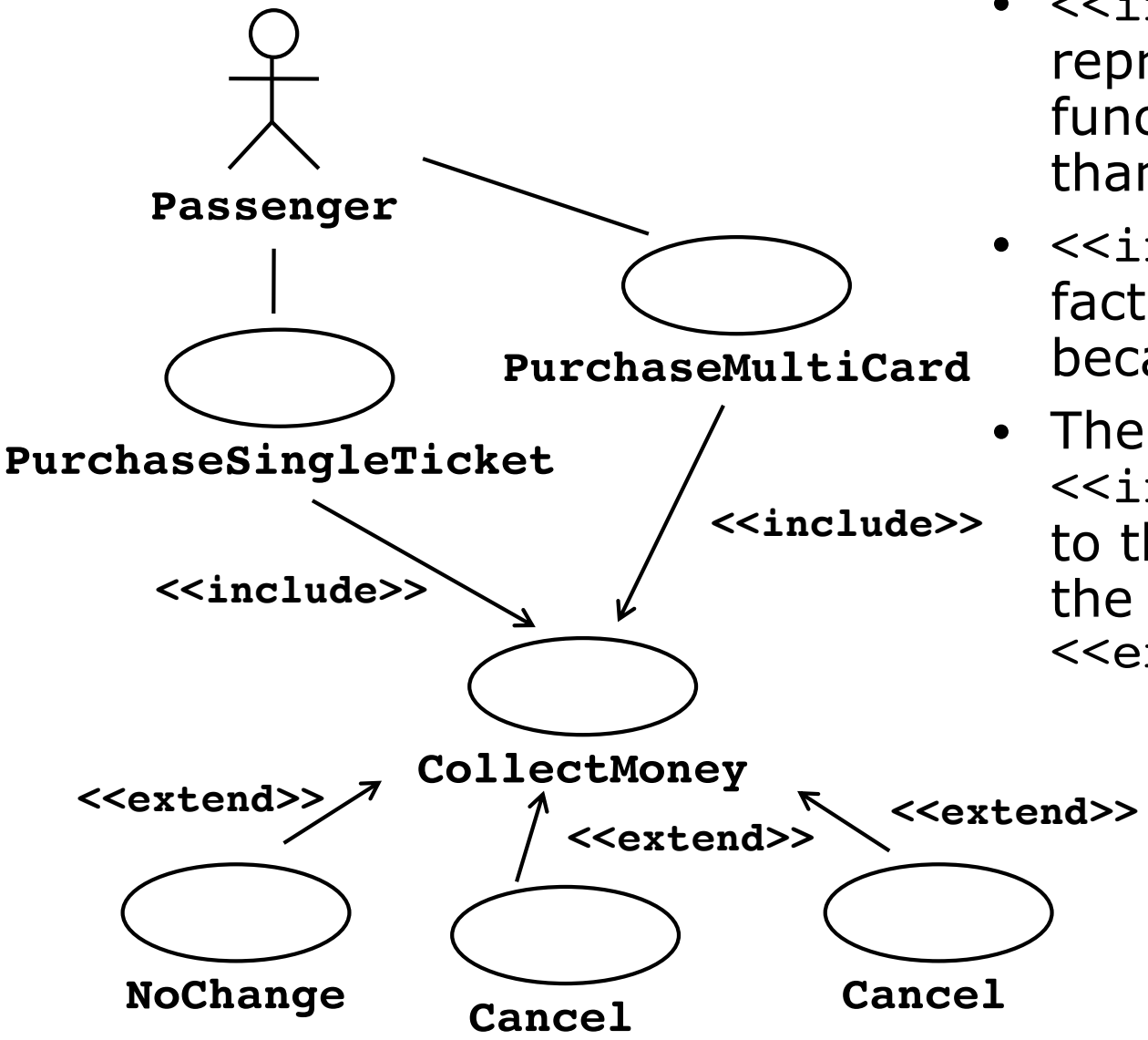
- Extend Relationship
  - To represent seldom invoked use cases or exceptional functionality
- Include Relationship
  - To represent functional behavior common to more than one use case.

The <<*extends*>> Relationship



- <<extend>> relationships model *exceptional* or *seldom* invoked cases
- The exceptional event flows are factored out of the main event flow for clarity
- The direction of an <<extend>> relationship is to the extended use case
- Use cases representing exceptional flows can extend more than one use case.

The *<<includes>>* Relationship



- <<include>> relationship represents common functionality needed in more than one use case
- <<include>> behavior is factored out for reuse, not because it is an exception
- The direction of a <<include>> relationship is to the using use case (unlike the direction of the <<extend>> relationship).

Activity Diagrams

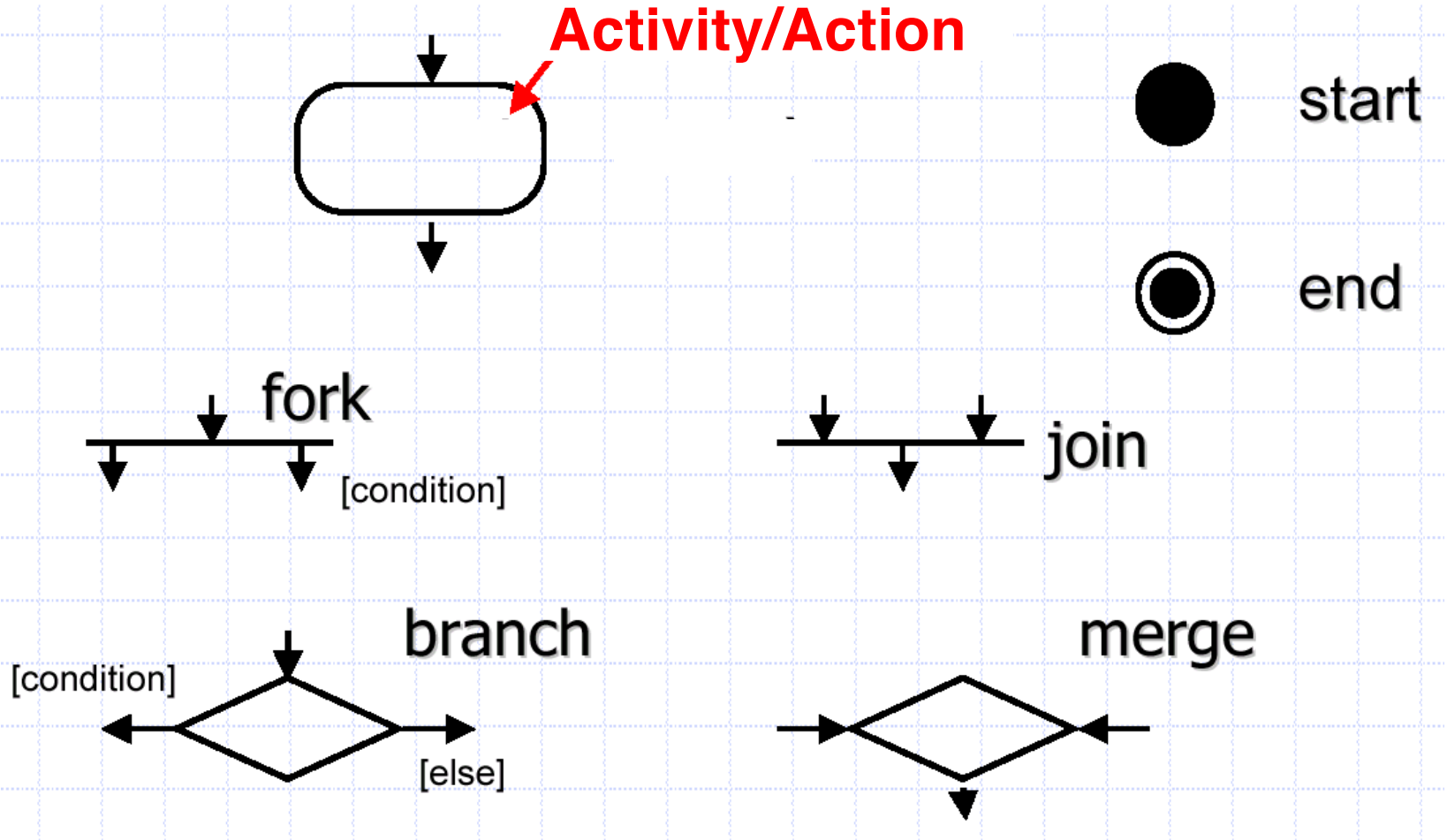
Activity Diagram (1)

- Forniscono la **sequenza di operazioni** che definiscono un'attività più complessa
- Permettono di rappresentare processi paralleli e la loro sincronizzazione
- Possono essere considerati State Diagram particolari
  - Ogni stato contiene (è) un' azione
- Un Activity Diagram può essere associato
  - A una classe
  - All' implementazione di un'operazione
  - Ad uno Use Case
  - …ma anche altro, ad un intero sistema, ad un contesto di business

Activity Diagram (2)

- [Derivano da event diagrams, reti di Petri]
- Servono a **rappresentare sistemi di workflow**, oppure la logica interna di un processo di qualunque livello
- Utili per modellare
  - comportamenti sequenziali
  - non determinismo
  - concorrenza
  - sistemi distribuiti
  - business workflow
  - operazioni
- Sono ammessi stereotipi per rappresentare le azioni

Elementi Grafici



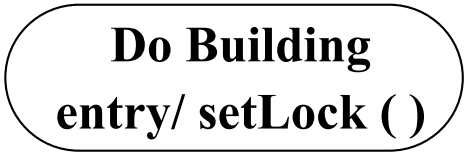
Le attività possono essere gerarchiche

Activity Diagram: Elementi

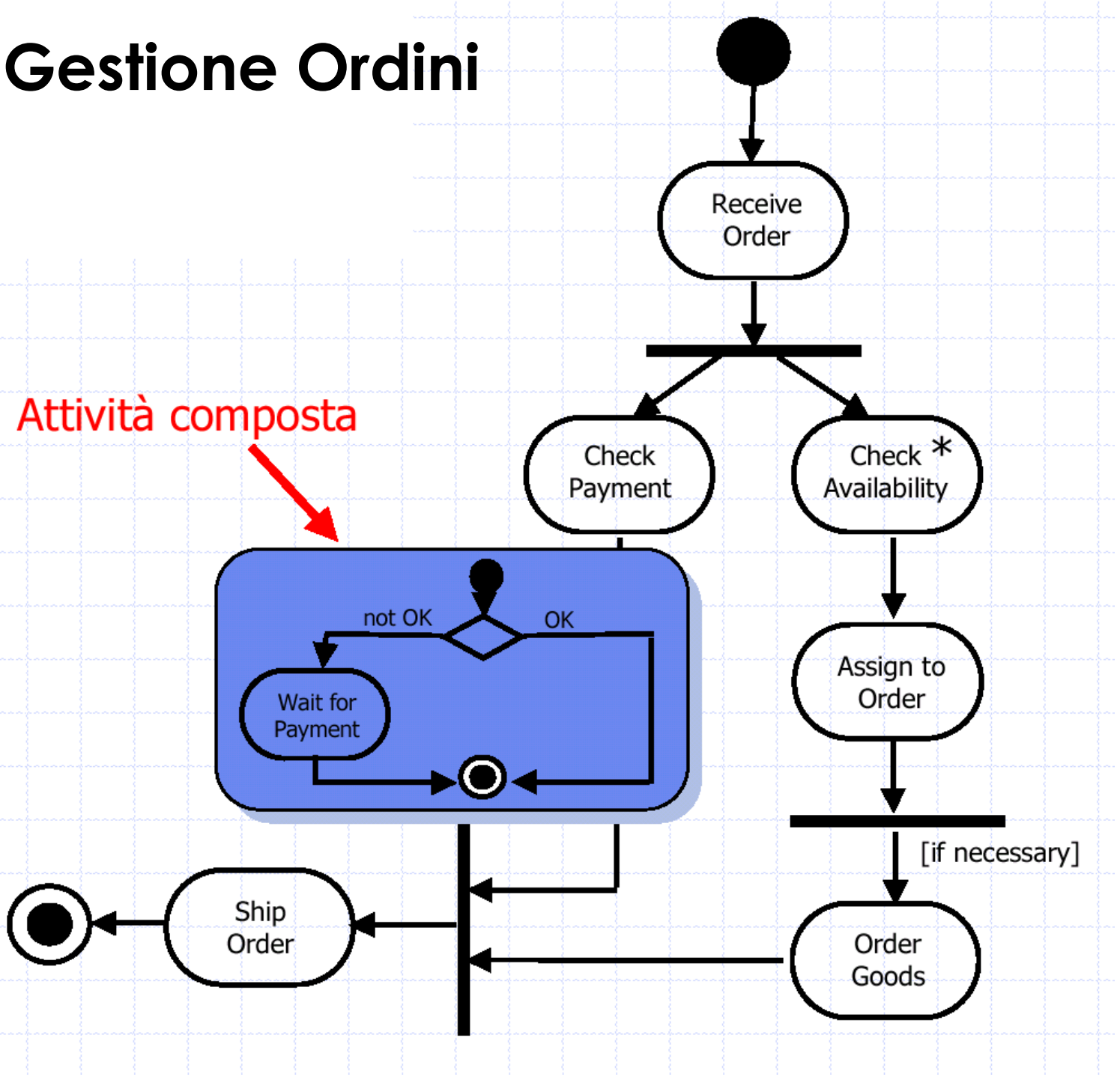
- *Activity*: una esecuzione *non atomica* entro uno state machine
  - Una activity è composta da action, elaborazioni atomiche comportanti un cambiamento di stato del sistema o il ritorno di un valore
- *Transition*: flusso di controllo tra due action successive
- *Guard expression*: espressione booleana (condition) che deve essere verificata per attivare una transition
- *Branch*: specifica percorsi alternativi in base a espressioni booleane; un branch ha una unica transition in ingresso e due o più transition in uscita
- *Synchronization bar*: usata per sincronizzare flussi concorrenti
  - *fork*: per splittare un flusso su più transition verso action state concorrenti
  - *join*: per unificare più transition da più action state concorrenti in una sola
    - il numero di fork e di join dovrebbero essere bilanciati

Action e Activity State

- *Activity state*: stati *non atomici* (cioè decomponibili ed interrompibili)
  - Un activity state può essere a sua volta rappresentato con un activity diagram
- *Action state*: azioni eseguibili atomiche (non possono essere decomposti né interrotti)
  - una action state può essere considerato come un caso particolare di activity state
- Activity e Action state hanno la stessa rappresentazione grafica
  - Un activity state può avere parti addizionali (es. entry ed exit action)



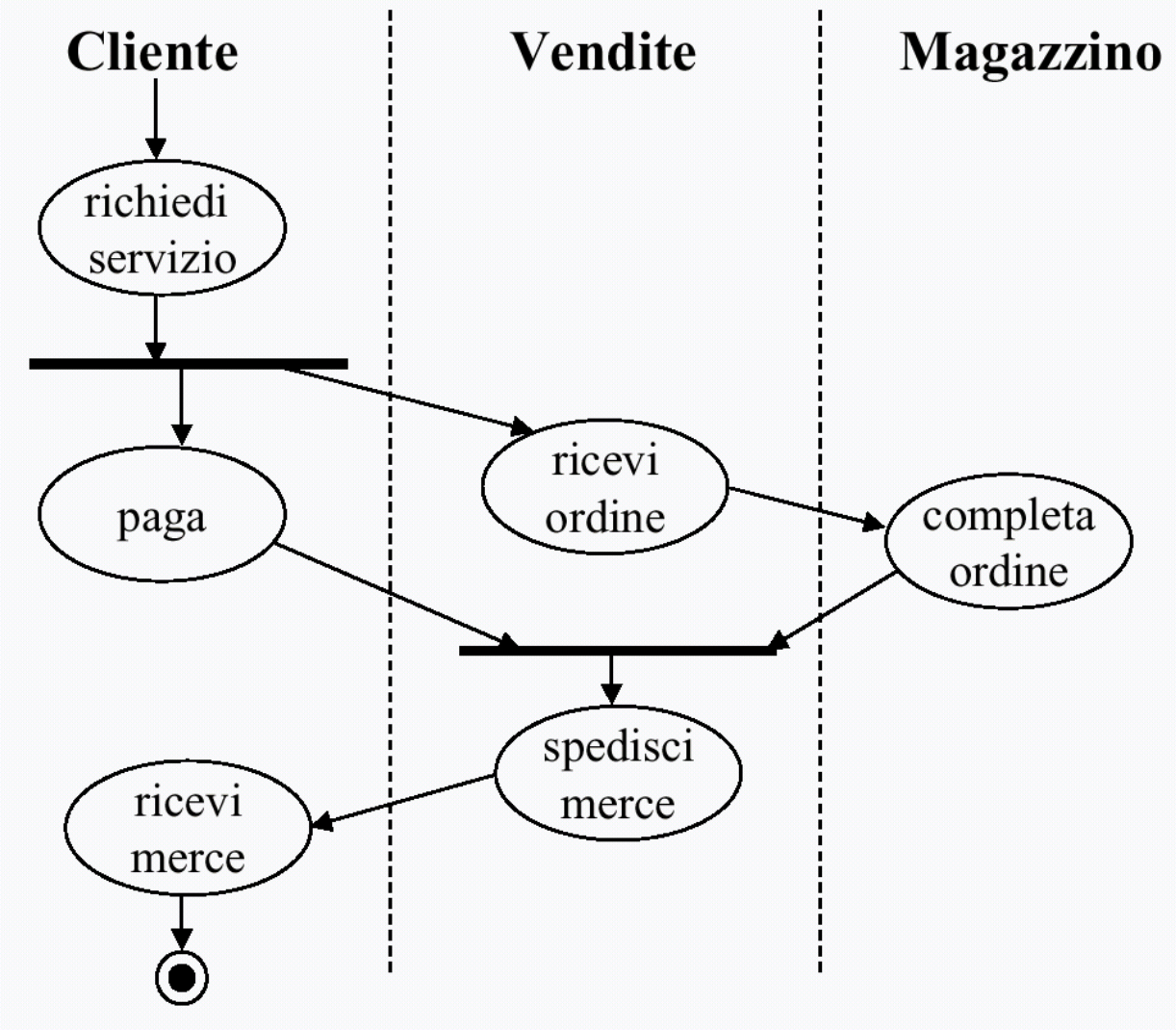
Esempio: Gestione Ordini



Swimlanes

- Costrutto grafico rappresentante un insieme partizionato di action/activity;
- Identificano le **responsabilità** relative alle diverse operazioni
- *In un Business Model identificano le unità organizzative*
- Per ogni *oggetto responsabile* di action/activity nel diagramma è definito una swimlane, identificata da un nome univoco nel diagramma
  - le action/activity state sono divise in gruppi e ciascun gruppo è assegnato alla swimlane dell'oggetto responsabile per esse
  - l'ordine con cui le swimlane si succedono non ha alcuna importanza
  - le transition possono attraversare swimlane per raggiungere uno state in una swimlane non adiacente a quello di start della transition

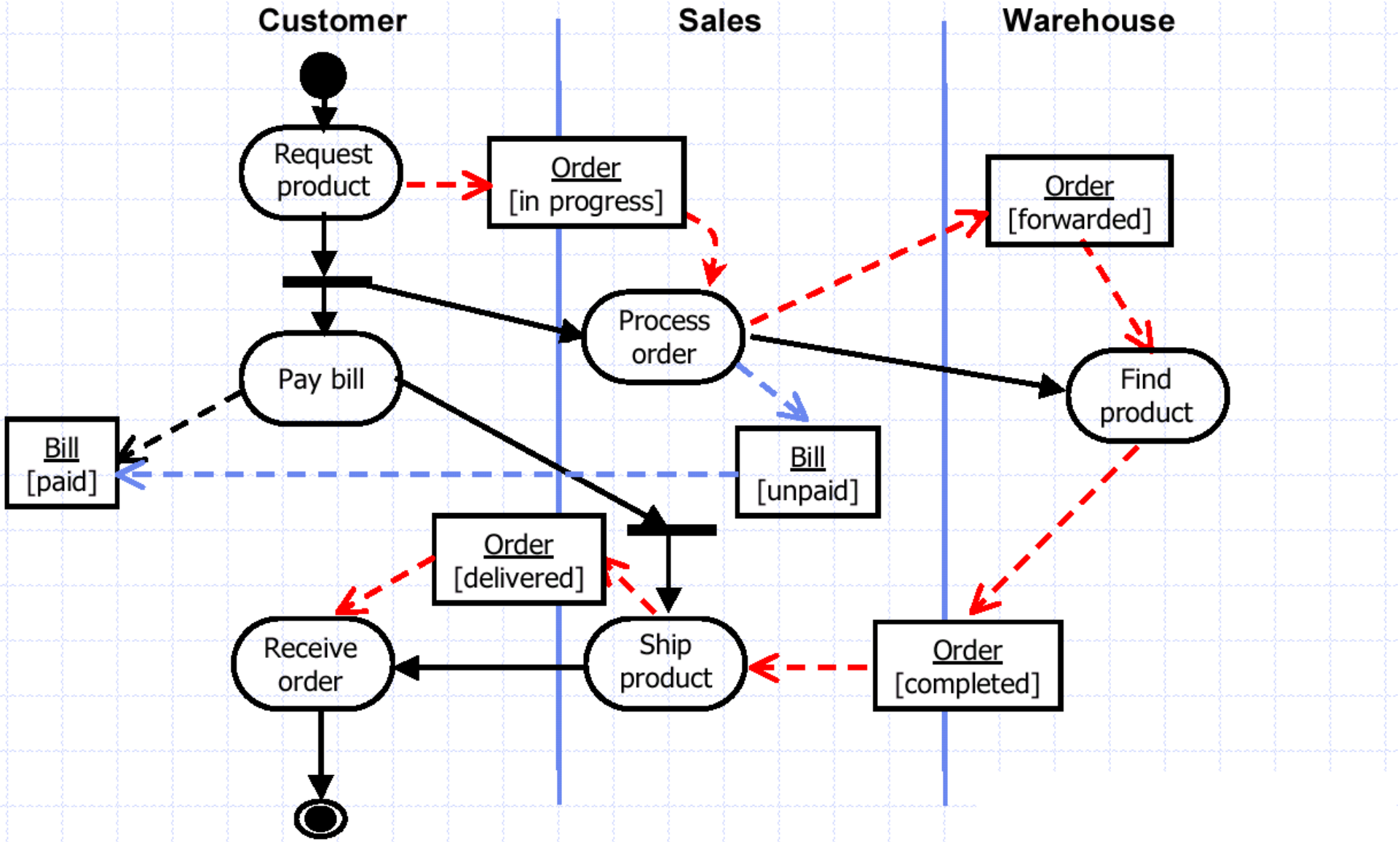
Esempio: Swimlanes



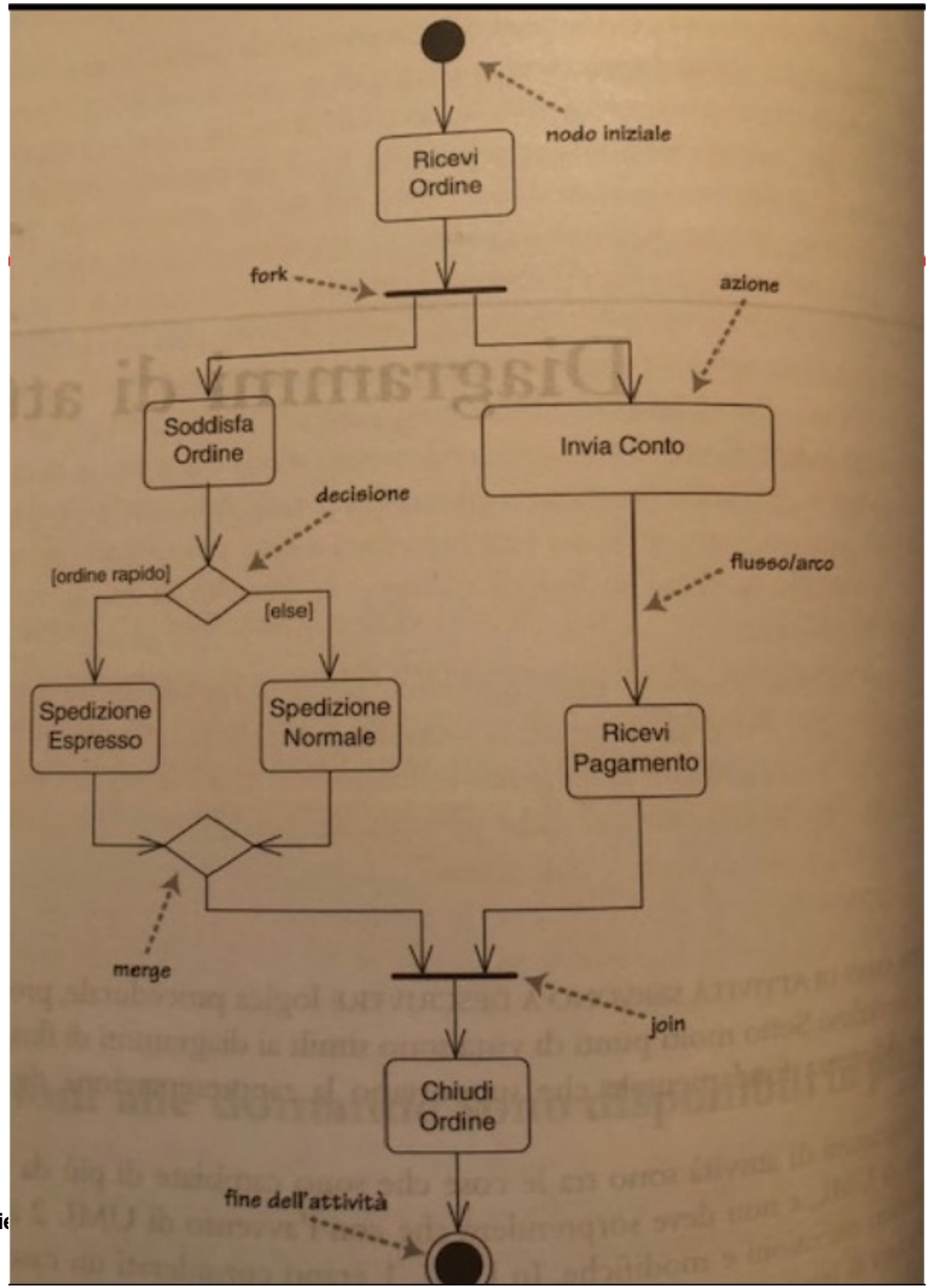
Attività e flussi di oggetti (Object flow)

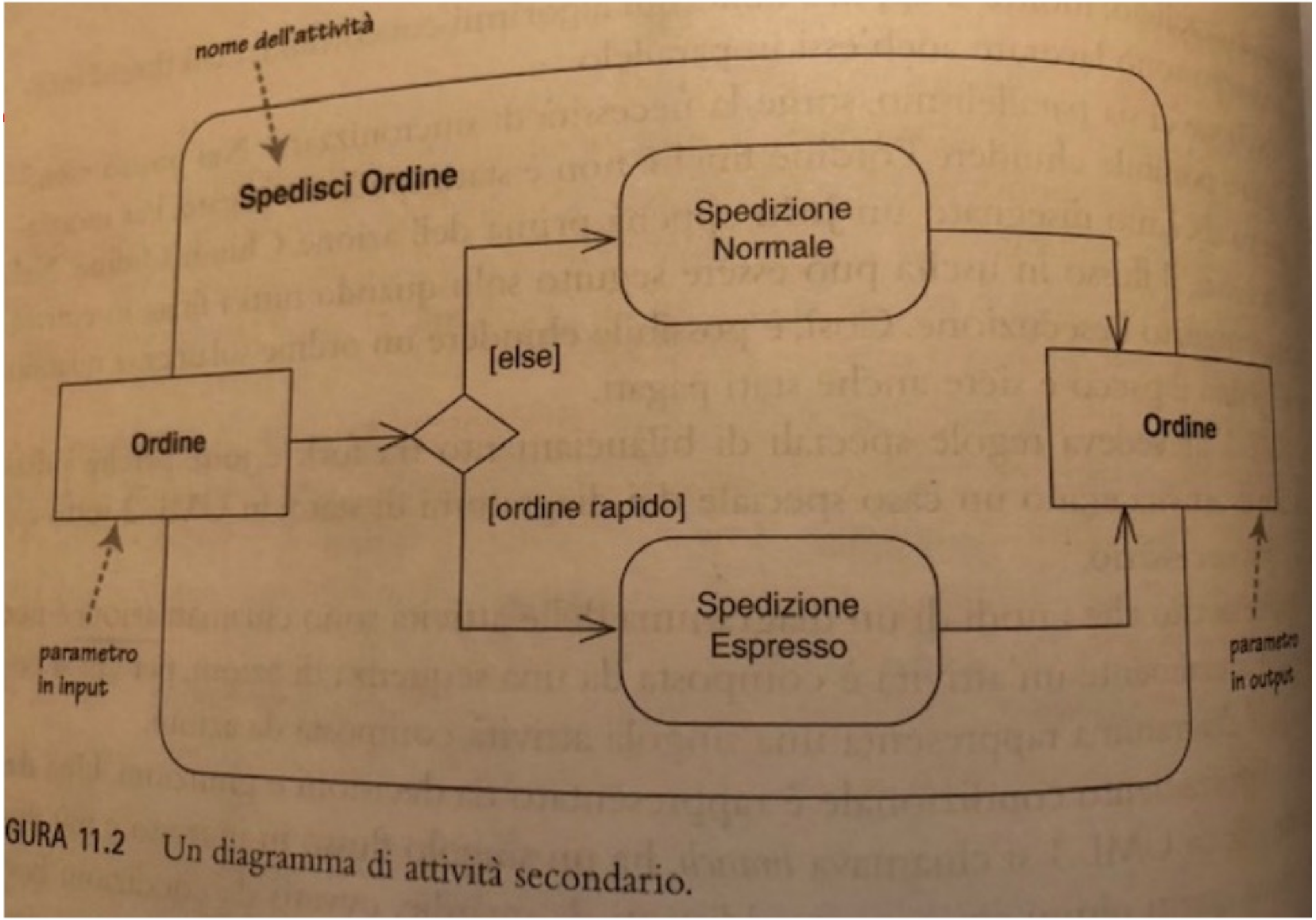
- Gli **object flow** sono
  - association tra action/activity state e object;
  - modellano l'utilizzo di object da parte di action/activity state e l'influenza di queste su essi
- Gli objects possono
  - essere l'output di una action: la action crea l'object, la freccia della relationship punta all'object
  - essere l'input di una action: questa usa lobject, la freccia della relationship punta all'action
  - essere manipolati da qualsiasi numero di action: l'output di una action può essere l'input di un'altra
  - essere presenti più volte nello stesso diagramma: ogni presenza indica un differente punto della vita dell'object
- può essere rappresentato lo *stato* di un object indicandolo tra [ ], al di sotto del nome dello object

Attività e flussi di oggetti



Un altro esempio: come fornire un approccio top-down alla descrizione





GURA 11.2 Un diagramma di attività secondario.

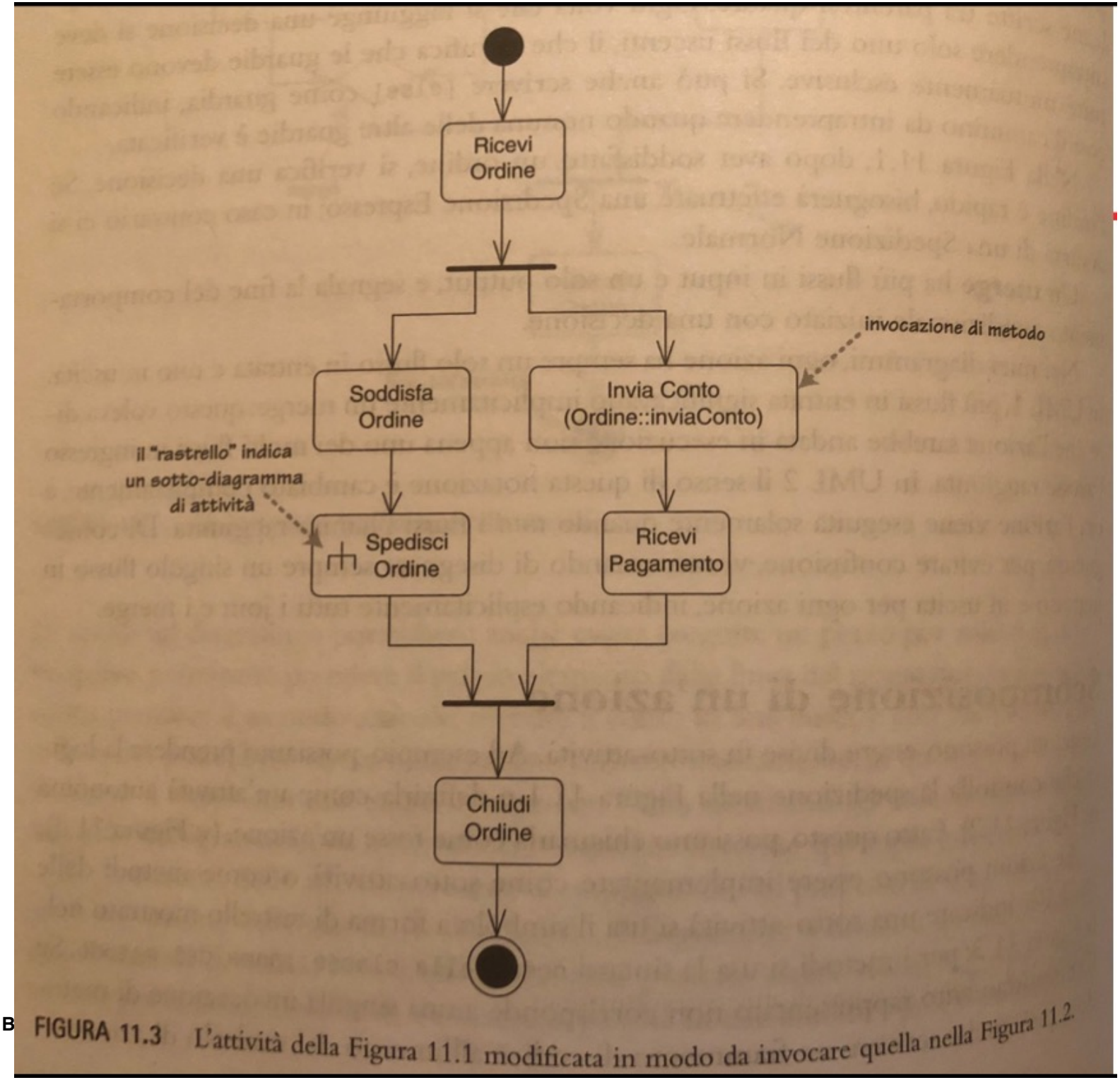


FIGURA 11.3 L'attività della Figura 11.1 modificata in modo da invocare quella nella Figura 11.2.

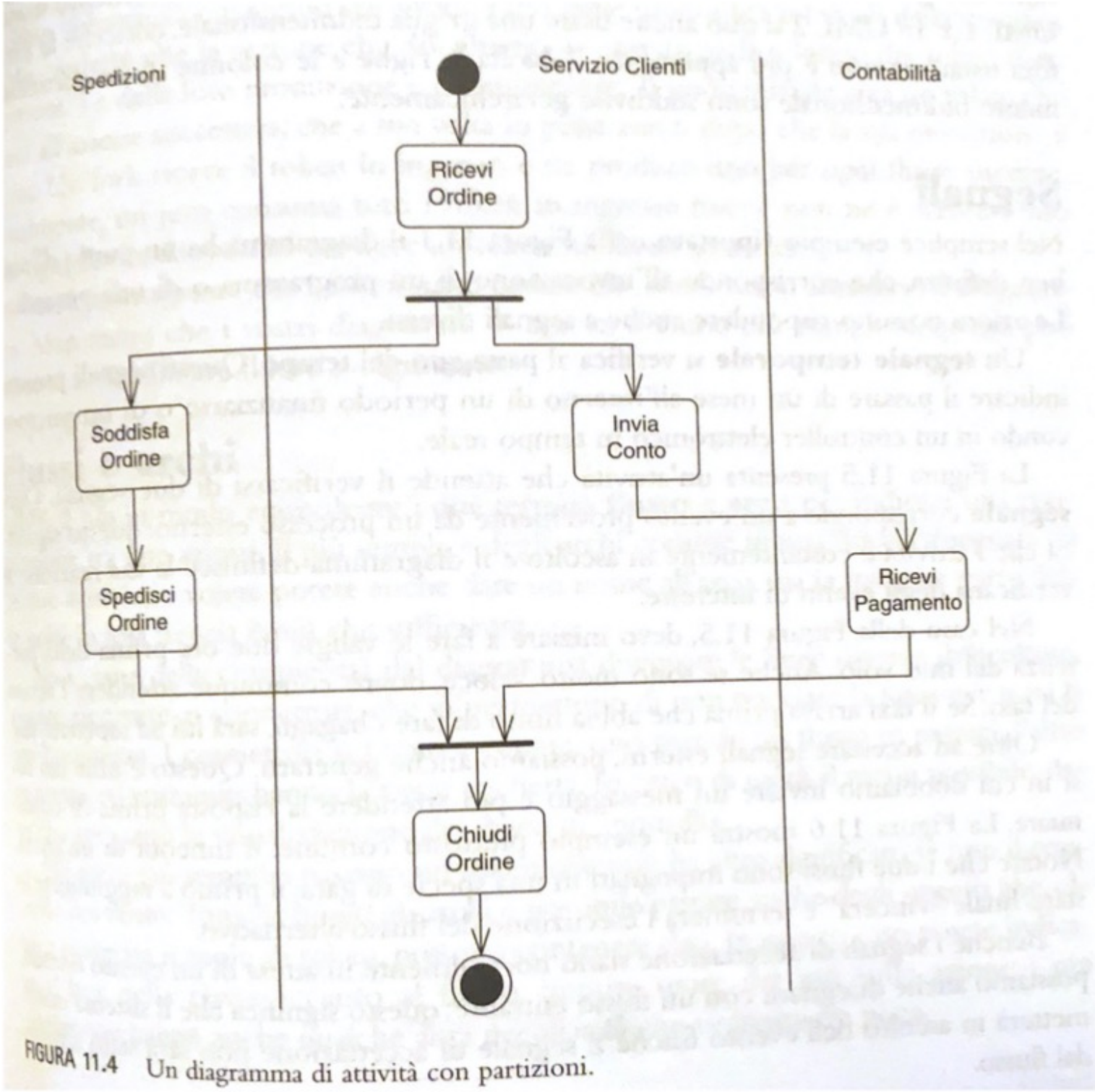


FIGURA 11.4 Un diagramma di attività con partizioni.

Come noi usiamo gli Activity Diagram

- Modellare il business workflow
- Es:
  - Come avviene la gestione degli ordini in un magazzino
  - Come avviene la registrazione degli esami, dall'inserimento della data di esame alla registrazione del voto
  - Il flusso di lavoro (workflow) necessario per la realizzazione di un tirocinio formativo (dalla individuazione della sede alla sua registrazione)
- Nel RAD utile per
  - modellare la situazione *as is* con i relativi problemi (sez. 2)
  - modellare la situazione *visionaria* (sez. 3)
  - capire come il sistema sw andrà a modificare il flusso di lavoro
- Rappresenta un complemento alla descrizione tramite scenari in maniera grafica, consentendo di evidenziare chi è responsabile di quali attività (swimlane), gli input e gli output delle varie attività (object), le dipendenze tra le attività (join..)

5
1

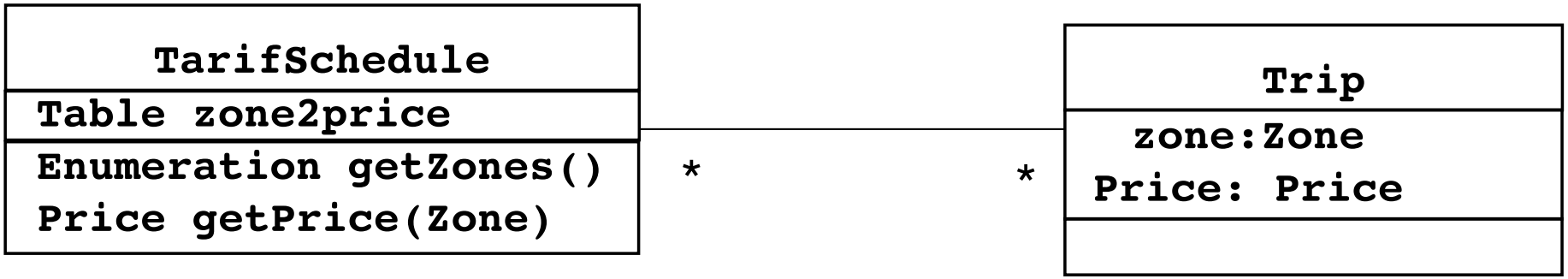
Class diagrams

Diagramma delle classi

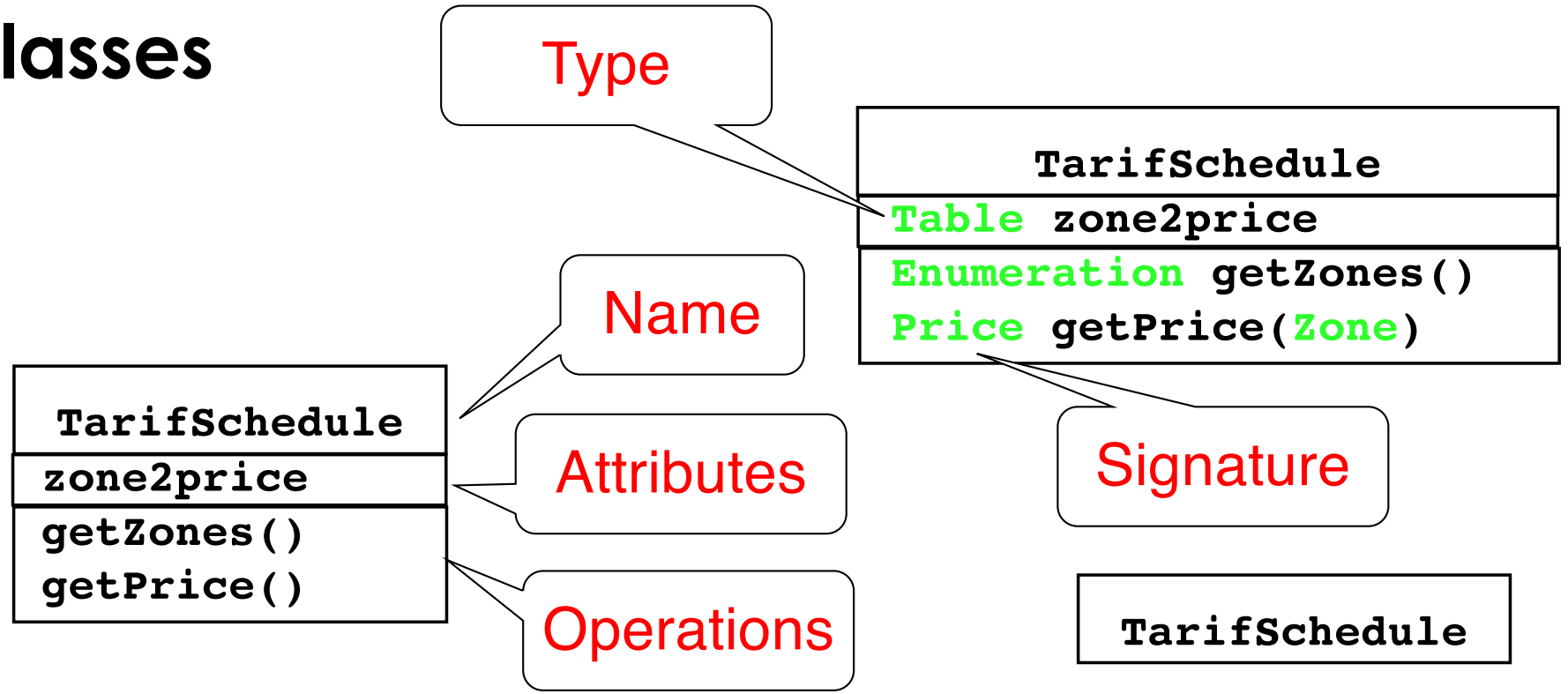
- è il caposaldo dell'object oriented
- rappresenta le classi di oggetti del sistema con i loro attributi e operazioni
- mostra le relazioni tra le classi (associazioni, aggregazioni e gerarchie di specializzazione/generalizzazione)
- può essere utilizzato a diversi livelli di dettaglio (in analisi e in disegno)

Class Diagrams

- Class diagrams represent the *structure* of the system
- Used
  - during *requirements analysis* to model **application domain** concepts
  - during *system design* to model **subsystems**
  - during *object design* to specify the **detailed behavior and attributes of classes**.



Classes

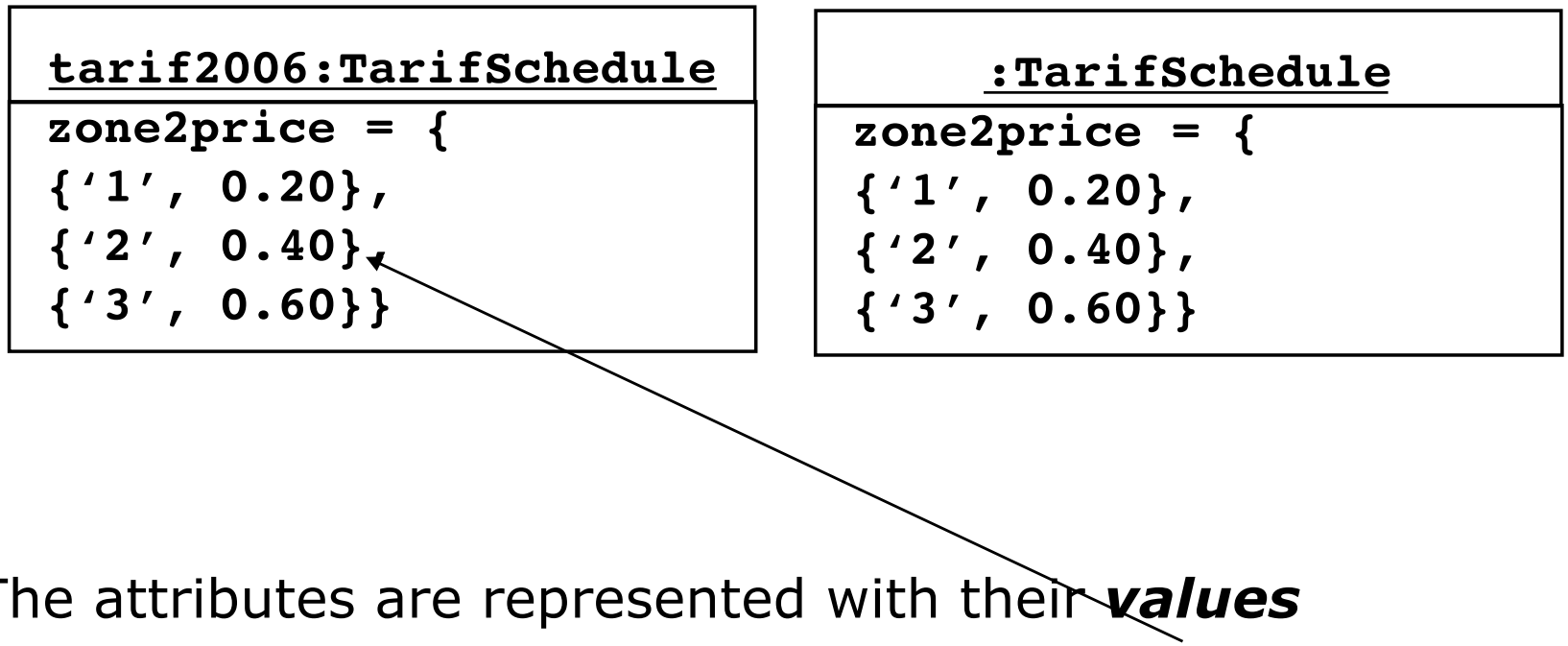


- A ***class*** represents a concept
- A class encapsulates state ***(attributes)*** and behavior ***(operations)***

  Each attribute has a ***type***
  Each operation has a ***signature***

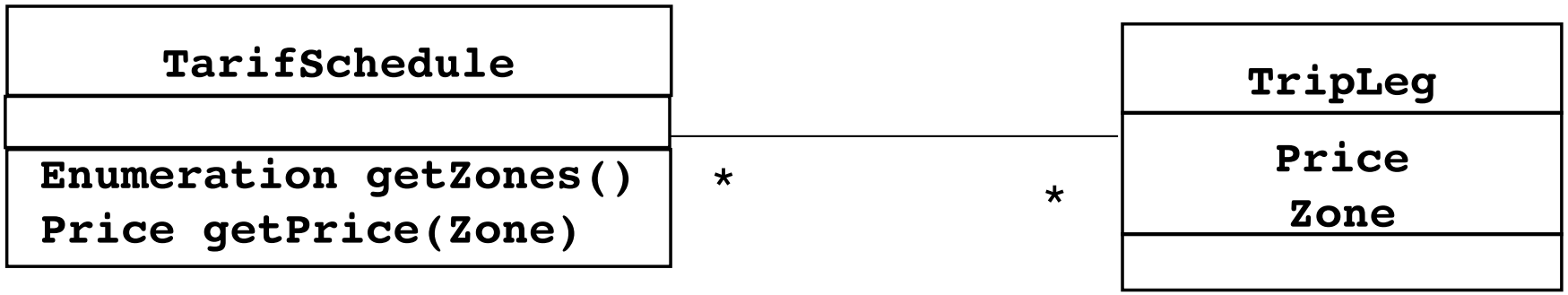
The class name is the only mandatory information

Instances



- The attributes are represented with their ***values***
- The name of an instance is underlined
- The name can contain only the class name of the instance (anonymous instance)

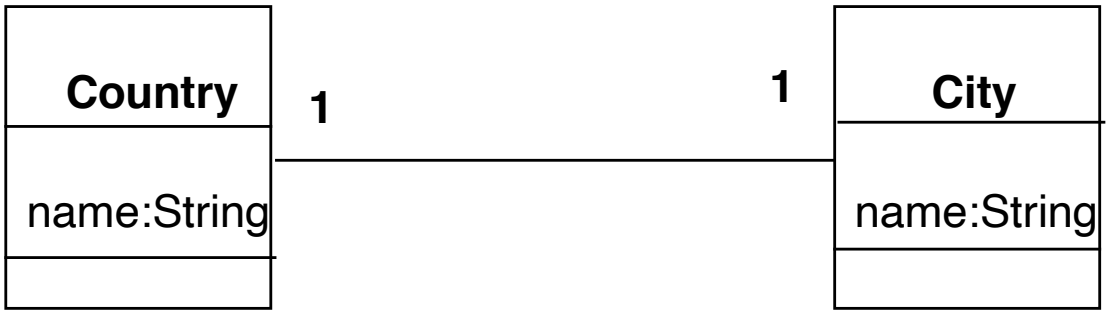
Associations



Associations denote *relationships* between classes

The multiplicity of an association end denotes how many objects the instance of a class can legitimately reference.

1-to-1 and 1-to-many Associations



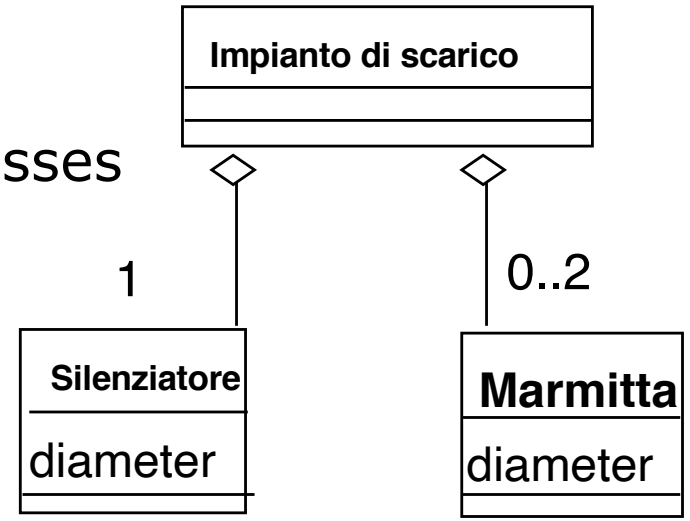
1-to-1 association

1-to-many association

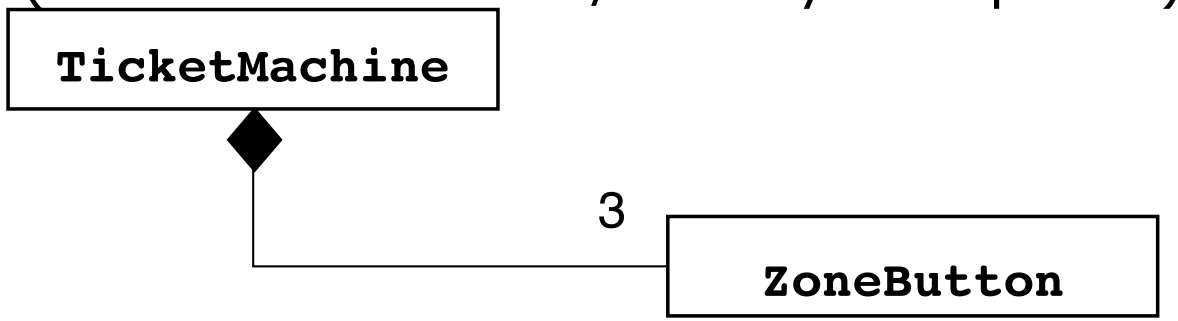
Many-to-Many Associations

Aggregation

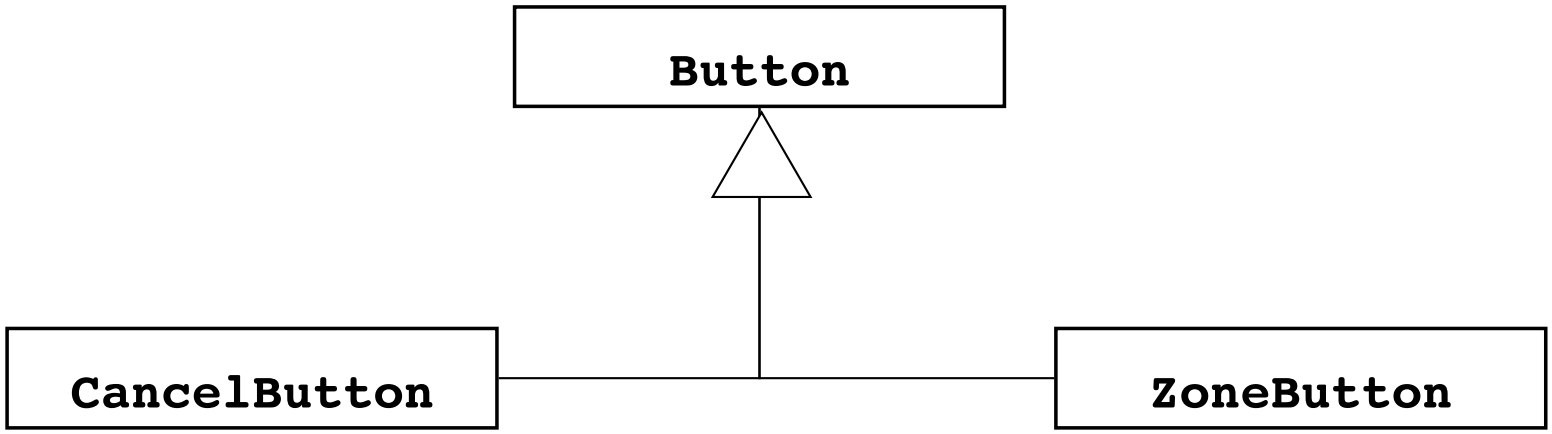
- An *aggregation* is a special case of association denoting a "consists-of" hierarchy
- The *aggregate* is the parent class, the components are the children classes



A solid diamond denotes *composition*: A strong form of aggregation where the *life time of the component instances* is controlled by the aggregate. That is, *the parts don't exist on their won* ("the whole controls/destroys the parts")



Inheritance



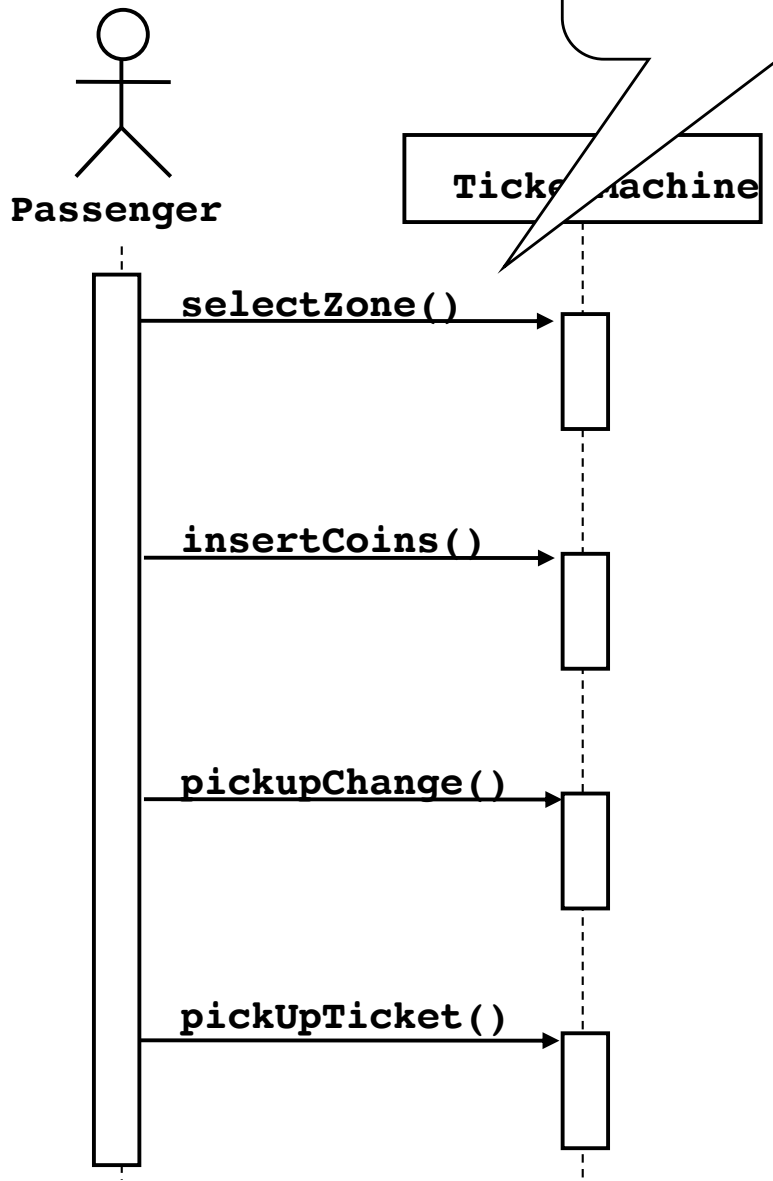
- *Inheritance* is another special case of an association denoting a "kind-of" hierarchy
- Inheritance simplifies the analysis model by introducing a taxonomy
- The **children classes** inherit the attributes and operations of the **parent class.**

Sequence diagrams

Diagramma di sequenza

- è utilizzato per definire la specifica sequenza di eventi di un caso d'uso (in analisi e poi ad un maggior livello di dettaglio nel design)
- è uno dei principali input per l'implementazione dello scenario
- mostra gli oggetti coinvolti specificando la sequenza temporale dei messaggi che gli oggetti si scambiano
- è un diagramma di interazione: evidenzia come un caso d'uso è realizzato tramite la collaborazione di un insieme di oggetti

Sequence Dia



Focus on Controlflow

Messages -> Operations on participating Object

- Used during analysis
  - To refine use case descriptions
  - to find additional objects ("participating objects")
- Used during system design
  - to refine subsystem interfaces
- ***Instances*** are represented by rectangles. ***Actors*** by sticky figures
- ***Lifelines*** are represented by dashed lines
- ***Messages*** are represented by arrows
- ***Activations*** are represented by narrow rectangles.

Sequence Diagram Properties

- UML sequence diagram represent *behavior in terms of interactions*
- Useful to identify or find missing objects
- Time consuming to build, but worth the investment
- Complement the class diagrams (which represent structure).

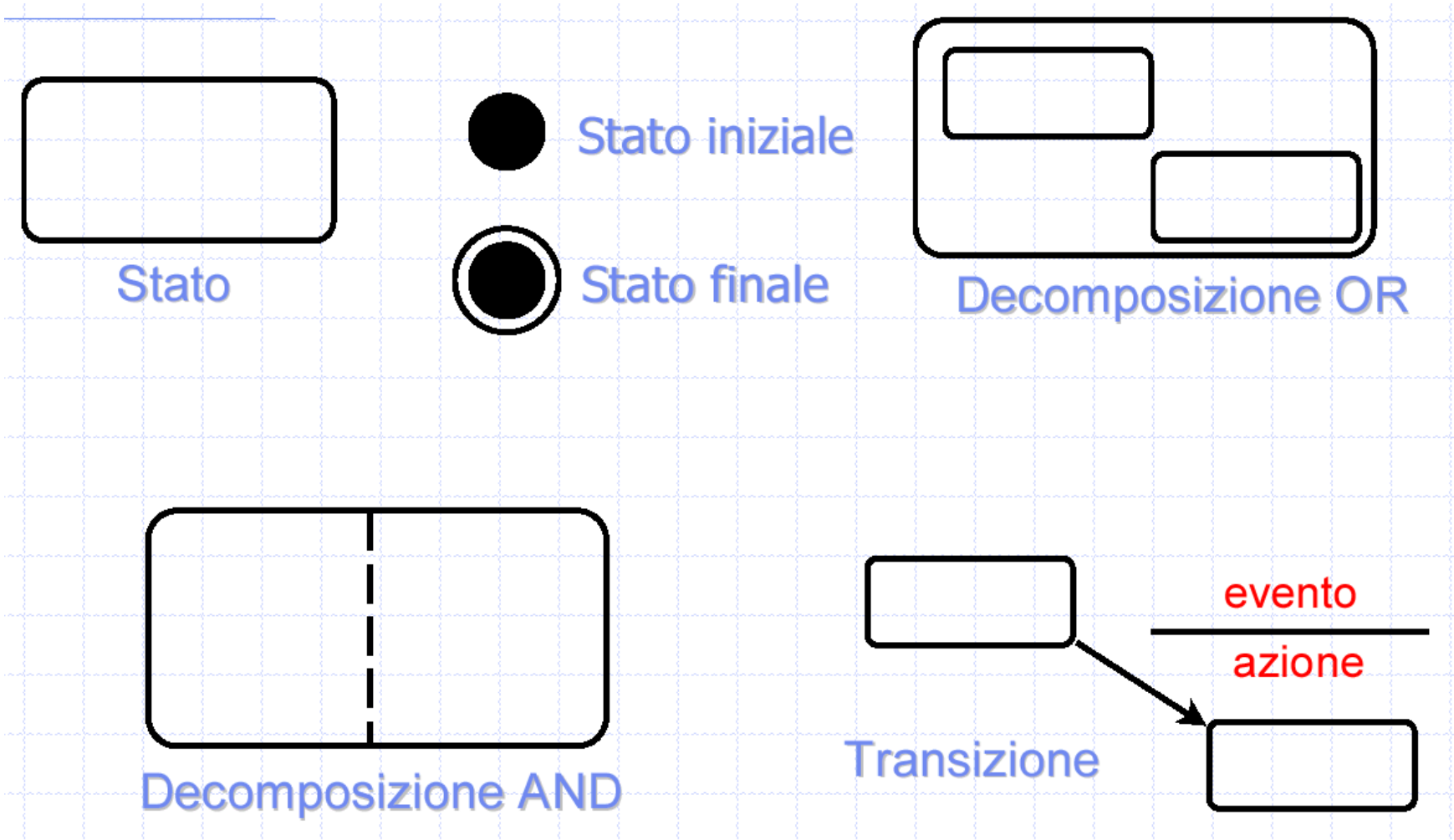
Class diagram and sequence diagram

- Use case diagram, Class diagram, and Sequence diagram
  - **Altri dettagli nelle prossime lezioni!!!**

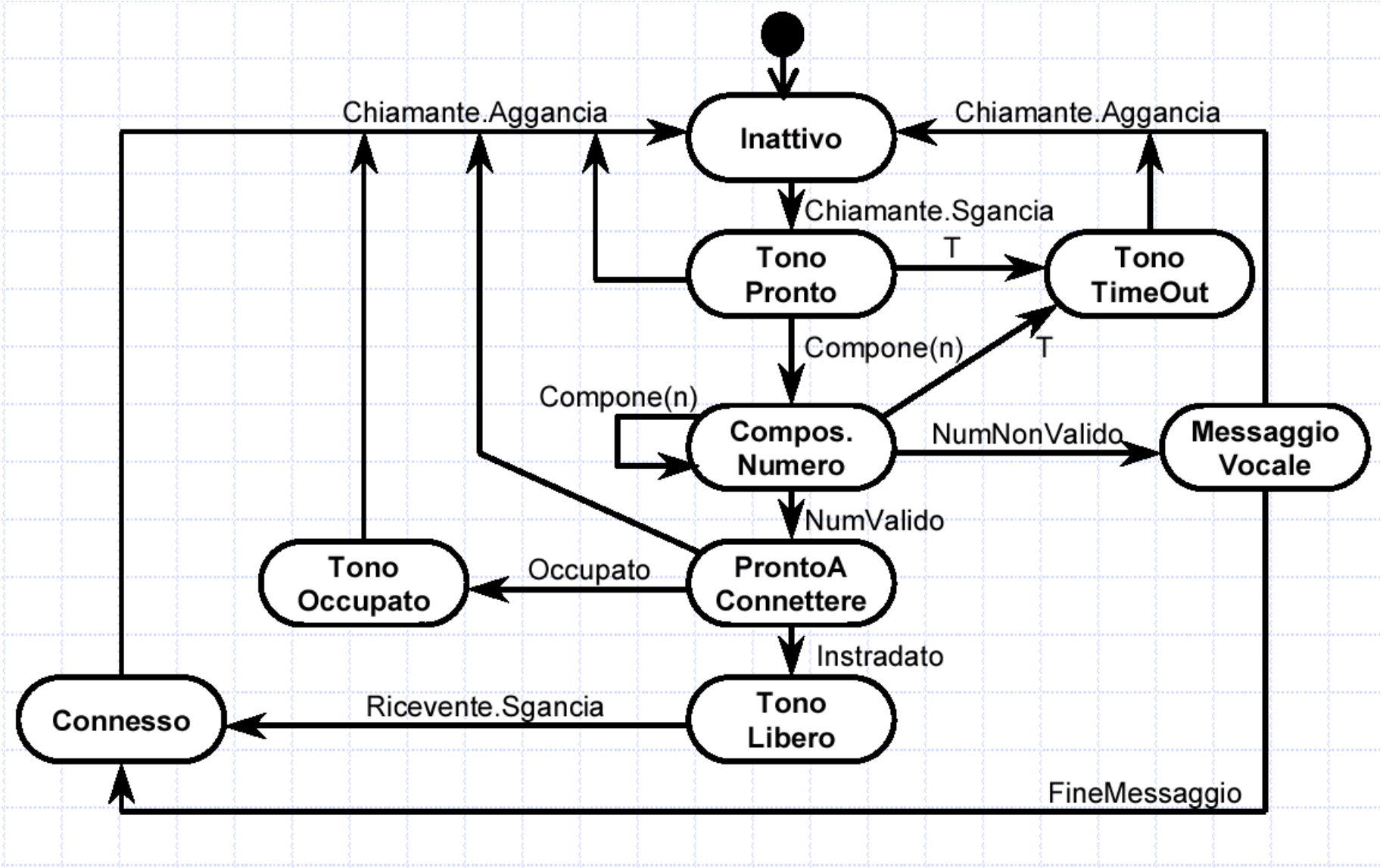
State (o Statechart) Diagrams

- Specifica il ciclo di vita di un oggetto
- Rappresentano il comportamento dei singoli oggetti in termini di
  - Eventi a cui gli oggetti (la classe) sono sensibili
  - Azioni prodotte
  - Transizioni di stato
    - Identificazione degli stati interni degli oggetti
- Possibilità di descrivere evoluzioni parallele
- Sintassi mutuata da StateChart (D. Harel)

Elementi grafici



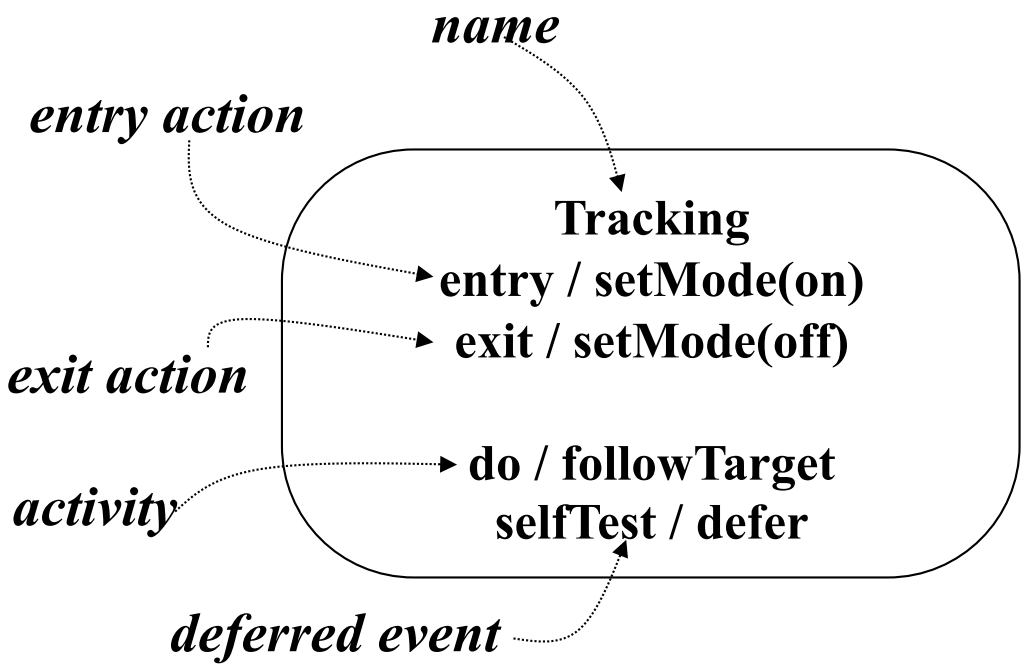
Esempio: telefonata



Stato

- Situazione in cui l'oggetto soddisfa qualche condizione, esegue qualche attività o aspetta qualche condizione
- Attributi (opzionali):
  - ***Nome***: una stringa; uno stato può essere anonimo
  - ***Entry / Exit actions***: eseguite all'ingresso / uscita dallo stato (non interrompibili, durata istantanea)
  - ***Transizioni interne***: non causano un cambiamento di stato
  - ***Attività*** dello stato (interrompibile, durata significativa)
  - ***Eventi differiti***: non sono gestiti in quello stato ma posposti ed accodati per essere gestiti da altri oggetti in un altro stato
  - ***Sottostati***: struttura innestata di stati; disgiunti (sequenzialmente attivi) o concorrenti (concorrentemente attivi)

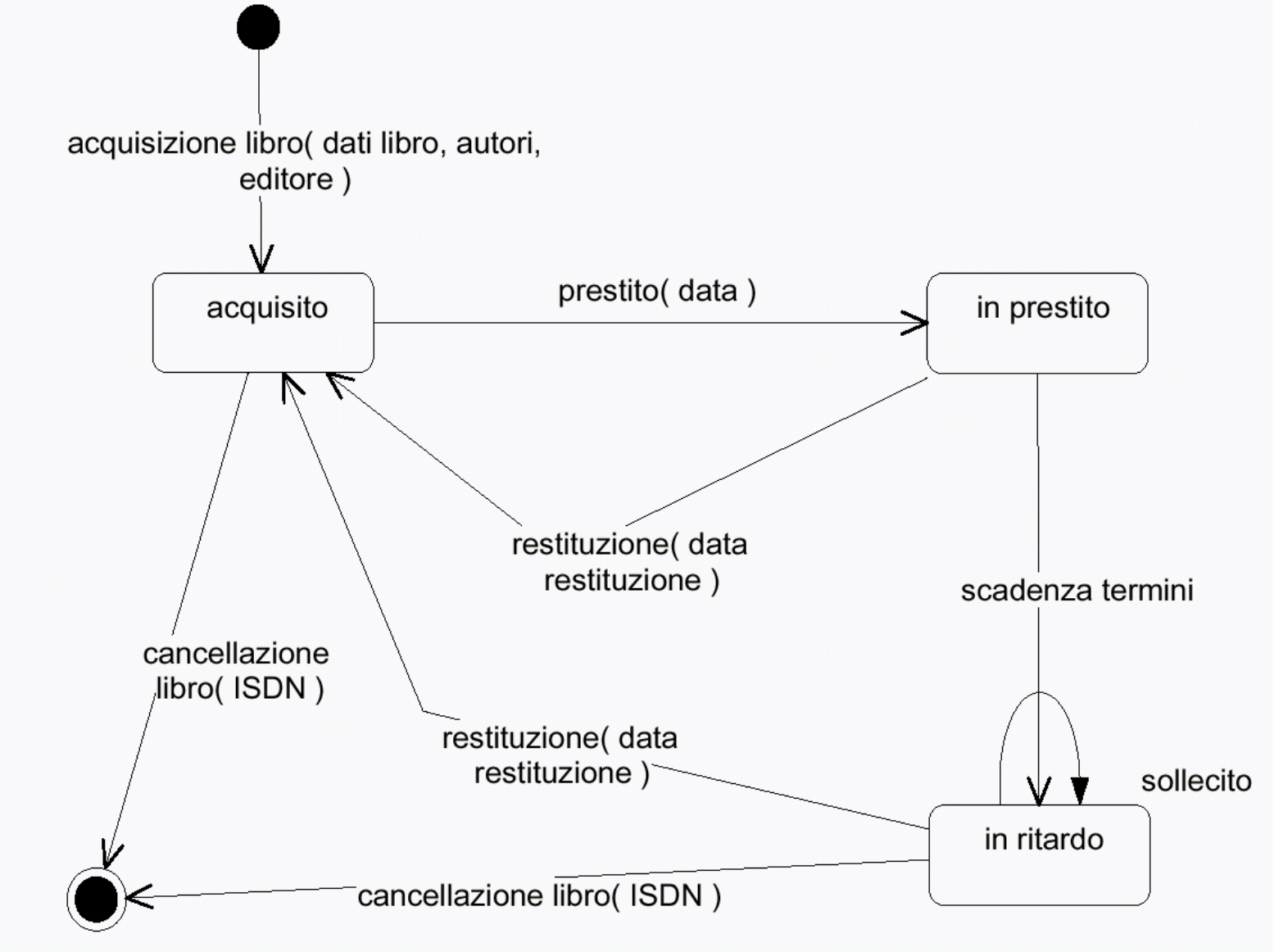
Stato completo



Transizioni

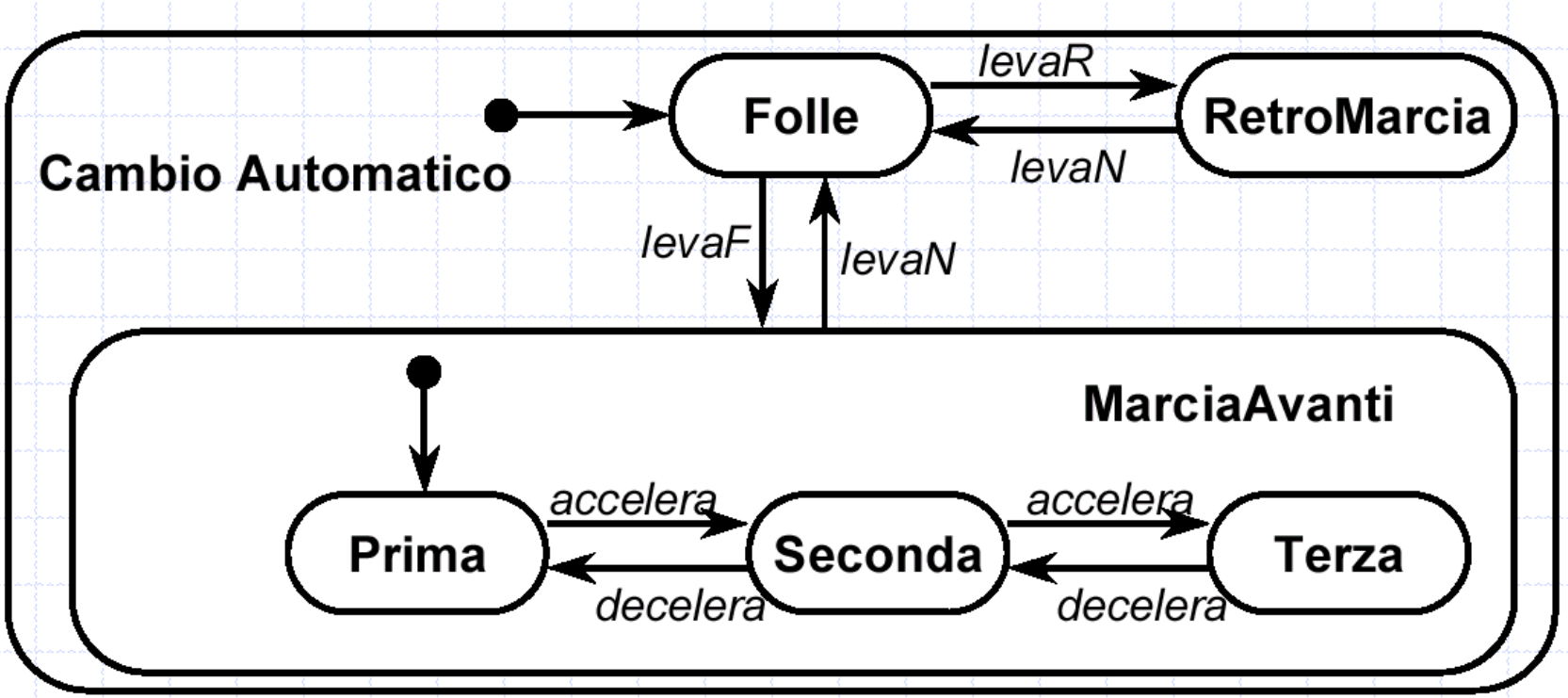
- Cambiamento da uno stato iniziale a uno finale
  - transizione esterna (stato finale diverso da stato iniziale)
  - interna (stato finale uguale a stato iniziale)
- Attributi (opzionali): evento [guardia] / azione
  - Evento
    - segnale, messaggio da altri oggetti, passaggio del tempo, cambiamento
  - Condizione di guardia
    - La transizione occorre se l'evento accade e la condizione di guardia è vera
  - Azione
    - E' eseguita durante la transizione e non è interrompibile (durata istantanea)
  - Transizioni senza eventi (triggerless) scattano
    - con guardia: se la condizione di guardia diventa vera
    - senza guardia: se l'attività interna allo stato di partenza è completata

Esempio: Libro



Sottostati sequenziali (decomposizione OR)

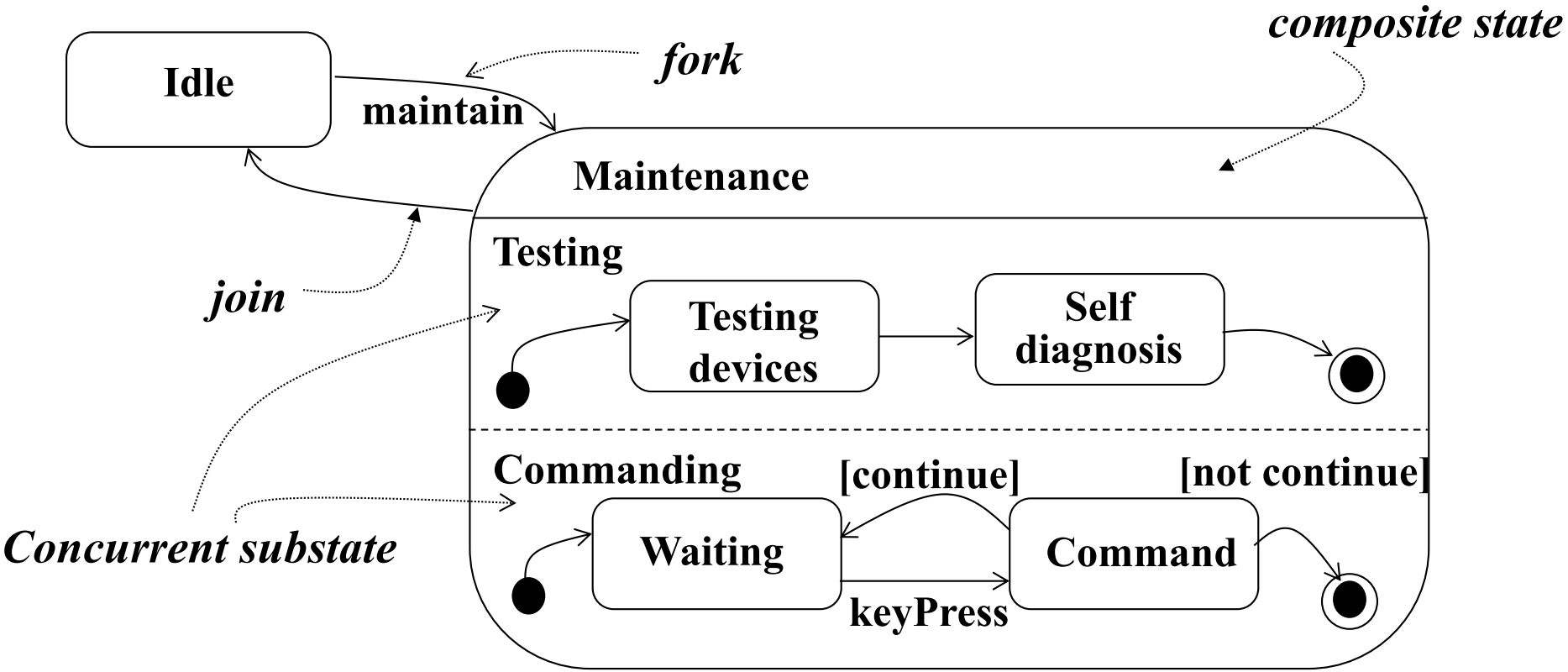
- Un macro stato equivale ad una scomposizione OR degli stati
- I sottostati ereditano le transizioni dei loro superstati



Concurrent substates (decomposizione AND)

- Più state machine sono eseguite in parallelo entro lo stato che li racchiude
- Se un substate machine raggiunge lo stato finale prima dell'altro, il controllo aspetta lo stato finale dell'altro
- Quando avviene una transizione in uno stato con concurrent substate, il flusso di controllo subisce un fork per ciascun concurrent substate; alla fine esso si ricompone in un unico flusso con uno join

Esempio


composite state
Idle
fork
maintain
Maintenance
Testing
join
Testing devices
Self diagnosis
Commanding
[continue]
[not continue]
Concurrent substate
Waiting
Command
keyPress

Unified Modeling Language

Packages

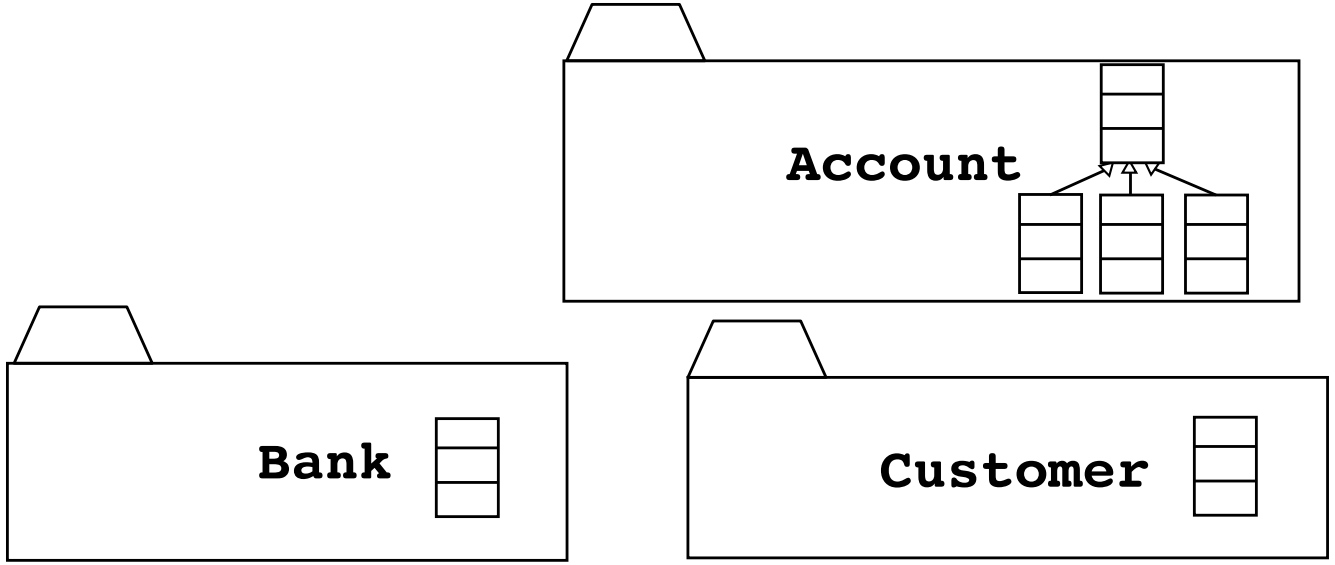
Component diagrams

Deployment diagrams

Packages

Packages

- Packages help you to organize UML models to increase their readability
- We can use the UML package mechanism to organize classes into subsystems



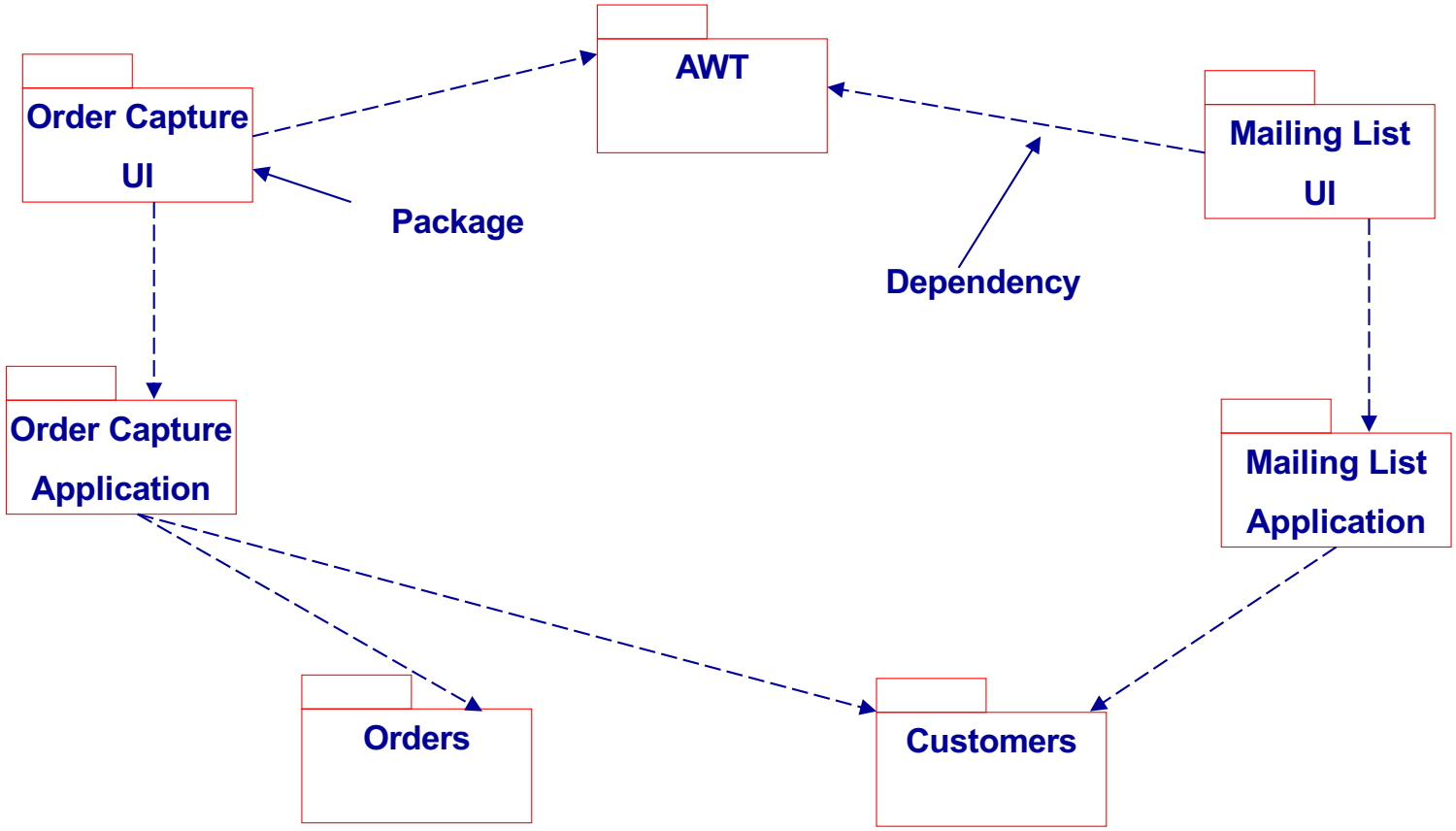
- Any complex system can be decomposed into subsystems, where each subsystem is modeled as a package.

Package Diagrams

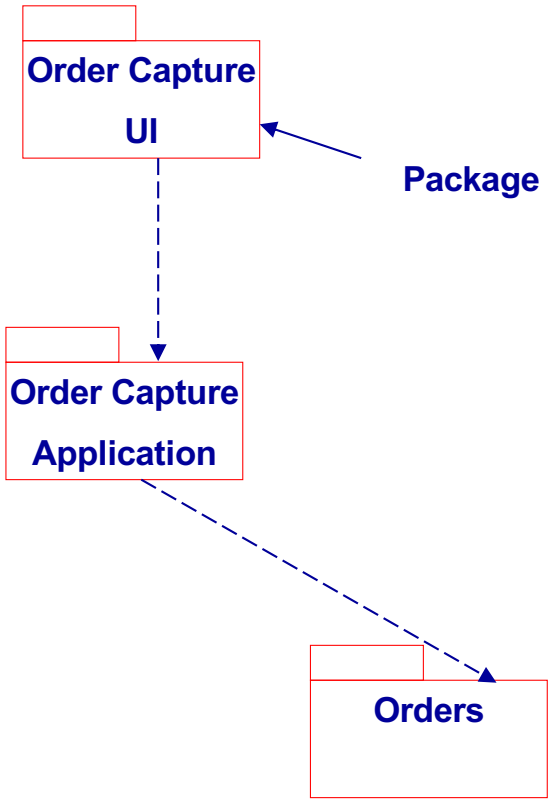
- Un package raccoglie un insieme di classi la cui interazione è funzionale all'assolvimento di una certa responsabilità da parte del sistema.
- Un package può raccogliere anche altri diagrammi …

Package Diagrams: Esempio

Separazione tra UI, Logica dell'Applicazione e Gestione Dati …

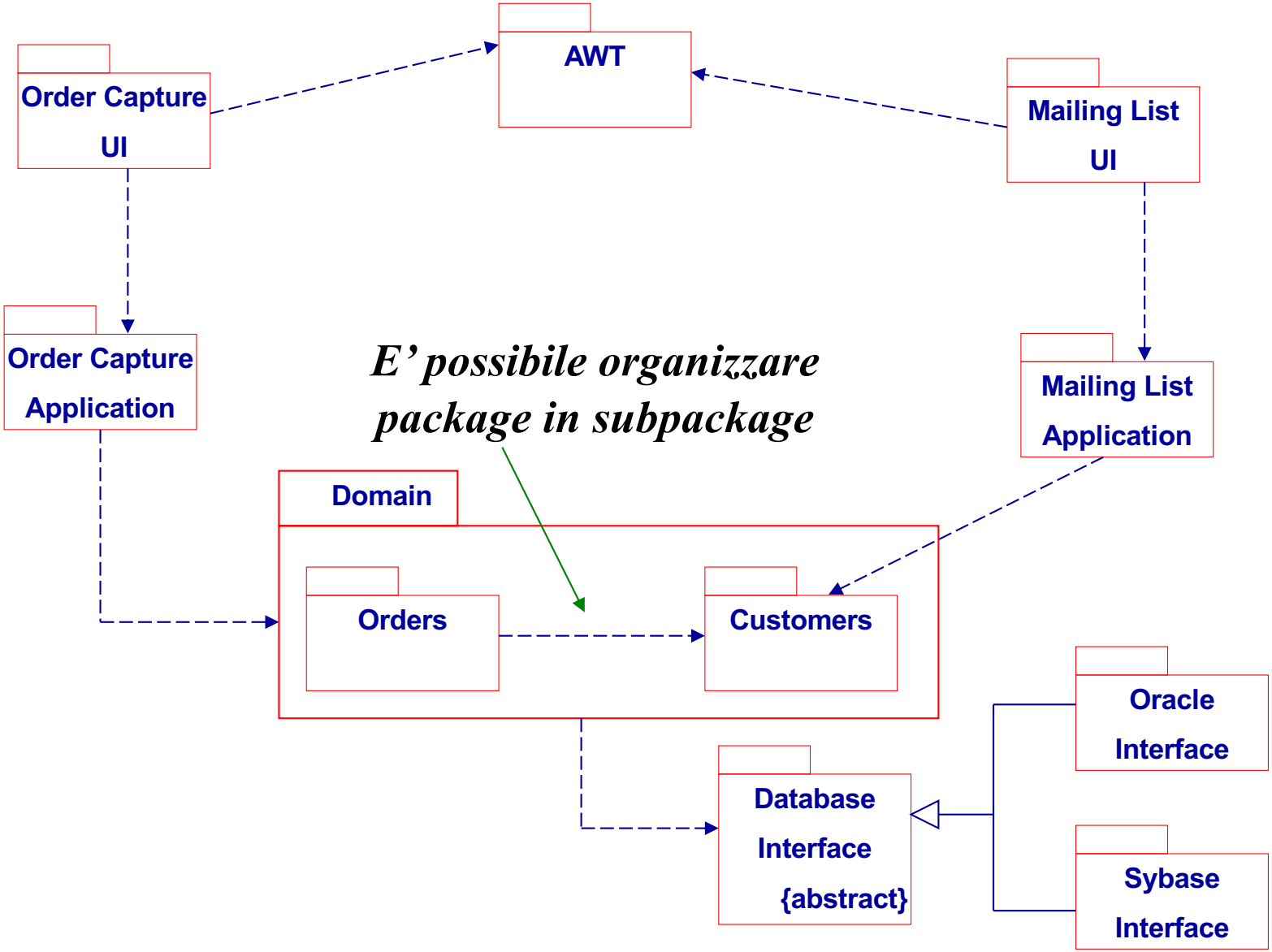


Package Diagrams: Dipendenze



- Una dipendenza tra due package sussiste se esiste una dipendenza tra almeno due classi appartenenti a ciascuno dei package in questione.
  - Se qualche classe in Orders cambia è opportuno verificare se qualche classe di Order Capture Application deve essere a sua volta modificata.
- Teoricamente solo delle modifiche all'interfaccia di una classe dovrebbero interessare (in modifica) altre classi.
  - Le dipendenze non sono transitive ...

Package Diagrams: Esempio



Package Diagrams

- L'organizzazioni in Package di un sistema è necessaria per dominare la complessità dello stesso.
- Un Package Diagram consente alle diverse figure coinvolte nello sviluppo del sistema una sua rapida comprensione.
- In fase di Manutenzione e Testing le dipendenze diventano di vitale importanza.

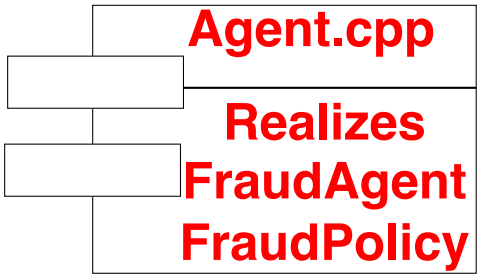
Component Diagrams

Component Diagram

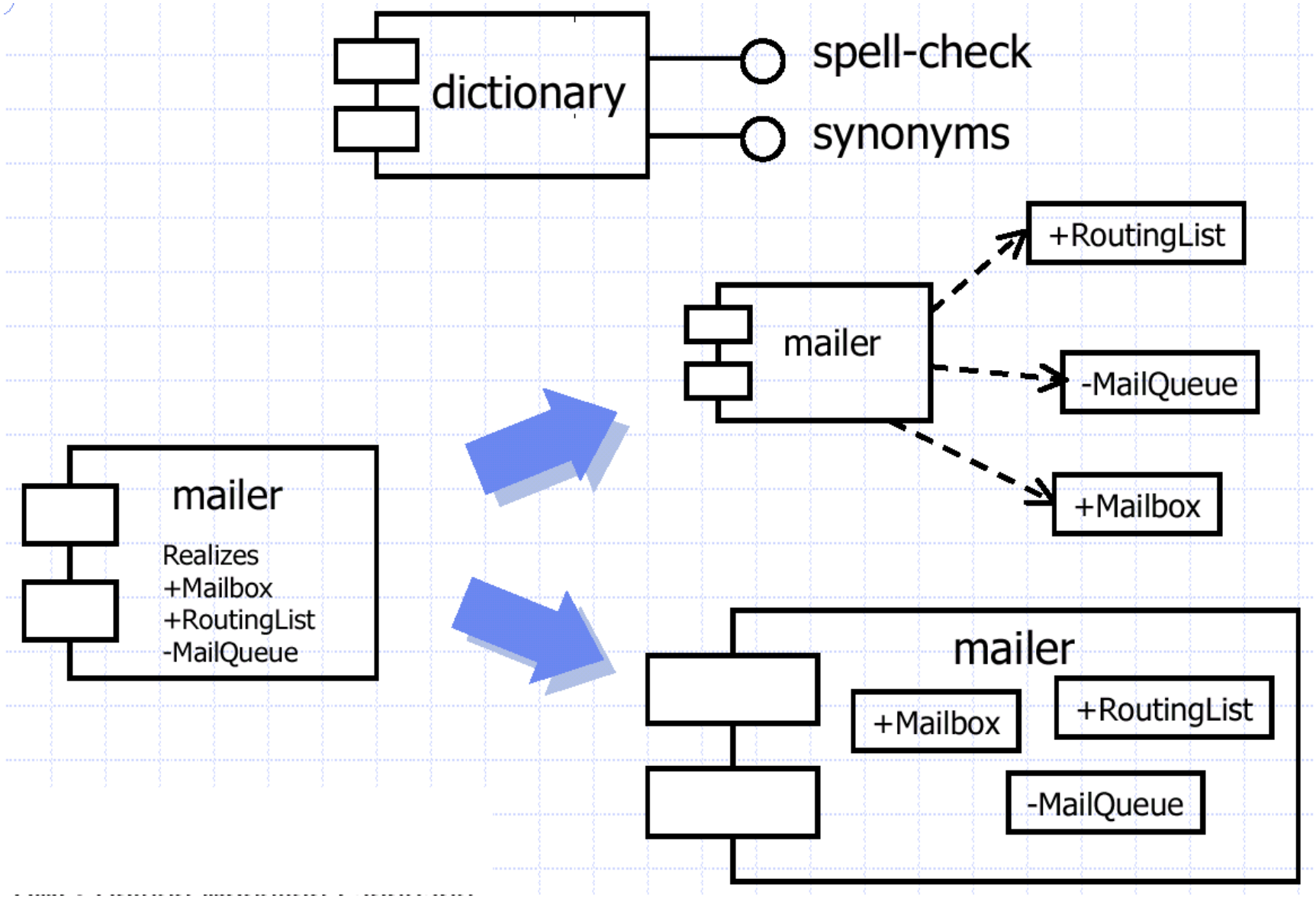
- Rappresentano l'implementazione del sistema
  - Un componente rappresenta un pezzo "fisico" dell'implementazione di un sistema
- Definiscono le relazioni fra i componenti software che realizzano l'applicazione
  - sorgenti, binari, eseguibili, …
- Parte della specifica architetturale …
- Evidenziano l'organizzazione e le dipendenze esistenti tra componenti
  - I diversi componenti offrono e usano interfacce specifiche
  - Primo passo verso *Component Programming*

Componenti

- Un componente ha un nome e una locazione
  - è mostrato, tipicamente, con il solo nome; per le classi è possibile 'adornarlo' con compartimenti riportanti altri dettagli
  - è possibile indicare le relazioni tra component e class e/o interface che essi realizzano
- I componenti (come a livello logico le classi) possono essere raggruppati in package
  - Se i componenti sono file, i package sono cartelle/ directory



Componenti: Esempi

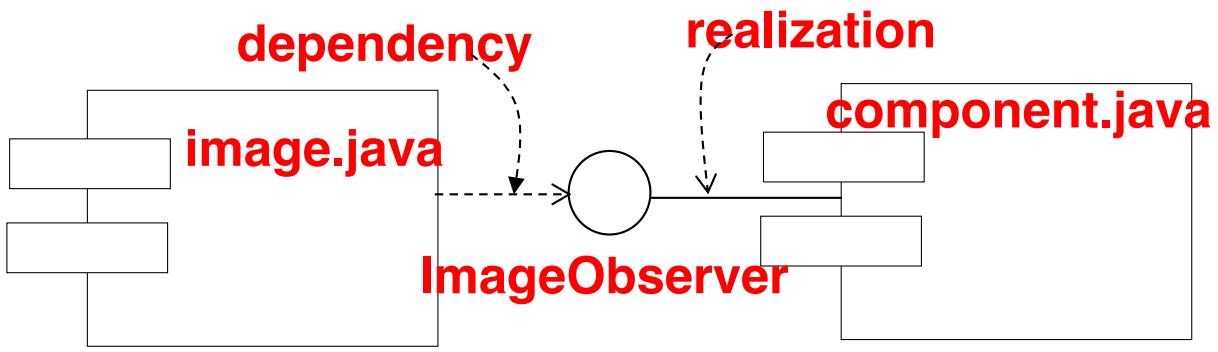


Componenti e interfacce

- Un componente può utilizzare le interfacce di altri componenti e realizzare un insieme di interfacce
  - "*a component conforms to and provides the realization of a set of interfaces* " (Booch '99)
- Le interfacce rappresentano una collezione di operazioni usate per specificare un servizio
  - Java e C++ più recente permettono di definire "interface"
- I sistemi di middleware più comuni
  - COM+ (ora .NET), CORBA, Enterprise JavaBeans,

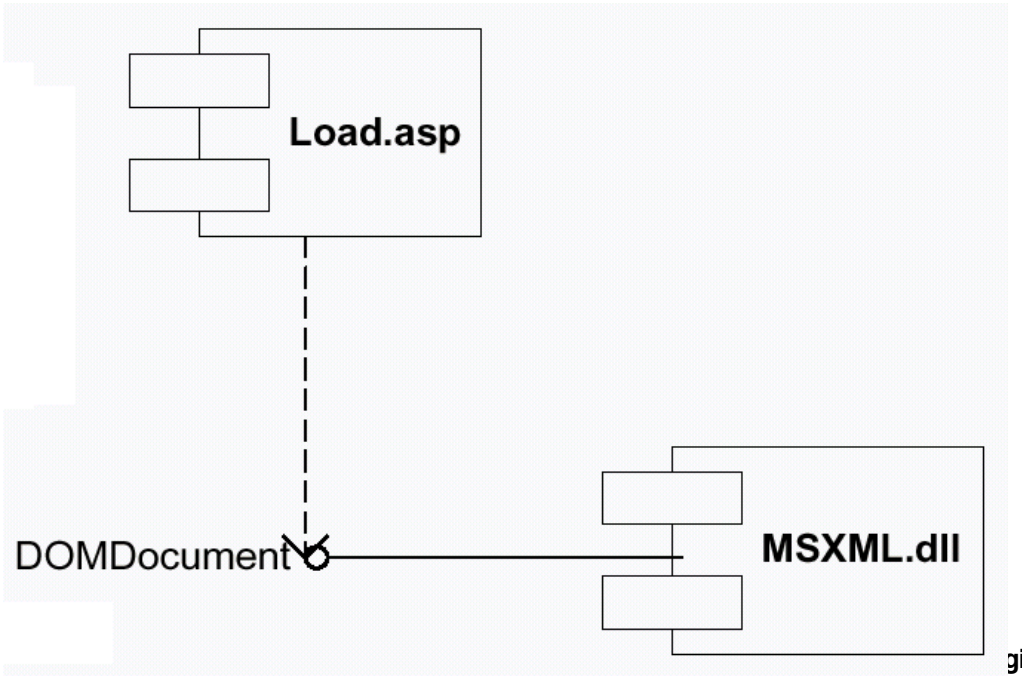
  sono basati su componenti e usano interfacce per legare le componenti tra loro

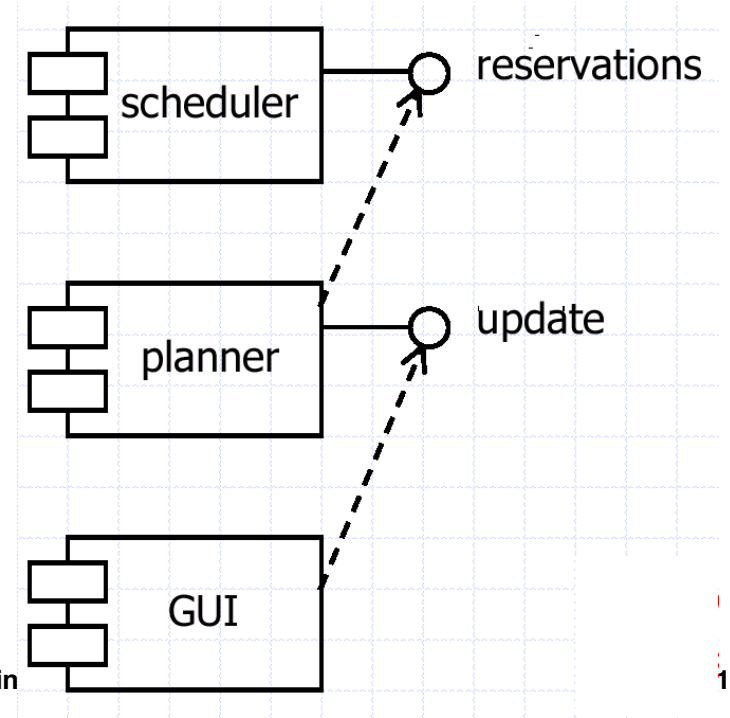
Componenti e Interfacce: Esempi



usa la interface

implementa la interface





Componenti e classi

- Livelli di astrazione distinti
  - Le classi rappresentano astrazioni logiche mentre le componenti rappresentano cose fisiche (hanno una locazione e sono viste dal software di base)
  - I servizi di una classe sono direttamente accessibili da programma tramite operazioni pubbliche, mentre i servizi di un componente sono accessibili tramite le interfacce che implementa
- Le classi sono realizzate da componenti
  - Tipicamente, un component mappa una o più class, interface o collaboration
  - Spesso un componente coincide con un package che racchiude un insieme di classi …
  - Importante mantenere la tracciabilità tra classi e componenti

Sostituibilità di un componente

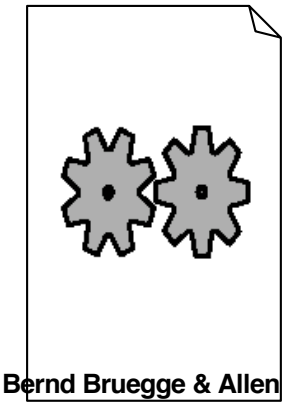
- I componenti collaborano tra di loro tramite interfacce
  - Interfaccia esportata: l'interfaccia che il componente fornisce come servizio alle altre componenti
    - Ci può essere più di un'interfaccia esportata
  - Interfaccia importata: l'interfaccia che il componente usa
- Un componente è sostituibile, ovvero è possibile sostituire un componente con un altro che è conforme alle stesse interfacce e che preserva l'interfaccia precedentemente esportata
  - È possibile estendere l'interfaccia esportata o aggiungerne nuove
  - E' possibile aggiungere nuove importazioni

Tipi di Componente

- E' possibile distinguere tre tipi di componenti
  - deployment components: i componenti necessari e sufficienti per formare un sistema eseguibile (DLL, programmi eseguibili, ...)
  - work product components: componenti che non partecipano direttamente nel sistema eseguibile ma che sono frutto del lavoro fatto per creare il sistema eseguibile;
    - sono, essenzialmente, il 'residuo' del processo di sviluppo (source code file, data file usati per la creazione di deployment components, ...)
  - execution component: compnenti creati come conseguenza del sistema eseguibile (es. COM+ object, JCL, ...)

Componenti e UML stereotypes

- UML definisce 5 tipi di stereotypes standard per i component
  - <<executable>>: un component che può essere eseguito in un nodo
  - <<library>>: una libreria statica o dinamica
  - <<table>>: una tabella di un database
  - <<file>>: un component contenente codice sorgente o dati
  - <<document>>: un component rappresentante un documento
- Non sono definite icone specifiche … alcune tra le più usate:

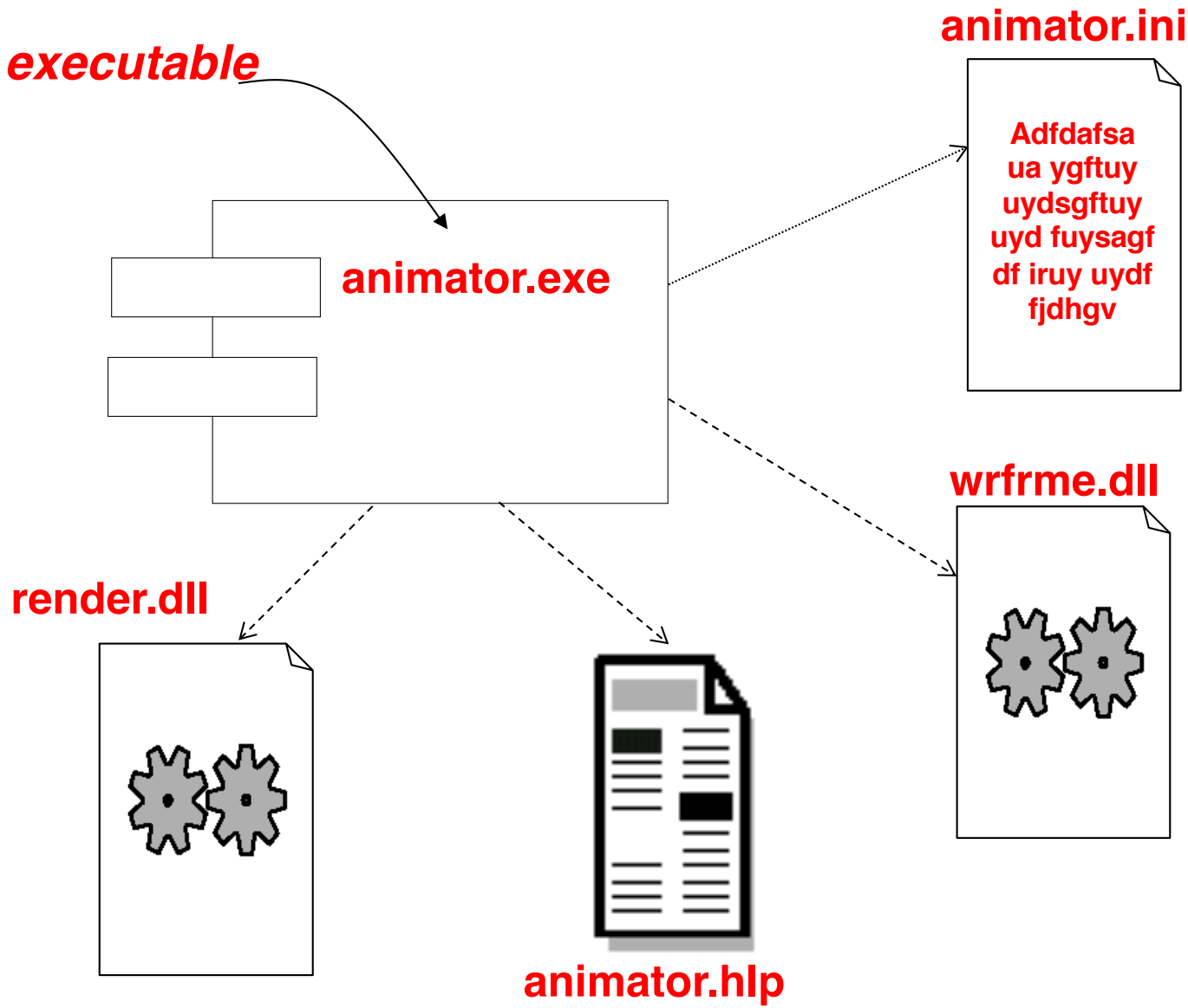


library

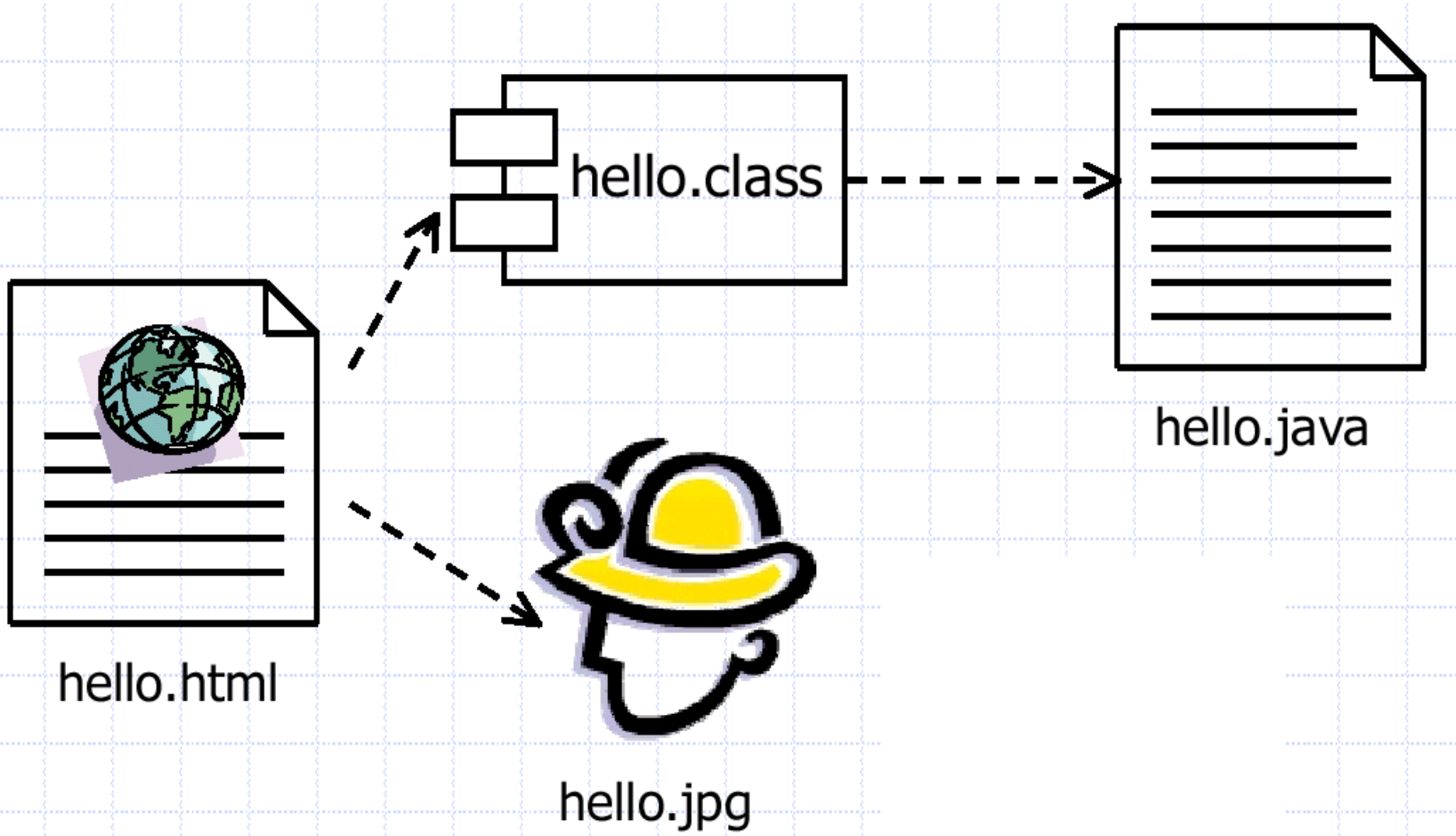
document

file

Esempio


executable
animator.ini
Adfdafsa
ua ygftuy
uydsgftuy
uyd fuysagf
df iruy uydf
fjdhgv
animator.exe
wrfrme.dll
render.dll
animator.hlp

Esempio con altre icone …

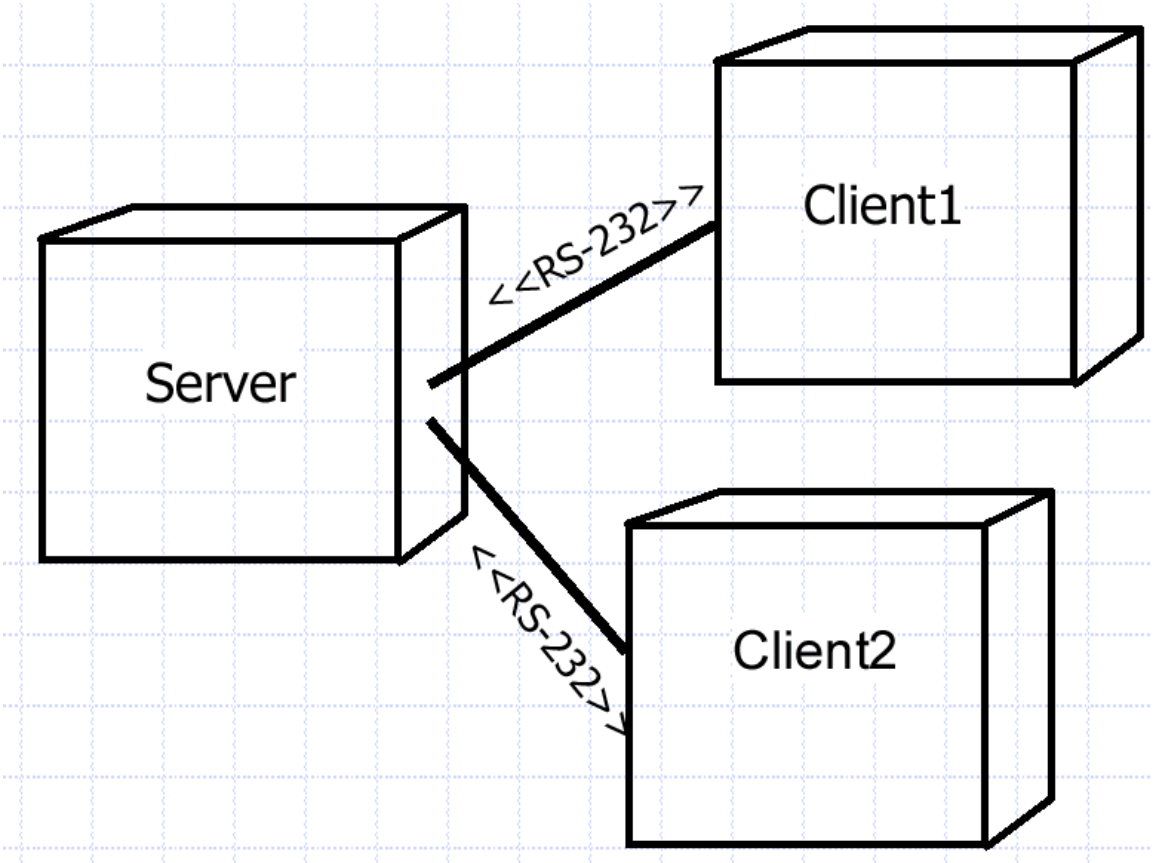


Deployment Diagrams

Deployment Diagram

- Anche detto diagramma di allocazione o di dislocazione
- Elementi: nodo, connessione tra nodi
- Permette di rappresentare, a diversi livelli di dettaglio, l'architettura fisica del sistema
- Permette anche di evidenziare la configurazione dei nodi elaborativi in ambiente di esecuzione (run-time), e dei componenti, processi ed oggetti ubicati in questi nodi

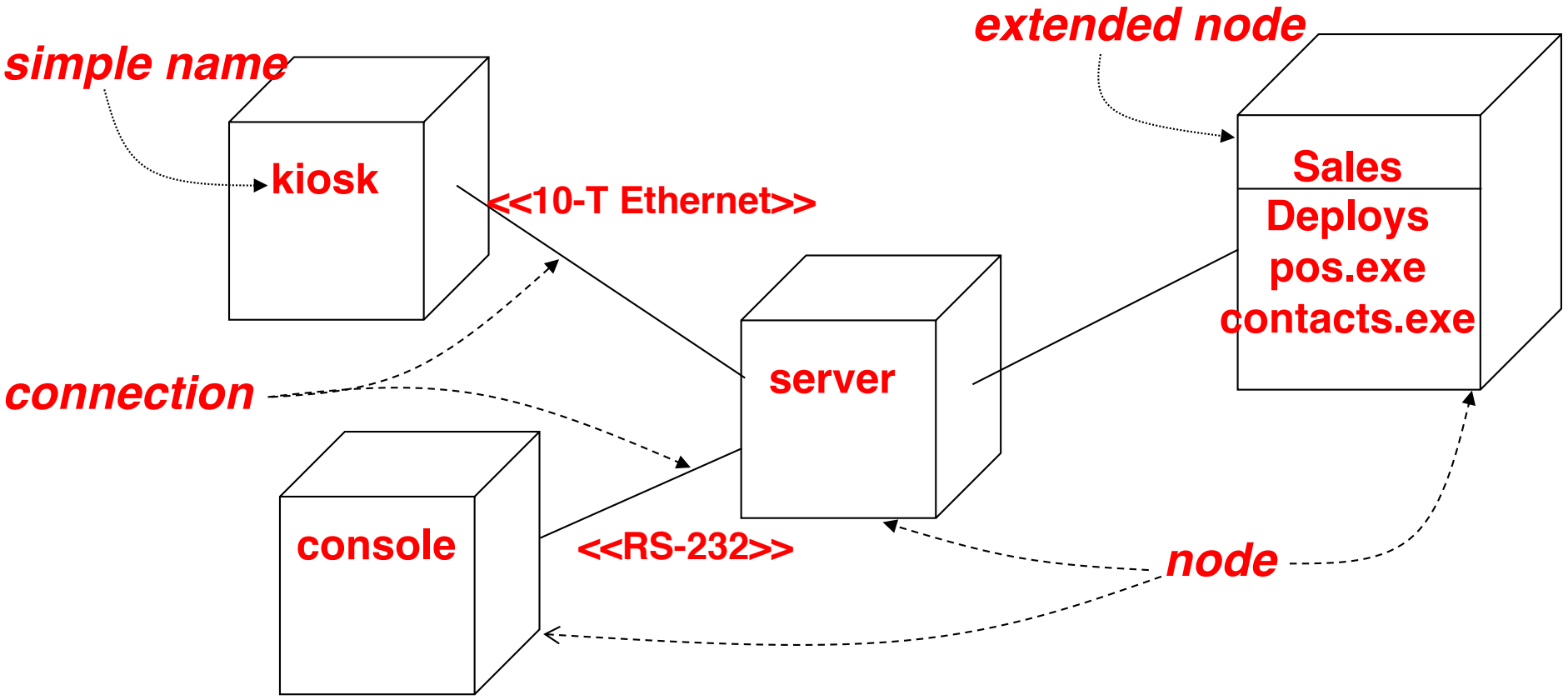
Esempio



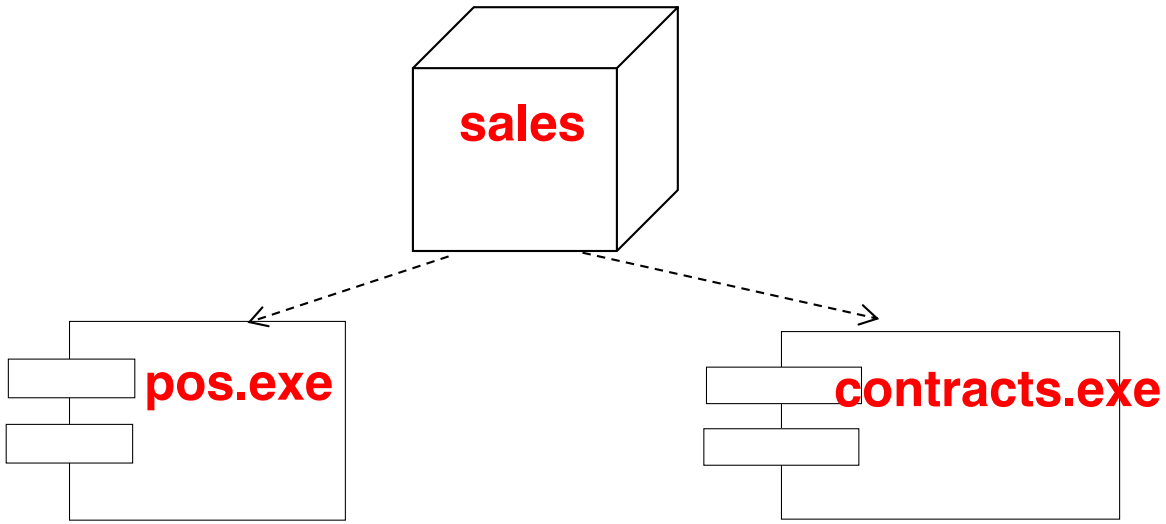
Deployment Diagram: Nodi

- Ciascun nodo è identificato da un nome
- Un nodo può riportare ulteriori dettagli in compartimenti addizionali o usando tagged value
- Un nodo può essere in connection con altri nodi, ed avere relationship con componenti e altri nodi
  - componenti sono things che partecipano nell'esecuzione di un sistema; nodi sono things che eseguono componenti
  - componenti rappresentano il packaging fisico di altri elementi logici; nodi rappresentano l'allocazione fisica di componenti

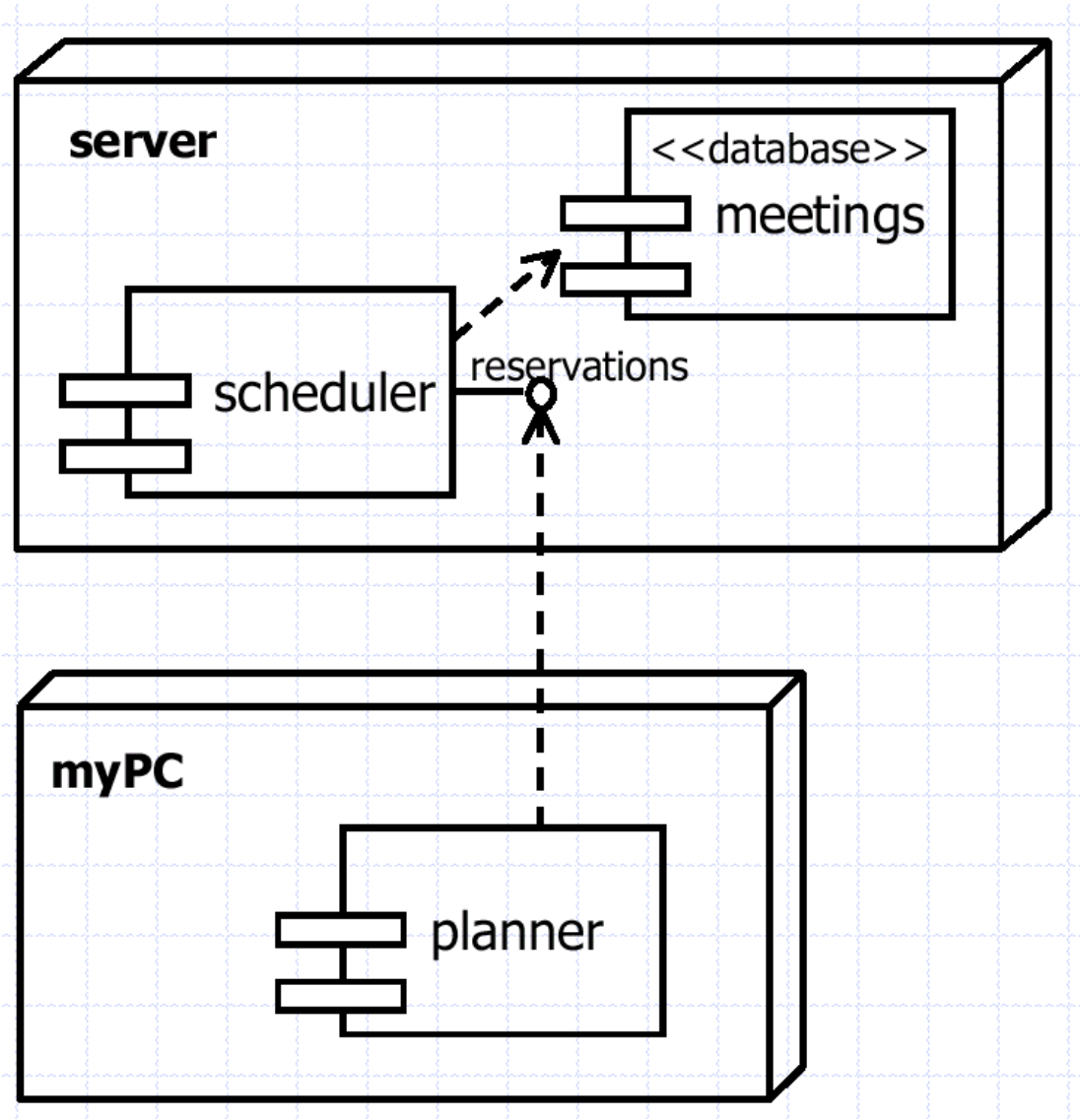
Esempio


simple name
kiosk
<<10-T Ethernet>>
extended node
Sales
Deploys
pos.exe
contacts.exe
server
connection
console
<<RS-232>>
node

Nodi e Componenti



Deployment Diagram: Esempio (1)



Deployment Diagram: Esempio (2)