2023
Ingegneria del software: teoria parte 1

Giorgio Bruno

Dip. Automatica e Informatica
Politecnico di Torino
email: giorgio.bruno@polito.it

Obiettivi e risultati del corso

Il corso presenta i concetti fondamentali dell'ingegneria del software e le discipline principali che essa comprende. Particolare attenzione è dedicata alla formalizzazione dei requisiti mediante l'uso di modelli UML per la parte strutturale e di reti di Petri estese per la parte comportamentale.

Risultati di apprendimento attesi

Le conoscenze trasmesse riguardano i concetti, i prodotti, le attività e i ruoli principali connessi con l'ingegneria del software.

Le abilità che gli studenti acquisiranno riguardano la capacità di formalizzare i requisiti di un sistema software mediante *modelli strutturali e comportamentali* e la capacità di risolvere semplici problemi inerenti alle varie discipline del processo di sviluppo del software.

Ingegneria del software

Con le iniziative in corso, ad es. Industry 4.0, smart cities, guida autonoma e robotica, il fabbisogno di software aumenta e l'ingegneria del software deve dare il suo contributo in termini di metodi, strumenti, processi e best practice.

Il paradigma *model-driven* è un fattore importante: l'uso di modelli facilita la comprensione e agevola il dialogo tra persone con background diversi. Inoltre la *generazione* (anche parzialmente) *automatica* di codice incrementa l'efficienza del processo di sviluppo.

L'ingegneria del software è una disciplina ampia. Non riguarda soltanto lo sviluppo del software, ma può aiutare il pensiero nel formulare *rappresentazioni più chiare e rigorose* di ciò che si vuole conseguire.

Programma del corso

Parte 1

Sviluppo del software: modelli e attività di supporto

Definizione dei requisiti

Uso di modelli: la famiglia UML

Classi e relazioni

Casi d'uso

Diagrammi di attività

Modelli di stato

Modelli di dataflow

Strutture

Patterns

Programma del corso

Parte 2

Reti di Petri

Proprietà comportamentali

Sottoclassi

Reti che trattano oggetti

Programma del corso

Parte 3

Processi di business: integrazione tra modelli informativi e modelli di processo

Reti che trattano entità

Building block e dataflow

Processi B2B

Modelli di collaborazione

Dall'analisi dei requisiti ai modelli

Service-Oriented Architecture

Domain-driven design

Microservices

Overview BPMN

Programma del corso

Parte 4

Attività di verifica: ispezioni e testing

Gestione della configurazione

Introduzione a Git

Gestione dei progetti

Processo del software: il modello CMM

Qualità del software e metriche

Sviluppo agile

DevOps

Esame

L'esame consiste in una prova scritta che comprende 4 esercizi.

Esercizio 1. Dati i requisiti in forma testuale di un processo B2B, si tratta di definire i modelli di collaborazione del processo, il modello informativo del processo e il modello comportamentale del processo.

Esercizio 2. Riguarda l'analisi delle proprietà di una rete di Petri.

Esercizio 3. Chiede la determinazione del numero minimo di test per la copertura di vari criteri in un white box testing.

Esercizio 4. Può consistere in 4 domande con risposta chiusa *oppure* nella definizione di un modello dati i requisiti.

Esempi di temi d'esame sono forniti con il materiale didattico.

La durata dell'esame non supera le due ore.

Parte 1 del corso

Definizioni di Software Engineering

The systematic approach to the development, operation, maintenance and retirement of software (IEEE glossary of software terminology).

The application of scientific principles to:

- the orderly transformation of a problem into a working solution;
- the subsequent maintenance of that software through the end of its useful life (Alan M. Davis).

Software life cycle and software process

A software product has a life cycle that starts when the product is conceived and finishes when the product is no longer available for use. It is made up of a number of **phases**.

Each phase has a set of inputs, a set of outputs and is carried out by means of a number of **activities**.

The activities and the order they are carried out form the **software process**.

The development of high quality software products requires the *software process* to be based on sound *software engineering principles* and *practices*.

La qualità del prodotto dipende dalla qualità del processo.

Software life cycle

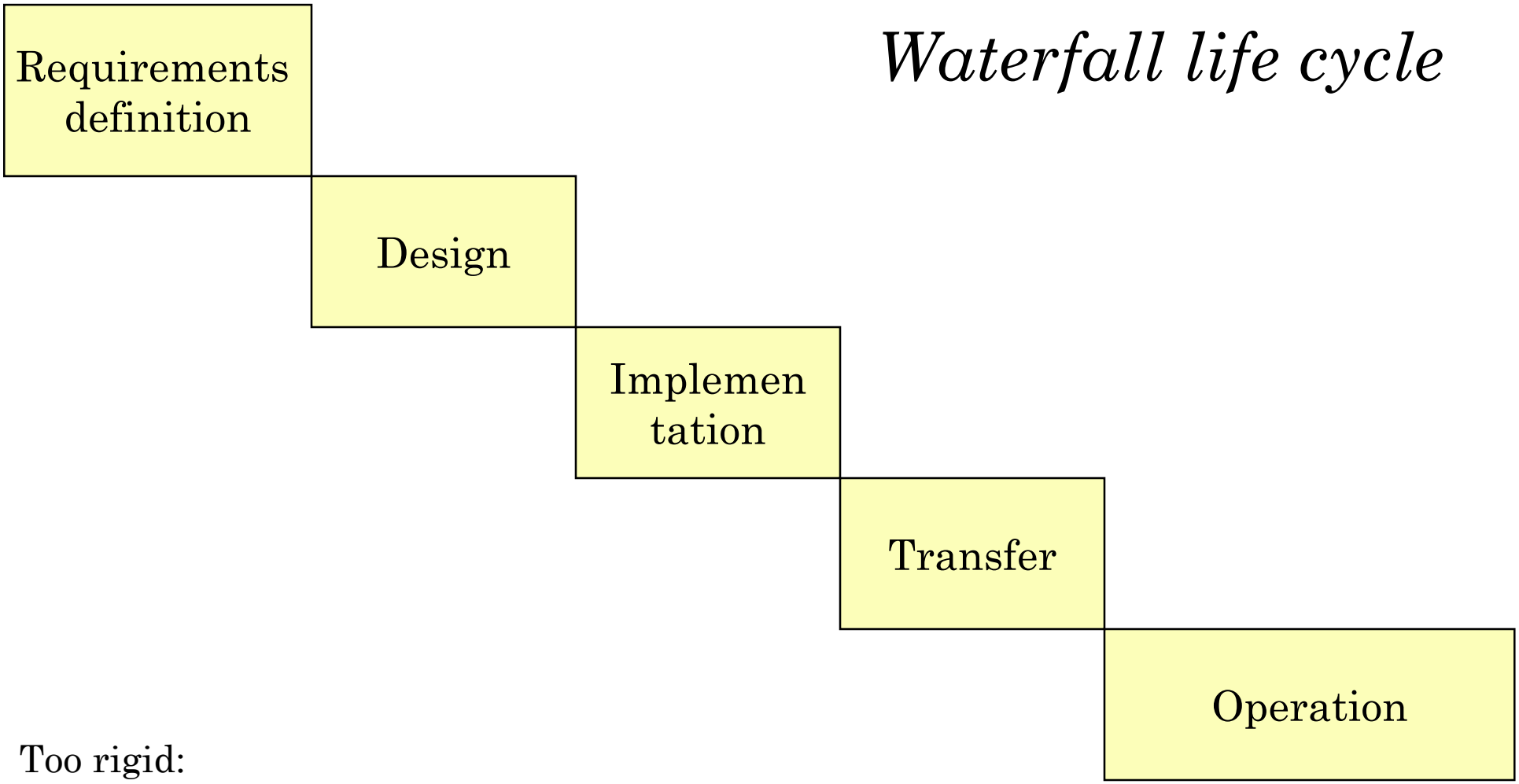
waterfall

- incremental
- evolutionary

unified

support activities

Waterfall life cycle



Too rigid:
software release at the very end (big bang release)
requirements frozen after the first phase

Phases

Requirements definition

overall description of the system in business terms, major functionalities and constraints

Architectural design

overall system architecture, interactions between components, linking requirements to components, make-or-buy decisions

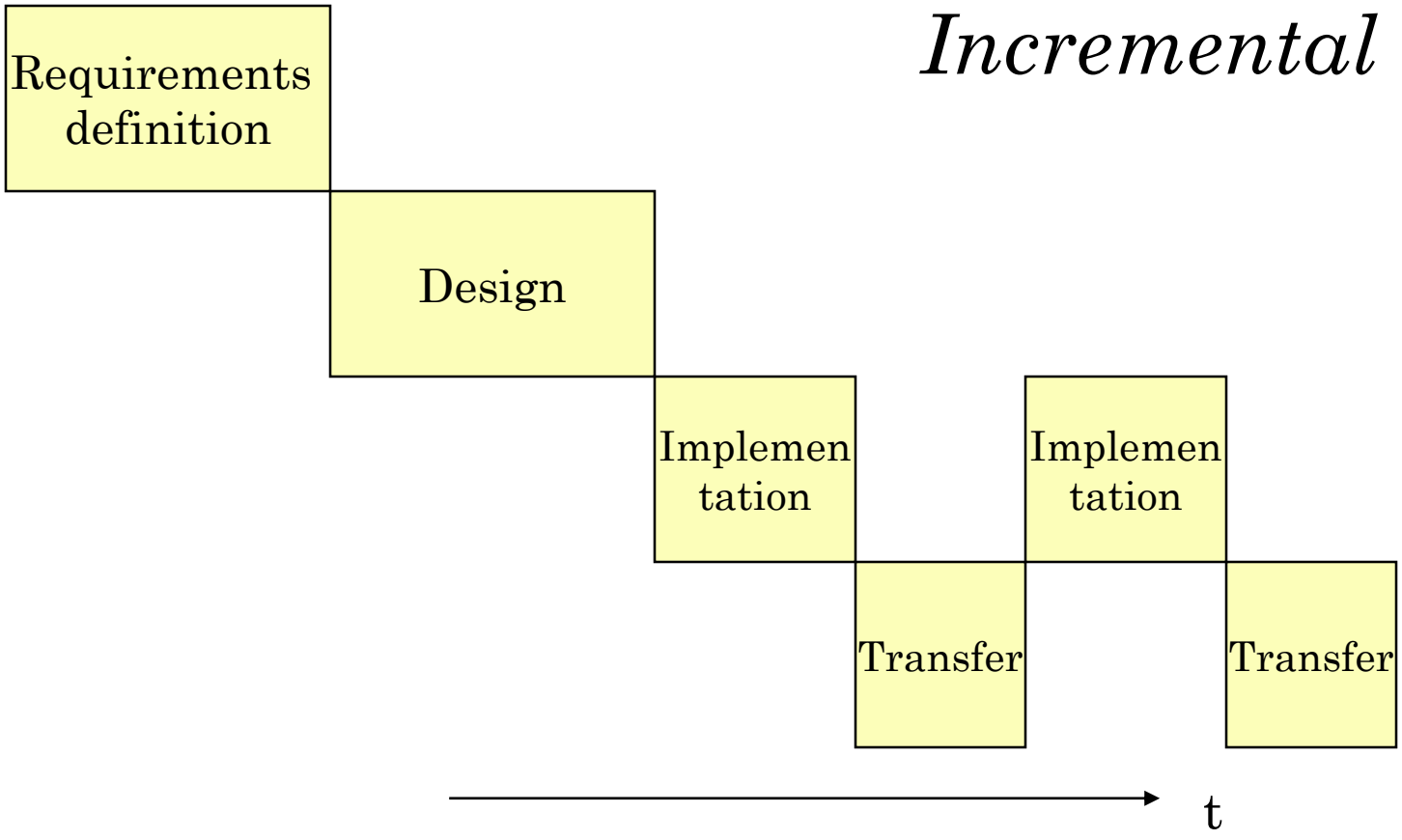
Implementation

detailed design, development and testing

Transfer

acceptance testing and deployment (making the system ready for use)

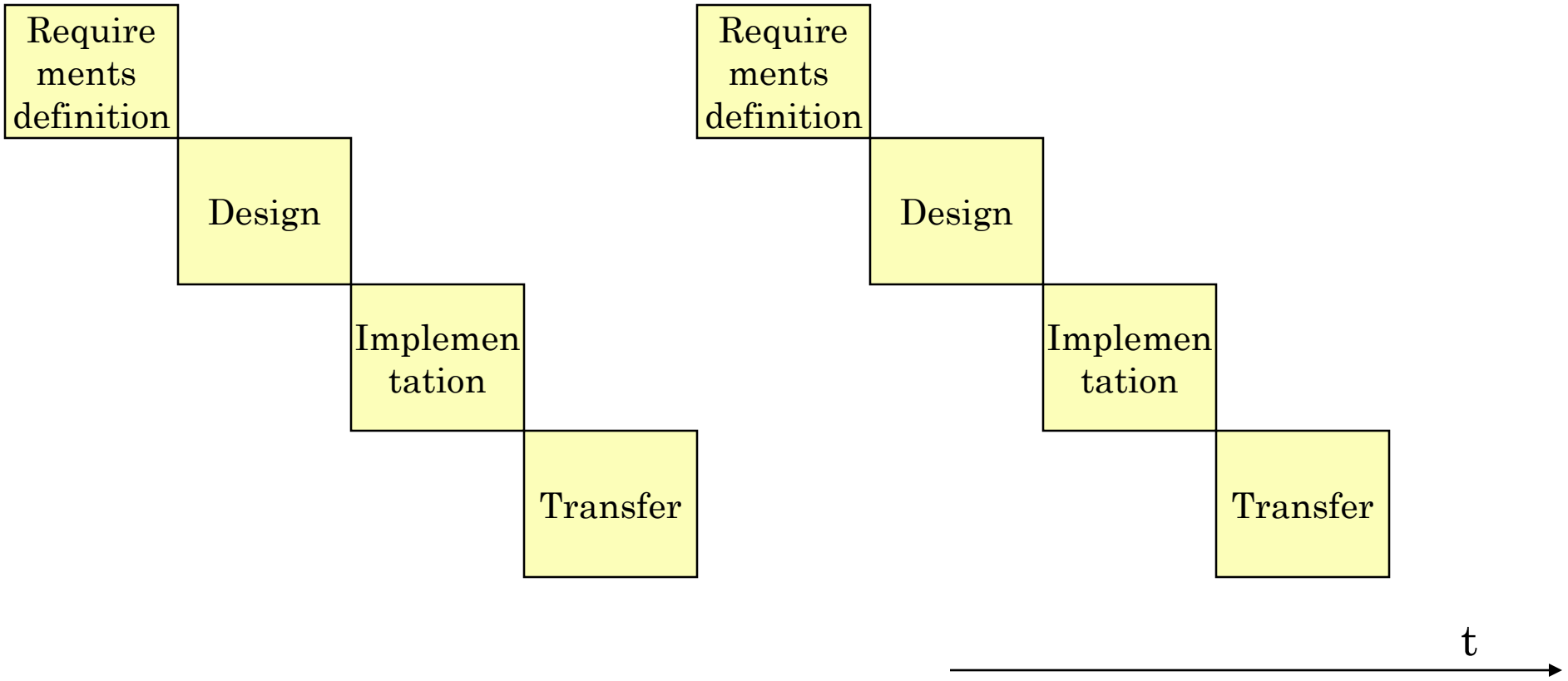
Incremental life cycle



Implementation and release in pieces rather than in one pass.

Incremental release demands a careful selection of the order in which the parts are released and the overall project cost may increase.

Evolutionary life cycle



All the phases can be repeated. This is useful when the initial requirements are poorly understood or unstable. It can give users an early operational capability. Each version of the system is a kind of prototype.

Prototyping

It is the technique of constructing a partial implementation of a system so that customers, users or developers *can learn* more about a problem or a solution to a problem.

Two approaches: throwaway and evolutionary.

A *throwaway prototype* should be **quick and dirty**. It can be used to validate a user interface, to check whether a particular architecture has the potential to meet the requirements, or to validate a particular algorithm.

On the contrary, since the *evolutionary prototype* evolves into the final product, it must exhibit all the quality attributes of the final product. It will not be particularly rapid.

Unified Process

It combines the incremental and evolutionary approaches, and is based on 4 phases. It uses UML diagrams.

A well known refinement is RUP (Rational Unified Process).

Inception: business case (business modeling), key requirements and constraints, risk assessment, preliminary project schedule and cost estimate.

Elaboration: detailed requirements analysis (resulting in use cases and class diagrams), design of the architecture and validation through prototypes, development plan.

Construction: implementation supported by various detailed UML diagrams.

Transition: deployment.

A type of activity does not coincide with a specific phase, although it may be the most important one in that phase.

Unified Process

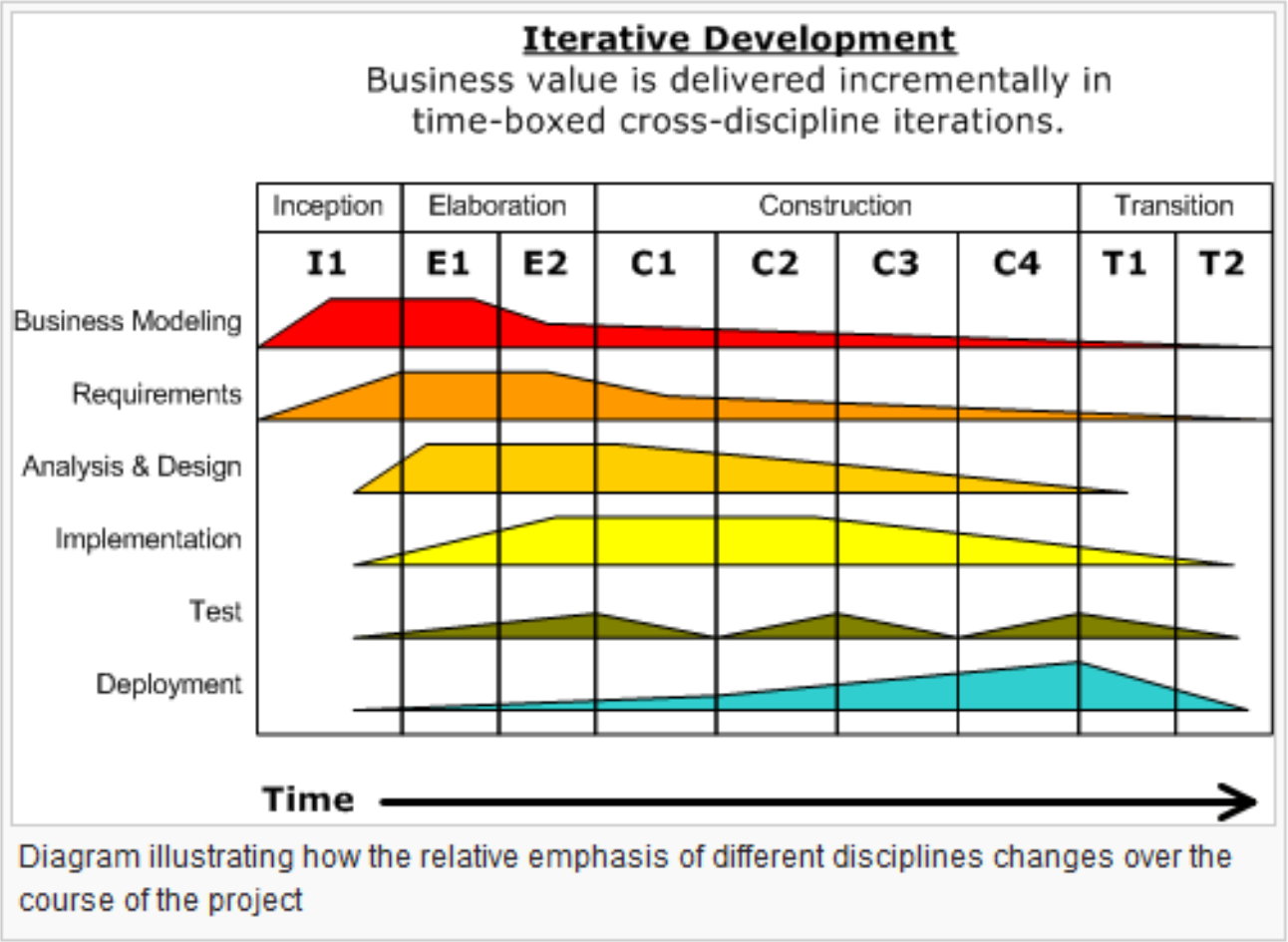


Diagram illustrating how the relative emphasis of different disciplines changes over the course of the project

from https://en.wikipedia.org/wiki/Unified_Process

Business modeling

Describes the context in which the intended software system will operate and the relationships with the other subsystems from a business point of view; it is a *bridge between business analysts and software analysts*.

Support activities

Project management: planning, staffing, monitoring, controlling and leading.

Version control and configuration management: defining the elements of the system, controlling their release and change.

Verification (Are we building the product right?):

static analysis: *inspection* (detecting defects through peer reviews);

testing.

Validation (Are we building the right product?): ensuring that the product meets the needs of the users.

Quality assurance: ensuring that standards and procedures are established and followed.

Nuovi approcci al software life cycle

Nella parte 4 si esamineranno

Agile development

SCRUM

Continuous integration

Continuous delivery

Conceptual models

Using models to study the properties of complex systems is common to all disciplines. Examples of models are scaled physical systems and mathematical representations.

In general, *a model is an abstract and rigorous representation* of a system: it enables the user to reason about the important properties of the system.

Unique to software development is the notion of *operational models*, i.e., models that can executed in a suitable support environment or that can automatically be turned into operational software.

model = code

model → code

model-driven development

UML

UML (Unified Modeling Language)

is made up of a number of modeling languages/notations; it is not a global methodology.

OMG standard: http://www.omg.org

Major kinds of diagrams

- Structural: class models, object models
- Behavioral: use cases, activity models, state models, sequence models

OCL (Object Constraint Language)

Class models

The system under consideration is made up of individuals called *objects*, and objects belong to *classes*. There are systematic links between objects and they are represented by links, called *relationships*, between their classes.

Classes may have properties called *attributes*.

A *class model* shows the classes, attributes and relationships.

An *object model* is a particular realization of a class model: it shows a number of interrelated objects for illustrative purposes.

Several flavors:

- *Class model*: technical (implementation) perspective
- *Domain model*: high level (conceptual, business) perspective
- *Information model*: data base perspective

Aspetti rilevanti dei modelli classi-relazioni

Relazioni: associative, composizione, inheritance, ricorsive

Attributi normali e associativi

Espressioni navigazionali

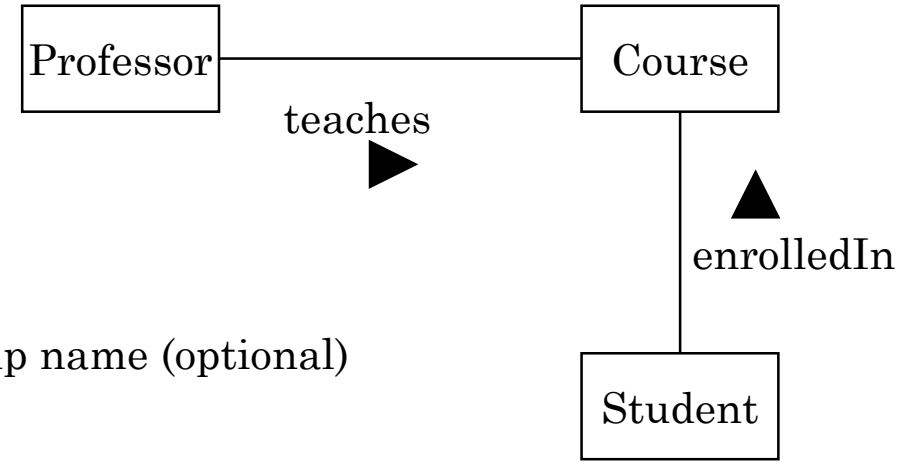
Attributi necessari e relazioni necessarie

Attributi derivati e relazioni derivate

Invarianti e regole di validazione

Example of domain model

In universities, there are professors and students; professors teach courses and students are enrolled in courses.



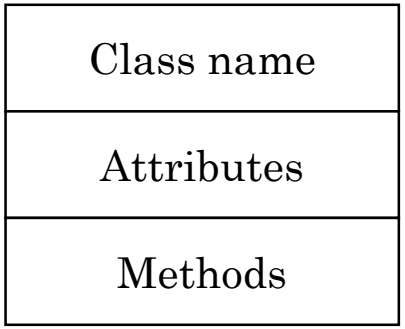
teaches: relationship name (optional)

There are links between professors and courses and between students and courses.

Two binary **associative relationships**; only *binary* relationships are considered.

Classi

Il simbolo di una classe può contenere attributi e metodi.



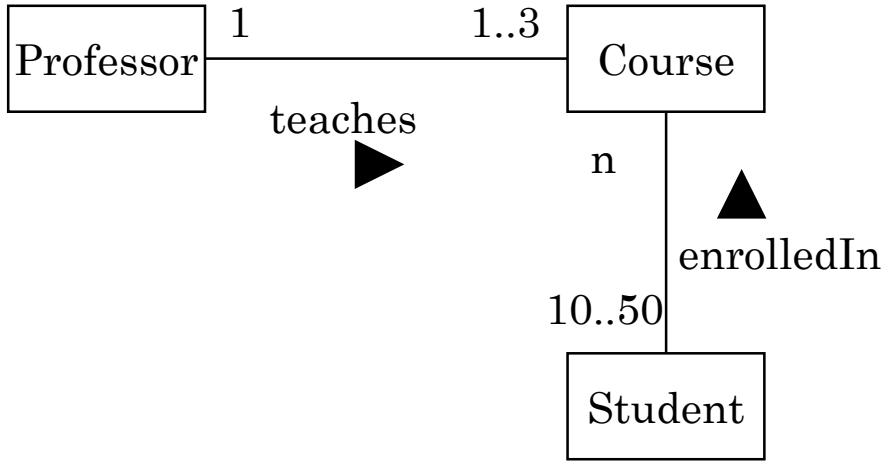
Vista completa

Attributi e metodi possono avere indicatori di visibilità: pubblica (+), privata (-), package (~), protected (#).

Constraints

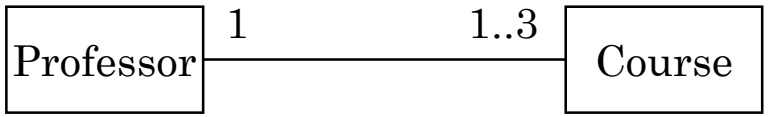
A course is taught by one professor. Professors teach 1..3 courses.

Students are enrolled in a number of courses. The minimum number of enrollees in a course is 10; the maximum is 50.



Multiplicity

How many associations a given object may be involved in?



La molteplicità definisce implicitamente regole di validazione.

Indicator	**Meaning**
0..1 or 0,1	Zero or one
*	Zero or more
"n"	One or more (not known a priori)
n	*n* (where *n* > 1)
0..n	Zero to *n* (where *n* > 1)
m..n	m to *n* (where *n* > m)

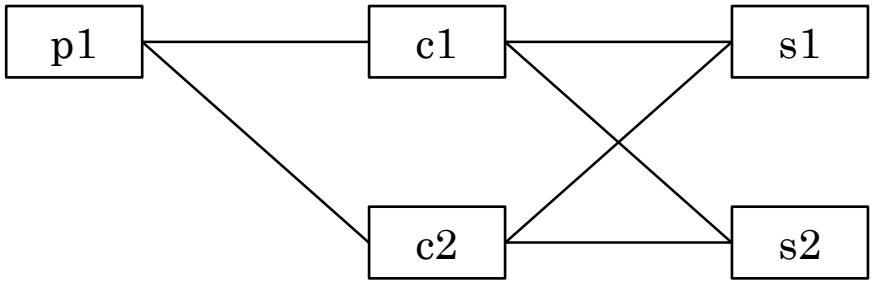
m, n denote integer values

"n" is the character n

Estensione: la molteplicità di default è *.

Object model

Professor Course Student swimlanes



Definite che cos'è un object model.

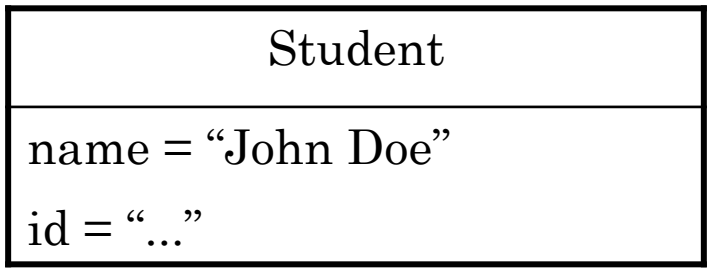
Struttura di oggetti basata su un class model.

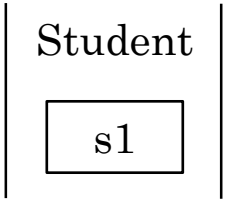
Il modello è valido?

Objects

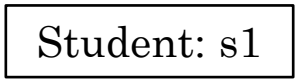
Classes represent individuals, also called instances or objects.

An object model shows a number of objects along with their attributes and associations.





Simplified representation: the type is the header of a swimlane



Simplified representation: type and label.

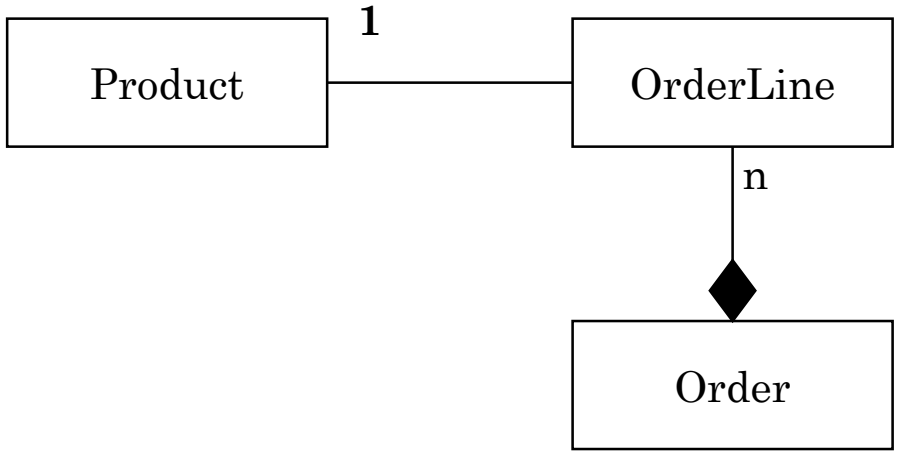
The label may be the value of an attribute.

Composition

Objects can be made up of other objects.

When a compound object is deleted, all its components are automatically deleted.

A component exists only in connection with a container.



An order is made up of order lines; each order line refers to a product and includes the number of units required.

Attributes:

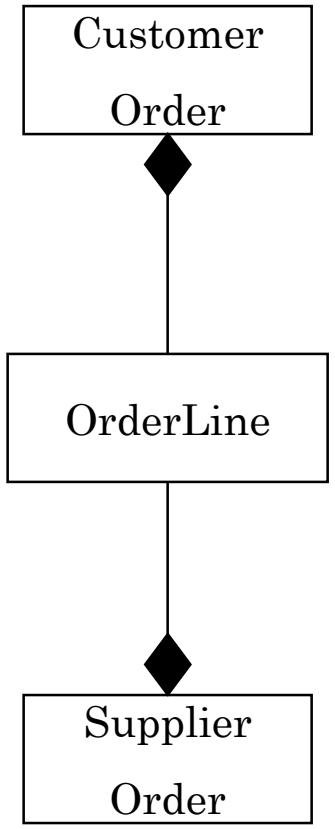
OrderLine: int n.

Qual è la molteplicità di OrderLine per Order?

La molteplicità di default di una composizione è n.

Composition

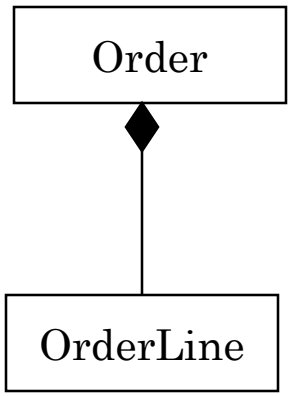
An orderLine may be part of a customerOrder or of a supplierOrder (but not of both).

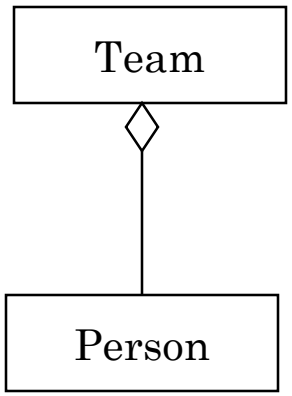


Aggregation

Composition is a strong link: a component belongs to only one container. The deletion of the container is cascaded to the parts.

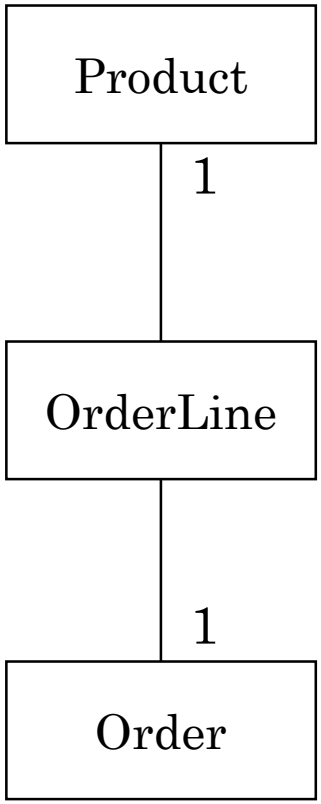
Aggregation denotes grouping and can be replaced by an associative relationship.



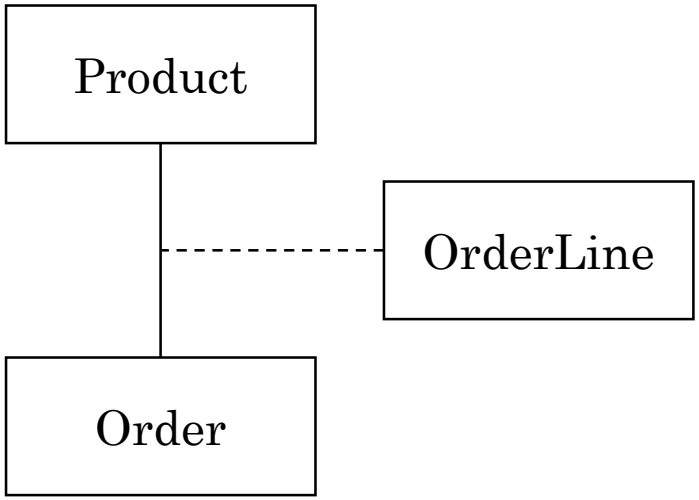


aggregation

Association class



associative pattern



OrderLine is an association class.

Association classes enable **attributes to be associated with relationships**.

Attributes: OrderLine: int n.

Inheritance

Vehicle Registration

A vehicle has exactly one owner that may be a person or a company.

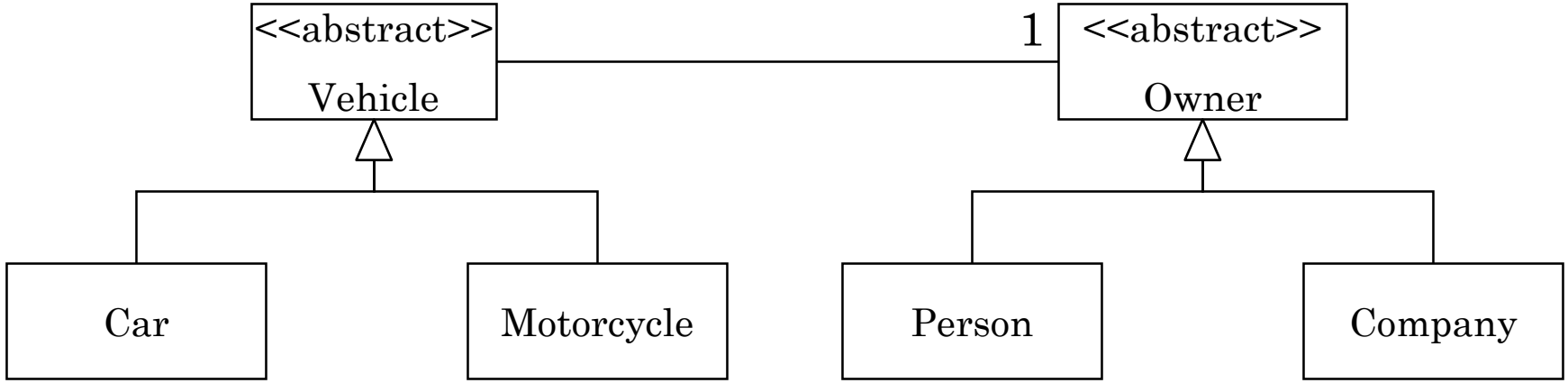
A vehicle may be a car or a motorcycle.

Vehicle and Owner are abstract classes (they cannot be instantiated).

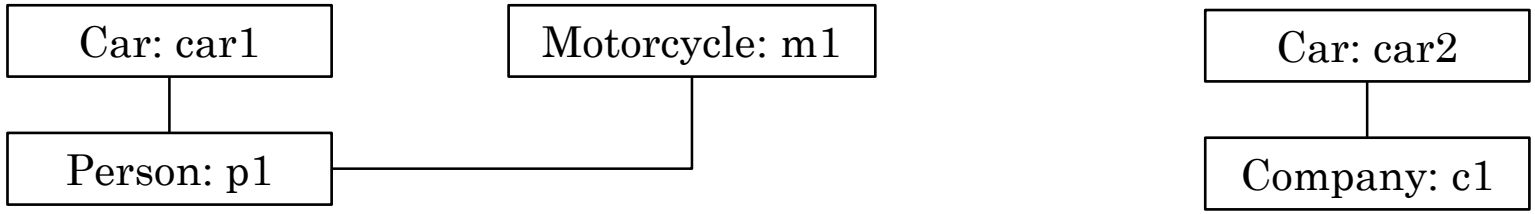
Inheritance

<<abstract>> is a *stereotype*

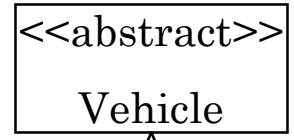
Class model



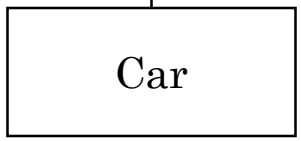
Object model



Inheritance

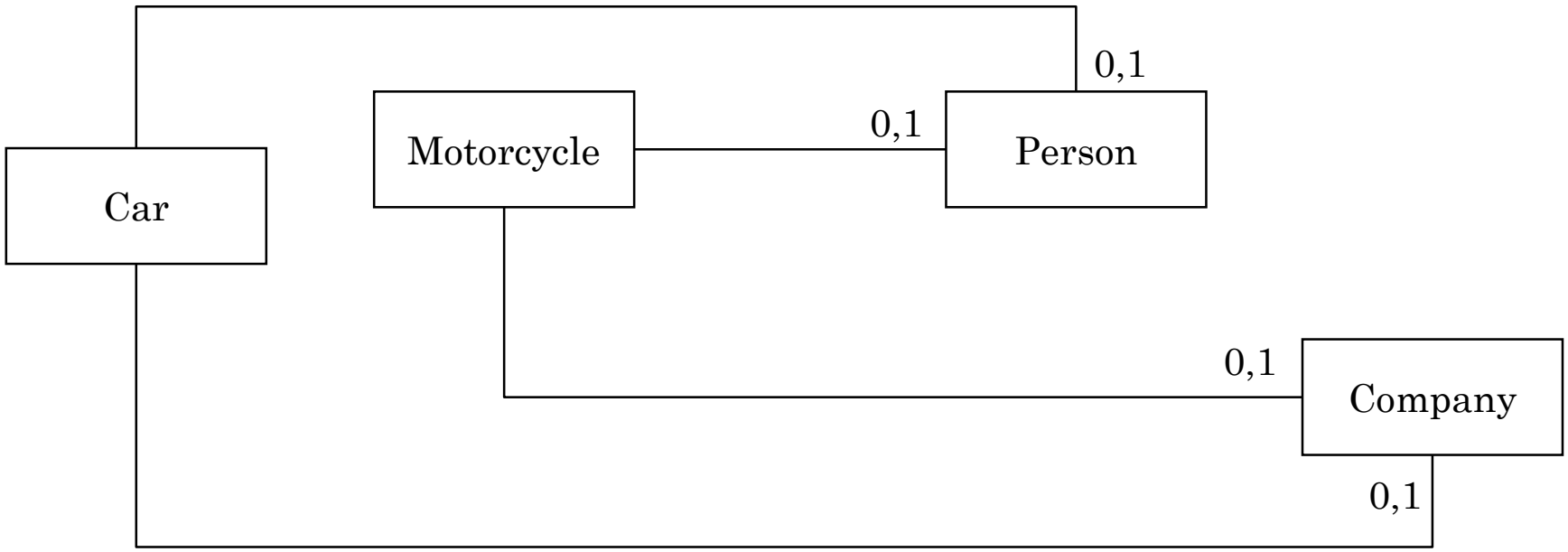


superClass



subClass

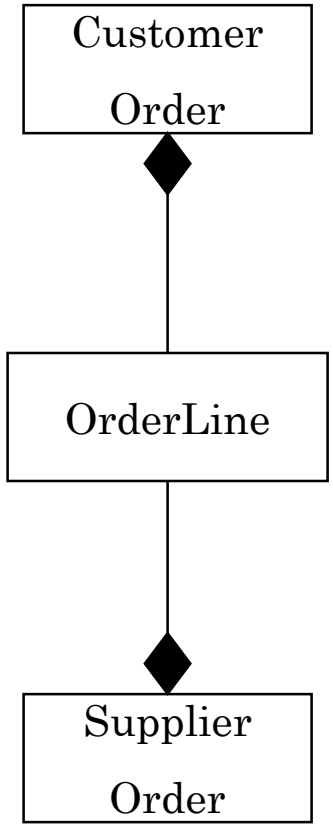
Without inheritance

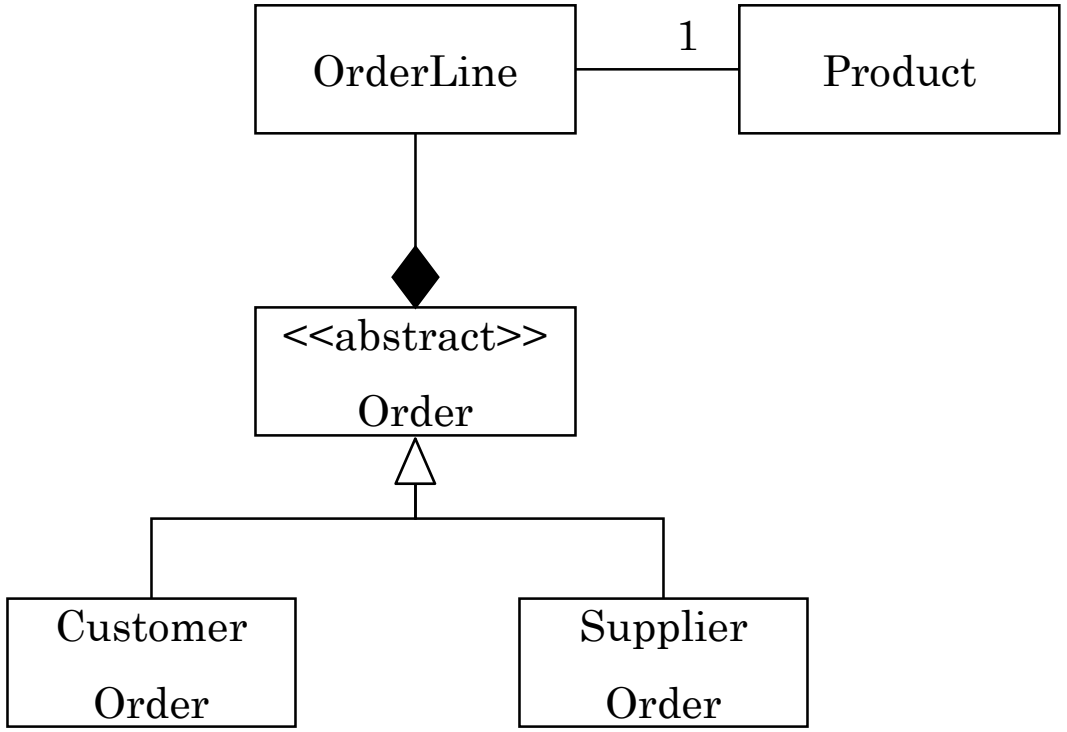


What are the problems?

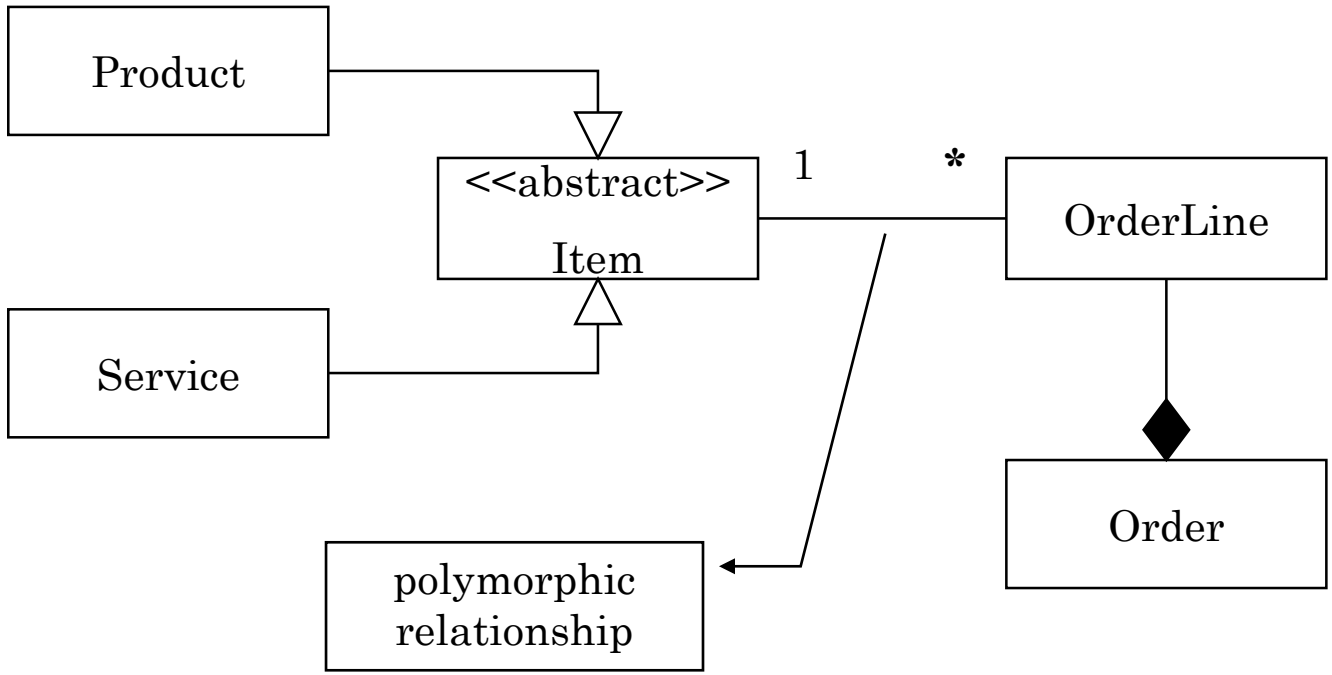
Inheritance and composition

An orderLine may be part of a customerOrder or of a supplierOrder (but not of both).





An order line refers to a product or a service



Product and Service inherit from Item; Item is an abstract class and hence it cannot be instantiated.

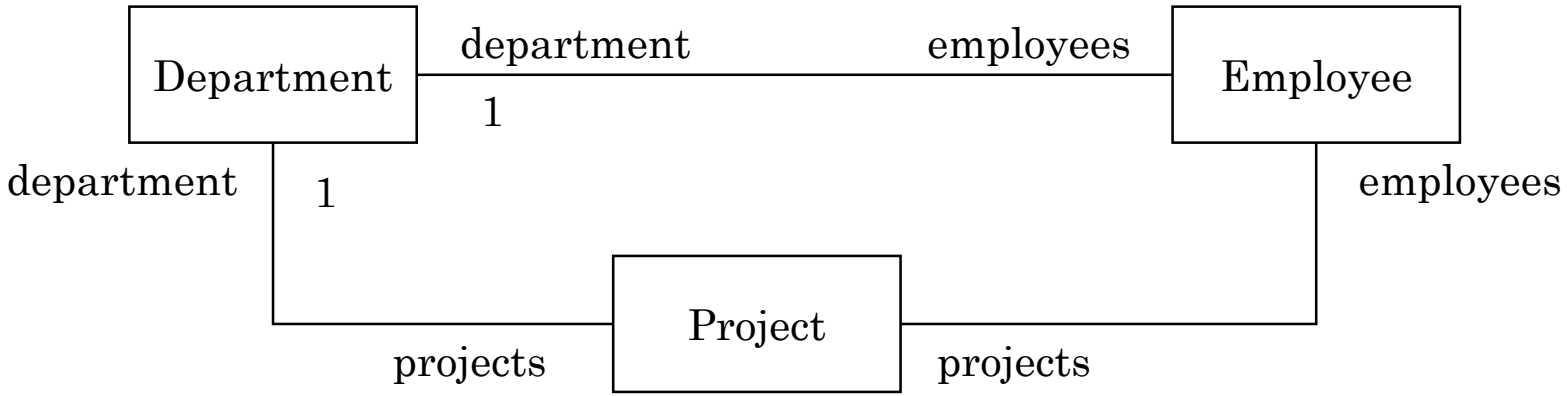
Associative attributes

A company is organized into departments. Each of its employees works in a department and may take part in a number of projects. Each project is managed by one department.

Each entity (department, employee and project) has a name; an employee has a position (String) as well.

Associative attributes

Associative attributes are references to other objects and are used in navigational constructs.



Attributes:

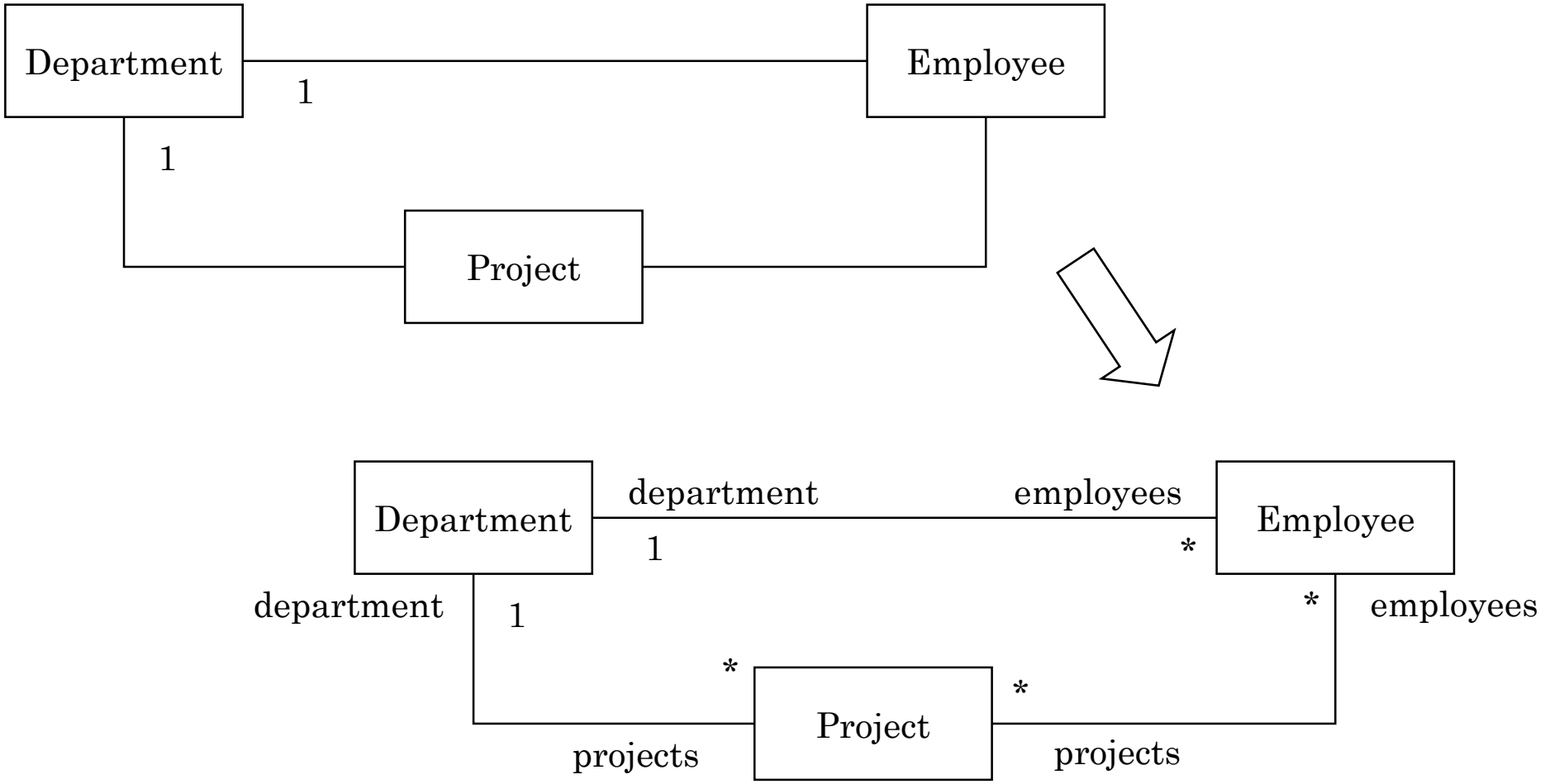
Department: String name.

Project: String name.

Employee: String name, String position.

Example: if p is a reference to a project, p.employees denotes the collection (set) of the employees participating in the project.

Associative attributes (default names)



When the associative attribute denotes a collection of partner entities, the **plural** of their class name with the initial in lower case is used, e.g. employees; otherwise the class name with the initial in lower case is used, e.g. department.

Strutture: alberi e grafi

Gli oggetti sono dello stesso tipo: nodi. I nodi hanno nomi distinti.

Nodo:

Attributi: String nome.

Invariante: nodi.nome distinct.

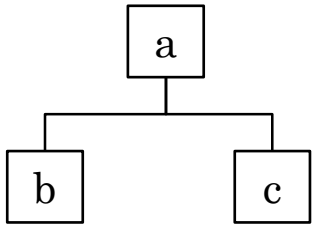
Dato un nodo n, n.predecessore indica il nodo che lo precede, n. successore indica il nodo che lo segue.

I nodi di un albero possono avere 0 o 1 predecessore e 0 o più successori.

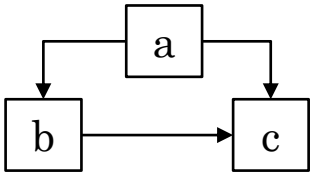
I nodi di un grafo possono avere 0 o più predecessori e 0 o più successori.

Esempi

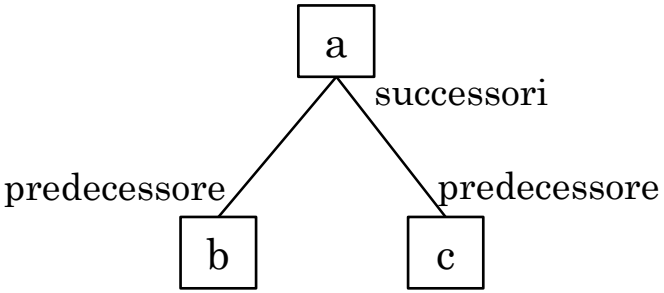
tree diagram

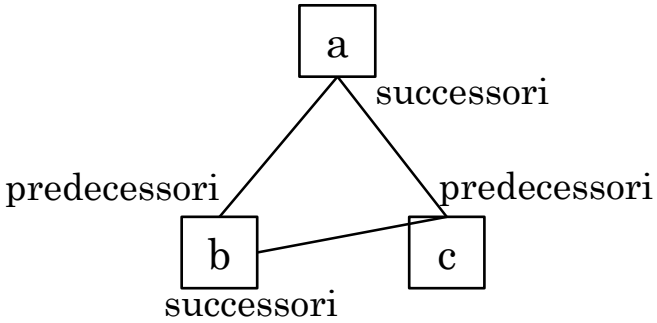


graph diagram



Object models



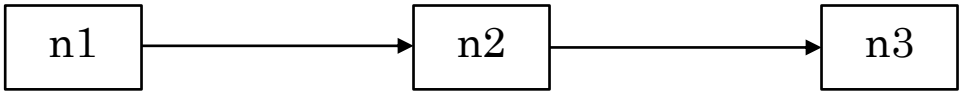


Sequenze

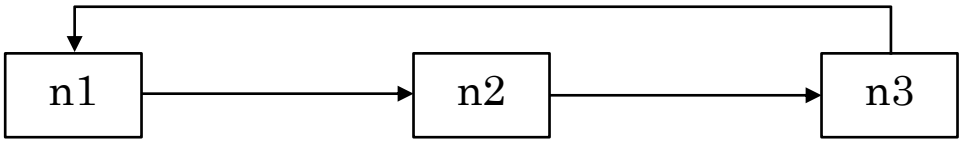
Una sequenza può avere un numero arbitrario di nodi, ma si escluda la possibilità di avere sequenze con un solo nodo; cioè non è possibile che un nodo non abbia né un predecessore né un successore (nodo isolato).

I nodi hanno nomi distinti.

Nell'esempio, n1 precede n2 e n2 segue n1.



sequenza aperta



sequenza chiusa

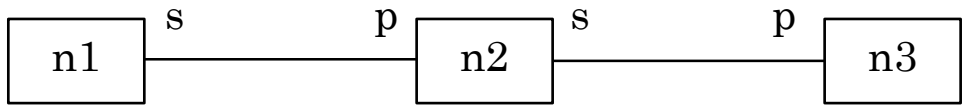
Object models

Attributo:

Nodo: String nome.

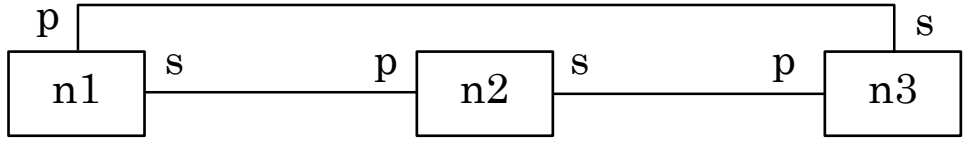
Invariante: nodi.nome distinct

Regola per evitare nodi isolati:
[nodo.s] == 1 and/or [nodo.p] ==1



sequenza aperta

[nodo.s] == 1 and [nodo.p] == 1



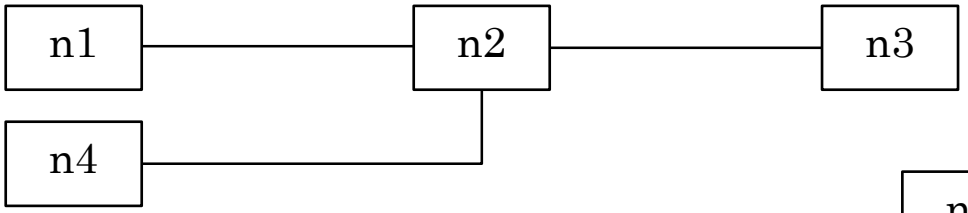
sequenza chiusa

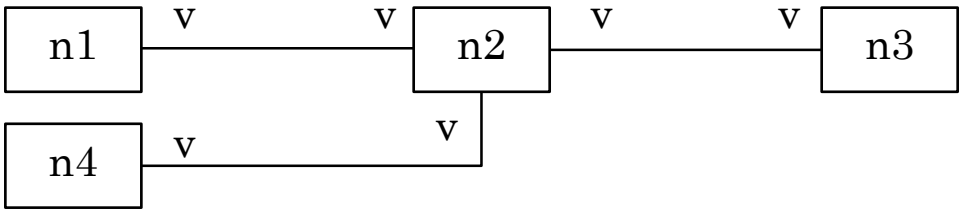
Nodi vicini

Si definisca un modello che rappresenti un grafo di nodi vicini.

Non è possibile che un nodo non abbia nessun vicino (nodo isolato).

I nodi hanno nomi distinti.





v = vicini

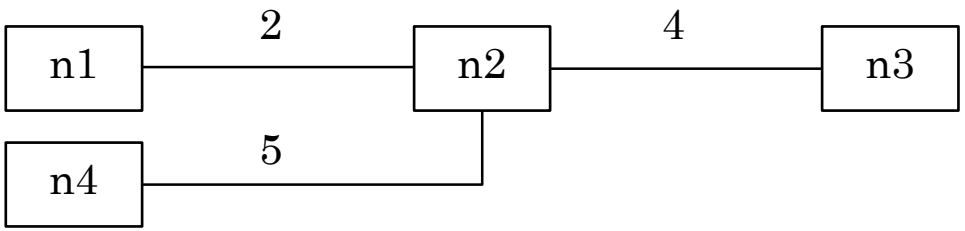
Attributo: Nodo: String nome.

Invariante: nodi.nome distinct

[nodo.v] > 0

Come si indicano i vicini di n1? n1.v. - n1.n2

Come si indicano i vicini di n2? n2.v. - n2.(n1, n3, n4)



Nodi con distanze

v = vicini sottintesi

Come si indica la distanza tra n1 e n2? n1-n2(2)

Come si indicano le distanze tra n2 e i suoi vicini? n2-n1(2),n3(4),n4(5)

Interpretations of class models

1. Programming-oriented perspective

The class model represents an object-oriented program (e.g. a java program): the model classes are mapped to java classes, the attributes of the model classes are mapped to the attributes of the java classes, inheritance relationships are mapped to “extend” relationships between java classes, and the other relationships are mapped to single references or collections in the java classes.

Interpretations of class models

2. Information-oriented perspective

According to the approach called *object-relational*, the class model represents persistent information that is stored in a relational database and accessed through java objects. Therefore two major outputs can be obtained from the class model: the definition of the relational tables and the collection of java classes whose objects are the representatives of the database records.

These objects are called *entities* and their classes are called *entity classes;* the class model (from this perspective) is called *entity model*.

A well known framework is **Hibernate**: it maps java classes to database tables, java data types to SQL data types, and provides an object-oriented query language.

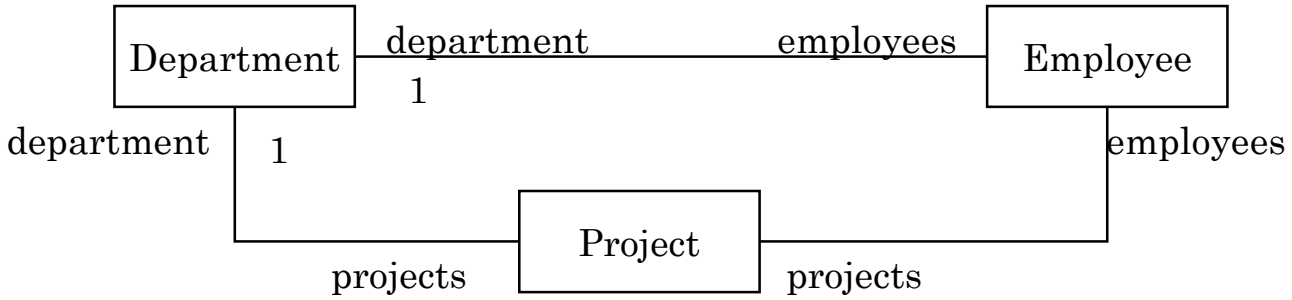
Operations on entities

There are six kinds of operations that can be performed on the entities in the information system. Such operations are referred to as entity operations.

Standard database operations (CRUD: *create, read, update and delete*).

Two additional operations work on associative attributes: adding an association and removing an association.

Associative attributes establish connections between entities; these connections can be concatenated so as to provide navigational paths.



Attributes: Department: String name. Project: String name.

Employee: String name, String position.

Espressioni navigazionali e condizioni

Sulla base di un modello classi-relazioni si possono definire dei **vincoli** che riguardano le operazioni relative al sistema rappresentato.

I vincoli sono basati su **espressioni navigazionali** che permettono di ottenere oggetti singoli o collezioni di oggetti a partire da un oggetto di partenza, seguendo gli attributi associativi (impliciti o espliciti) definiti nel modello.

Le condizioni sono espressioni booleane che spesso includono espressioni navigazionali. Sono usate per esprimere invarianti, regole di validazione, pre-condizioni e post-condizioni. Sono asserzioni sullo stato del sistema, quindi hanno un intento dichiarativo e non procedurale.

L'UML propone il linguaggio OCL: questo corso ne utilizza una versione semplificata detta SimpleOCL.

OCL

OCL (Object Constraint Language) is a UML formal language used to define additional constraints on the objects related to a class model.

(OMG standard, http://www.omg.org/spec/OCL/)

It can be used as a *query* language, to specify *invariants* on classes and *pre-conditions* and *post-conditions* on operations.

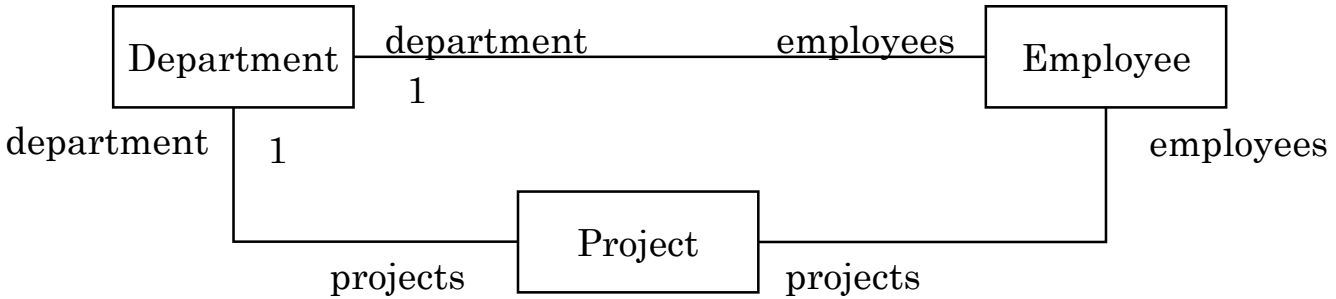
In general it can be used to express functional behavior declaratively.

OCL is a pure specification language; therefore, an OCL expression is guaranteed to be *without side effects*. When an OCL expression is evaluated, it simply returns a value.

Espressioni navigazionali

Attributes

Project: int budget.



Dato l'impiegato **e** si ottengano

- i suoi progetti: **e.projects** (1 step di navigazione)
- i dipartimenti dei suoi progetti: **e.projects.department** (2 step)
- il n. dei suoi progetti: **[e.projects]**
- il nome del suo dipartimento: e.department.name

Nota: il risultato di una navigazione può essere un oggetto, un valore, una collezione di oggetti o di valori (di solito un set per eliminare i duplicati).

Il numero di elementi di una collezione si ottiene con l'operatore **[]**.

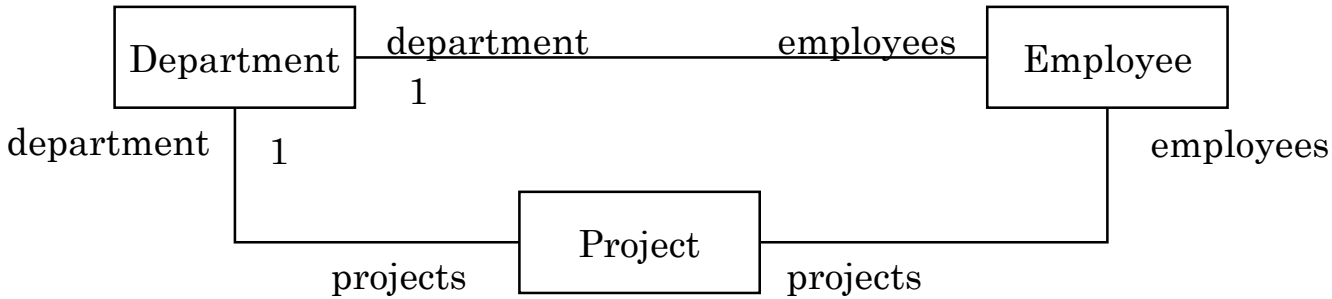
Navigazione

Concatenazione di attributi associativi in base alle relazioni del modello.

Con partenza da un impiegato, ad es. e

e.projects.department

e.department.projects



Filtri

Dato l'impiegato **e** si ottengano

i suoi progetti con budget > 1000: e.projects (budget > 1000)

Forma completa

e.projects (p, p.budget > 1000)

dove **p** è un cursore che indica l'oggetto corrente (un project).

Data la collezione c, l'espressione c (...)

dà la collezione c **priva degli oggetti** che non soddisfano il **predicato** contenuto tra le parentesi ().

Max, min, sum, average

Dato l'impiegato **e** si ottengano

- il suo progetto con budget massimo (minimo): **e.projects.max(budget**) oppure min;
- il budget massimo dei suoi progetti: **e.projects.max(budget).budget** oppure **e.projects.budget.max**
- la somma e la media dei budget dei suoi progetti: e.projects.sum(budget), e.projects.average(budget), oppure e.projects.budget.sum, e.projects.budget.average.

Note: data la collezione di oggetti c

c.max(a) oppure min, sum, average danno l'oggetto che ha il valore massimo o minimo nell'attributo *a*, oppure la somma o la media dei valori contenuti nell'attributo *a* degli oggetti della collezione.

Se c denota una collezione di valori, c.max dà il valore massimo; analogamente con min, sum, average.

Altri esempi

La collezione di tutti gli oggetti della stessa classe si indica con il nome della classe al plurale (e l'iniziale minuscola): es. **employees**, projects.

Indicare la somma dei budget di tutti i progetti:

projects.sum(budget) oppure projects.budget.sum.

Indicare l'impiegato per il quale la somma dei budget dei suoi progetti risulti massima:

employees.max(e, e.projects.sum(budget))

dove e è un cursore che indica l'oggetto corrente (un employee).

L'espressione e.projects.sum(budget) dà la somma dei budget dei progetti a cui lavora l'impiegato corrente.

Indicare l'impiegato per il quale il n. di dipartimenti diversi dal suo e relativi ai suoi progetti risulti massimo:

employees.max(e, [e.projects.department (d, d != e.department and distinct)])

Nota: e, d sono cursori.

Entità contestuali

Dato il dipartimento d e il progetto p fornire gli impiegati di d che lavorano al progetto p.

with Department d, Project p: d.employees (e, e.projects include p)

il predicato *projects include p* si applica a tutti gli impiegati della collezione d.employees; se il valore è false l'impiegato è escluso dal risultato. Il termine projects è un attributo associativo dell'impiegato.

La clausola *with* introduce le entità contestuali scelte dall'esecutore dell'operazione; si indica il tipo e il nome del riferimento all'entità.

Invarianti

Sono condizioni che valgono sempre, ad esempio:

Il n. massimo di progetti per impiegato è 3:

[employee.projects] <= 3.

Il nome della classe con l'iniziale minuscola significa ogni elemento della classe e ha quindi valenza di quantificatore universale.

Tutti i progetti di un impiegato sono gestiti dal suo dipartimento:

employee.projects in employee.department.projects

oppure

[employee.projects (p, p.department != employee.department)] == 0

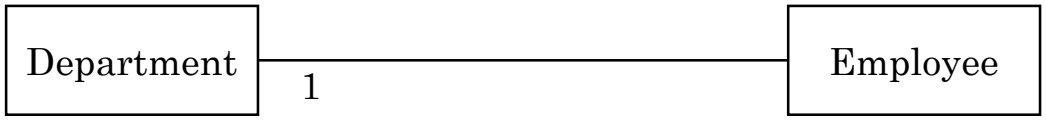
cioè il n. progetti gestiti da un altro dip. è 0.

Per tutti gli x vale il predicato p equivale a

il numero degli x per i quali non vale p è 0.

Invarianti sugli attributi associativi

Un impiegato deve appartenere ad un dipartimento.



Si può usare un invariante: employee.department def.

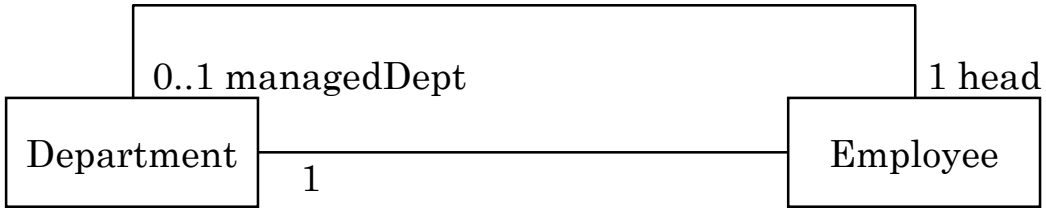
L'operatore **def** indica che l'attributo è definito.

Oppure si può segnare la relazione Employee-Department come necessaria per Employee, sottolineando la molteplicità.

Implicazione:

quando si inserisce un nuovo impiegato, lo si collega ad un dipartimento.

Regole di validazione



Ogni dipartimento ha un direttore che deve appartenere al dipartimento.

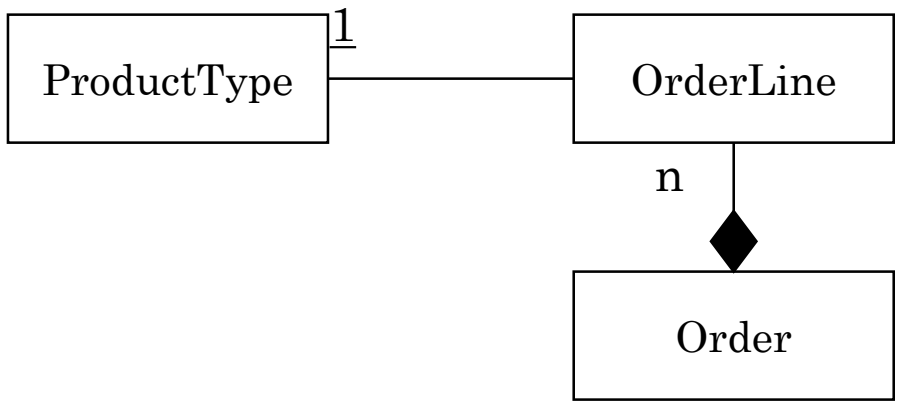
Si tratta di una *regola di validazione* (il dip. è inserito nel sistema prima dei suoi impiegati) che si verifica su richiesta.

Regola di validazione

department.head in department.employees

Operator distinct

The product types associated with the lines of an order must be distinct.

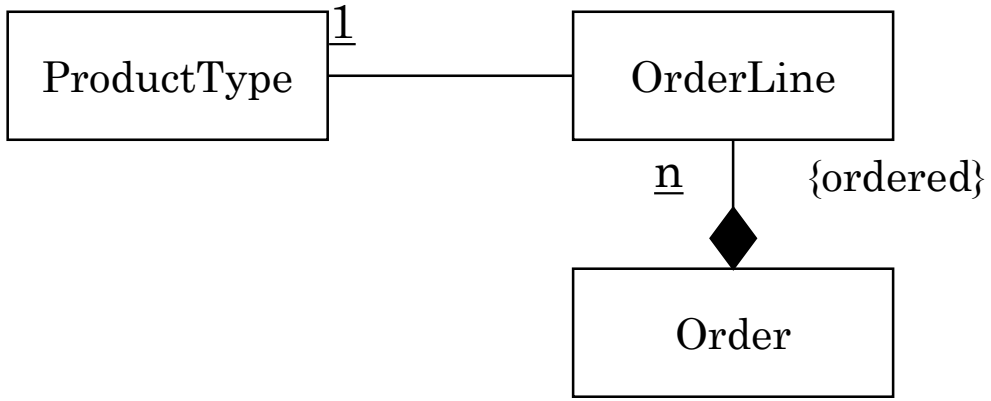


Invariant

order.orderLines.ProductType **distinct**.

Ordered associations

Lines are ordered.

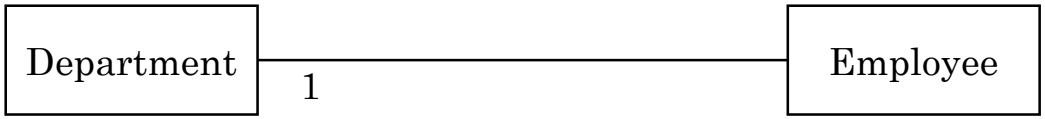


order.orderLines(0) denotes the first line of the order.

Operazioni

Le operazioni sono trattate in modo dichiarativo: i vincoli si esprimono mediante una **pre-condizione** e il risultato mediante una **post-condizione**.

Es. inserimento di un impiegato in un dipartimento purché non si superi il max di 10.



enrollment:

with Department department // scelta del dipartimento

pre: [department.employees] < 10 // vincolo sulla scelta

post: new Employee e, e.department == department.

Come risultato il sistema contiene un nuovo impiegato collegato al dipartimento scelto. La relazione è bidirezionale.

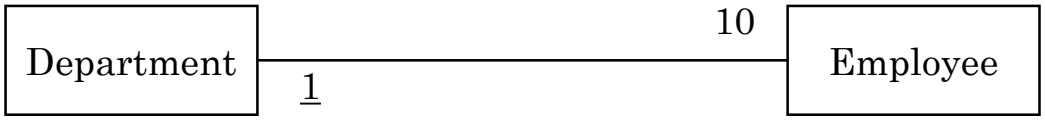
Oppure

with Department department, [department.employees] < 10 // with con vincoli

post: new Employee e, e.department == department.

Operazioni: semplificazione

Es. inserimento di un impiegato in un dipartimento purché non si superi il max di 10.



enrollment:

Si può omettere la precondizione data la molteplicità 10.

with Department department // scelta del dipartimento

post: new Employee e

Dato l'attributo associativo necessario department, l'inserimento di un nuovo employee comporta l'associazione con un dipartimento. Il dipartimento è quello indicato con la clausola with.

Required attributes and derived ones

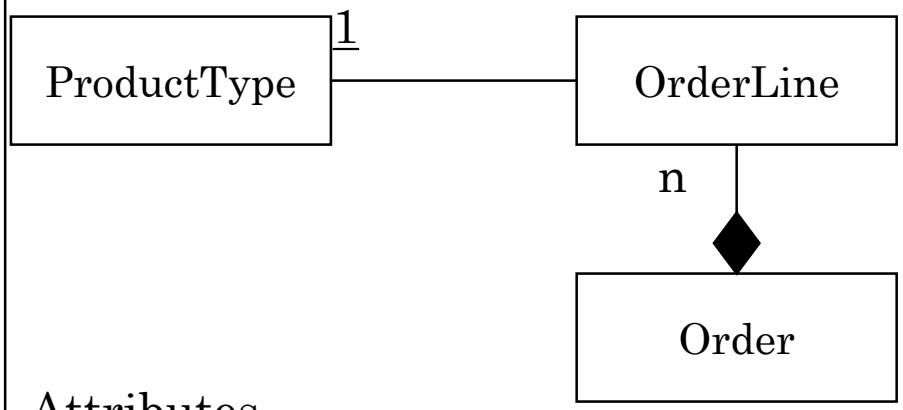
Required attributes must be *initialized* at instantiation time. They may be ordinary attributes or associative ones.

For example, when a product type is introduced in the system its description and price have to be defined.

A derived attribute is obtained, when needed, from ordinary and/or derived attributes.

Required attributes and derived ones

An order is made up of a number of order lines; each order line refers to a product type and contains the number of units required. Product types include descriptions and prices. What is the amount of an order?



Required attributes are underlined.

Derived attributes return a value, an object or a collection.

Class OrderLine has the derived attribute *amount*, as well as class Order.

Attributes

ProductType: String description, float price.

OrderLine: int n.

Derived attributes

OrderLine: float amount = n * productType.price.

Order: float amount = orderLines.sum(amount);

Derived relationships

A derived relationship is a compact way to express derived associative attributes.

Order – ProductType = Order – OrderLine – ProductType

Equivalent to the introduction of the following derived attributes:

Order: Set<ProductType> **productTypes** = orderLines.productType

ProductType: Set<Order> **orders** = orderLines.order.

Requirements definition

Software Requirement

Something that is wanted or needed (Webster's Dictionary)

Functional requirements

A functional requirement can be about a service (or an activity) that transforms input data into output data. The aim is to satisfy a customer need.

The focus is on what has to be done, not on how it is done.

The customer may be any actor involved in the process including the end user.

Functional requirements often include **structural** requirements.

Constraints (or **non functional** requirements)

related to performance, compliance to standards, conventions and regulations, ...

Quality issues

Requirements should be unambiguous, complete, consistent and verifiable.

Unambiguous requirement: only one interpretation.

Consistent requirement: no conflicts with the other requirements.

Verifiable requirement: its fulfillment can be verified (with tests or other means) in a cost-effective way.

Complete requirements: they handle all the relevant issues.

Traceability of requirements

forward: from a requirement to the subsystem(s) fulfilling it.

backward: from subsystems to requirements.

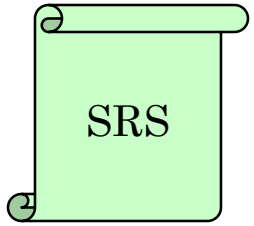
Requirements analysis

Elicitation: getting needs out of *stakeholders* (all the people having an interest in the system to be developed).

Analysis: making sure that requirements meet quality criteria.

Documentation: producing documents and diagrams on the basis of current standards.

Guide for Software Requirements Specification (IEEE STD-830)



1. Introduction
   ....
2. General description
   ...

3.1 **Functional requirements**
   3.1.1 Functional requirement 1
   3.1.1.1 Introduction
   3.1.1.2 **Inputs**
   3.1.1.3 **Processing**
   3.1.1.4 **Outputs**
   ...

3.2 **External interface requirements**
   3.2.1 User interfaces
   3.2.2 Hardware interfaces
   3.2.3 Software interfaces
   3.2.4 Communications interfaces
3.3 **Performance requirements**
3.4 **Design constraints**
   3.4.1 Standards compliance
   3.4.2 Hardware limitations
   ....

3.5 **Attributes**
   3.5.1 Availability
   3.5.2 Security ...

Comment: functional requirements are stated in terms of inputs and outputs and the transformation needed to produce the outputs from the inputs.

Model-driven development di un'applicazione

Dati i requisiti funzionali di un'applicazione (es. prestito di libri di una biblioteca),

1. si ricava il **domain model** (classi relazioni UML) con attributi e regole (invarianti);
2. si modellano le attività degli utenti con **use cases** UML.
3. i casi d'uso rappresentano task ai quali sono associate precondizioni per i vincoli e postcondizioni per gli effetti.

Requirements for managing book loans in a library 1/2

The library contains *books*: book attributes include id, title, authors, publisher. Books are added by librarians (staff).

Members are the persons entitled to get books on *loan*. Members are enrolled by librarians: the enrollment date and the librarian who made the enrollment must be recorded. Member attributes include name, address, email. A personal id is provided by the enrollment procedure.

A librarian has a name and an id.

To borrow a book, members must open a loan. First, they get a list of books by specifying subject and authors, then they choose an item from the list. The loan is opened if the book is available and the number of open loans of the member does not exceed the limit (6). The duration of the loan is 14 days.

Members cannot borrow additional books if they have some loans overdue.

When a member returns a book, the system closes the loan associated with the book id.

Requirements for managing book loans in a library 2/2

A loan has two dates: the *end date* and the *due date*. The end date is set when the loan is closed.

An *ongoing* (*past*) loan has the end date undefined (defined).

When a loan is one week *overdue*, the system sends a *reminder* to the borrower of the book.

The major states of a loan are ongoing, past, overdue.

Members can update personal information such as address and email.

Librarians can make inquiries on ongoing and past loans.

Functional requirements:

Librarian add books, enroll members and make inquiries.
Members borrow books, return books, and update personal information.
The system sends reminders to borrowers.

Rules

The loan is opened if the book is available and the number of open loans of the member does not exceed the limit (6). The duration of the loan is 14 days.

Members cannot borrow additional books if they have some loans overdue.

When a loan is one week overdue, the system sends a reminder to the borrower of the book.

Terms to be clarified: loan, ongoing loan, past loan, overdue loan.

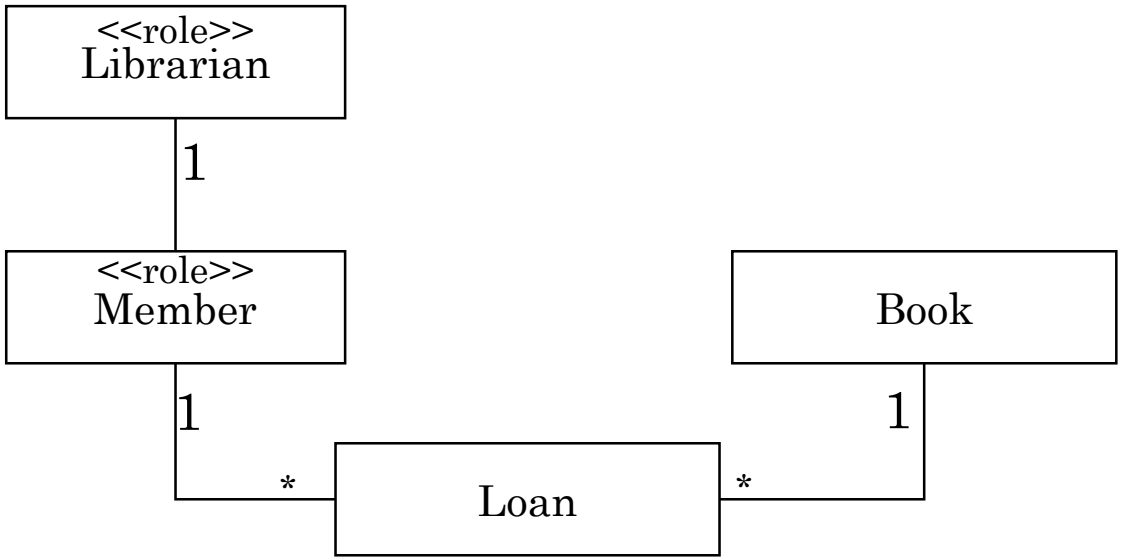
Rules define *constraints on activities* (e.g. borrowBook, returnBook, sendRemainder).

Rules may be expressed by invariants or preconditions (and validation rules).

The number of ongoing loans for each member must be <= 6

[member.loans (ongoing)] <= 6. // invariant: *member* stands for each member

Domain model



* means 0 or many and is the default multiplicity

Attributes

Book: String id, String title, String authors, String publisher.

Member: String id, String name, String address, String email, Date enrolmentDate.

Loan: Date dueDate, Date endDate.

Librarian: : String id, String name.

Comments

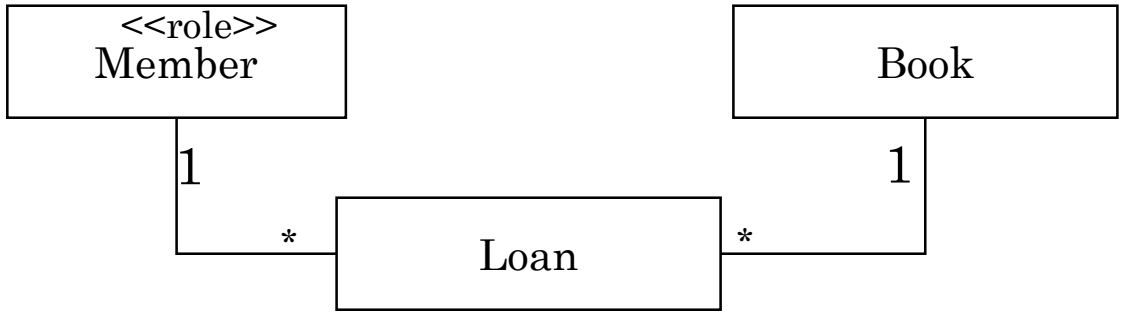
Users are represented by role classes: librarian, member. A stereotype (<<role>>) may be added.

Stereotipi

Lo stereotipo role vuole indicare che la classe Member rappresenta persone con un determinato ruolo.

Gli stereotipi si possono raggruppare in un profilo UML.

Loan



Loan: Date dueDate, Date endDate.

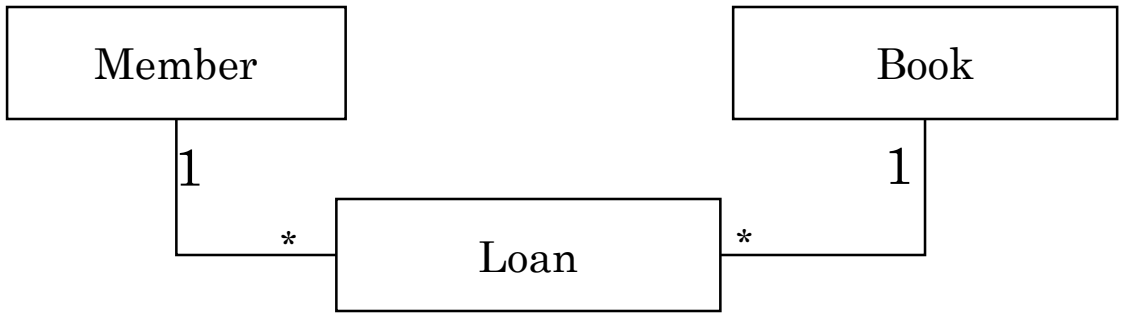
Loan is an association class between Member and Book. When a loan is generated, its dueDate attribute is set, while the endDate is null.

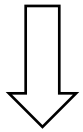
An ongoing loan is a loan whose endDate is null.

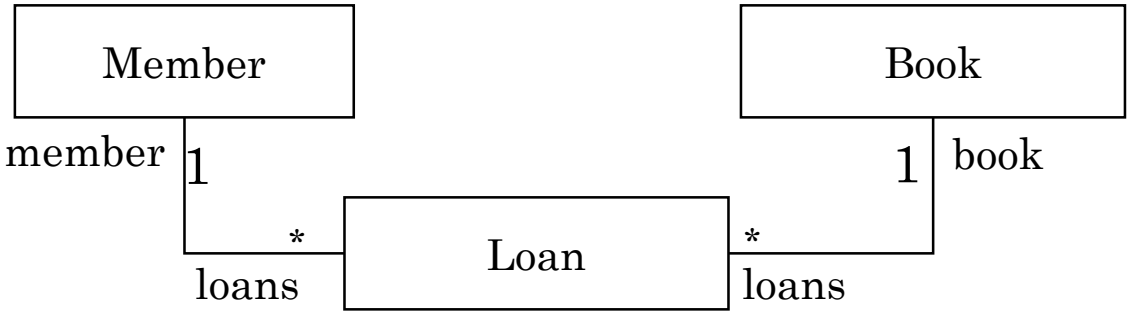
A past loan is a loan whose endDate is not null.

A loan is overdue if the endDate is null and the dueDate is before today.

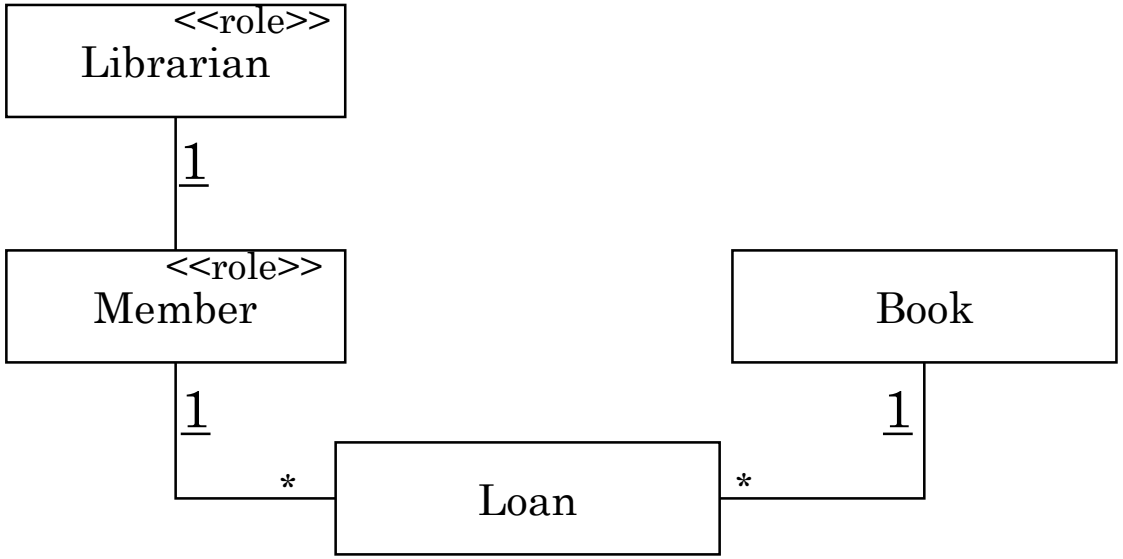
Standard associative attributes







Domain model with required attributes



Attributes

Member: String id, String name, String address, String email, Date enrolmentDate.

Book: String id, String title, String authors, String publisher.

Loan: Date dueDate, Date endDate.

Librarian: String id, String name.

Commento

Gli attributi necessari sono definiti quando l'entità è generata; ciò vale anche per gli attributi associativi. Quando è generata un'entità Member è inserito il collegamento all'entità Librarian che rappresenta il bibliotecario che ha registrato il nuovo iscritto.

Derived attributes

A derived attribute is obtained, when needed, from ordinary and/or derived attributes.

A book is available if there is no ongoing loan associated with it.

Derived attributes

Loan:

boolean *ongoing* = endDate == null;

boolean *past* = endDate != null;

boolean *overdue* = ongoing and dueDate < today; // dueDate before today

Book:

boolean *available* = [loans (ongoing)] == 0.

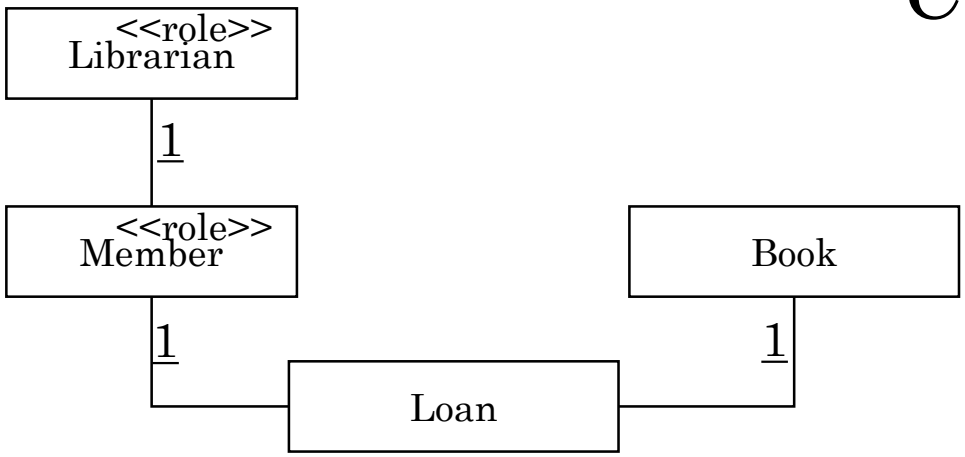
Notation:

loans denotes the list of loans associated with a book,

today indicates the current date when the derived attribute is computed.

[] returns the number of elements in a collection.

Complete domain model



Attributes

Member: String id, String name, String address, String email, Date enrolmentDate.

Book: String id, String title, String authors, String publisher.

Loan: Date dueDate, Date endDate. Librarian: String id, String name.

Derived attributes

Loan: boolean *ongoing* = endDate == null;

boolean *past* = endDate != null;

boolean *overdue* = ongoing and dueDate before today;

Book: boolean *available* = [loans (ongoing)] == 0; Loan currentLoan = loans (ongoing).

Invariants: [member.loans (ongoing)] <= 6.

Note

Book: boolean *available* = [loans (ongoing)] == 0;

Loan currentLoan = loans (ongoing).

L'attributo derivato currentLoan denota il prestito corrente, cioè l'unico tra i prestiti del libro che ha l'attributo ongoing true. Può essere null. Dà un'eccezione se loans (ongoing) ha più di un elemento.

Use cases

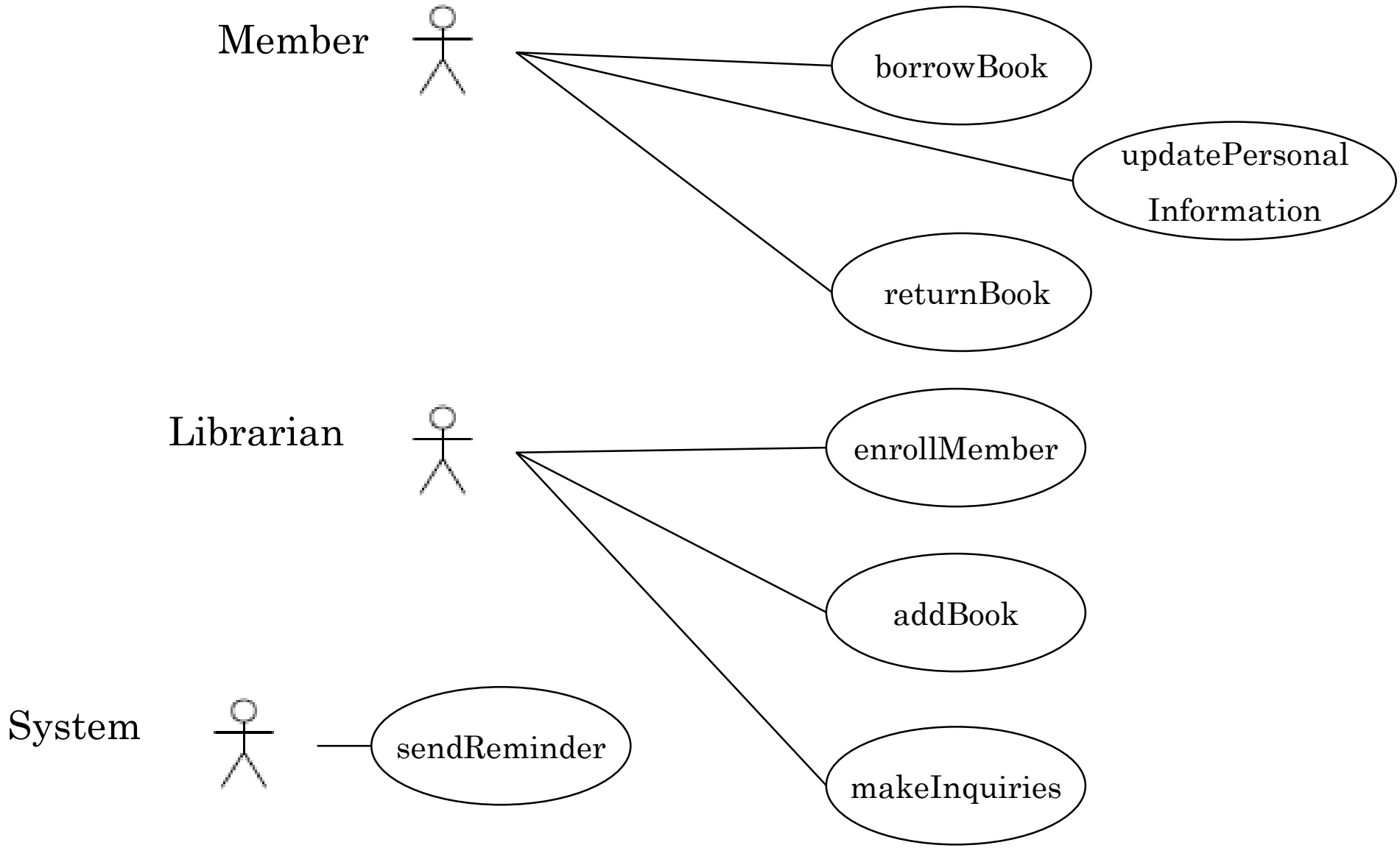
Un modello use cases presenta le attività degli utenti del sistema.

Gli use cases contengono descrizioni delle interazioni tra l'utente e il server.

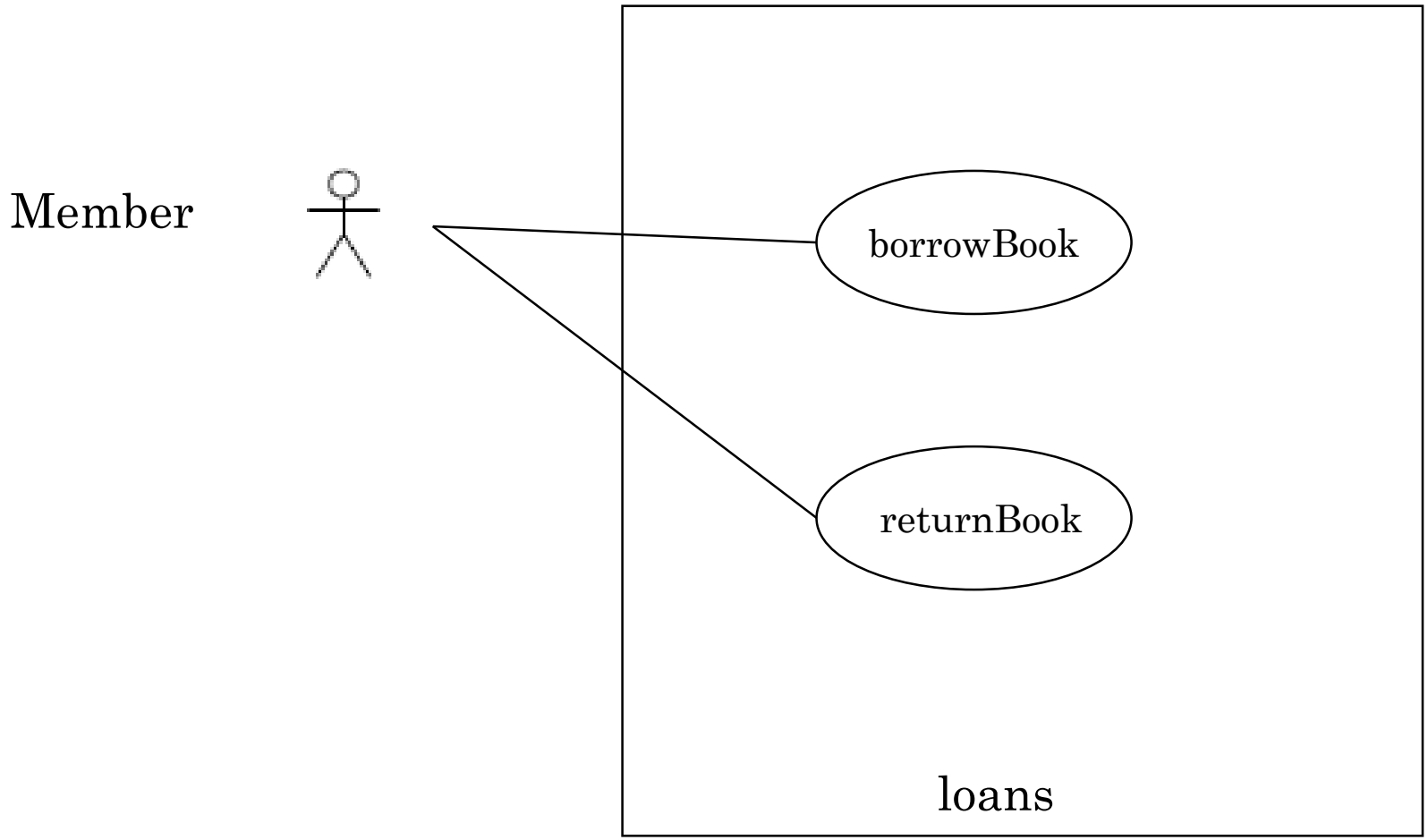
Possono essere raggruppati in subjects.

Inoltre tra gli use cases possono esservi relazioni di inclusione (include) e relazioni di estensione (extend).

Use cases della biblioteca

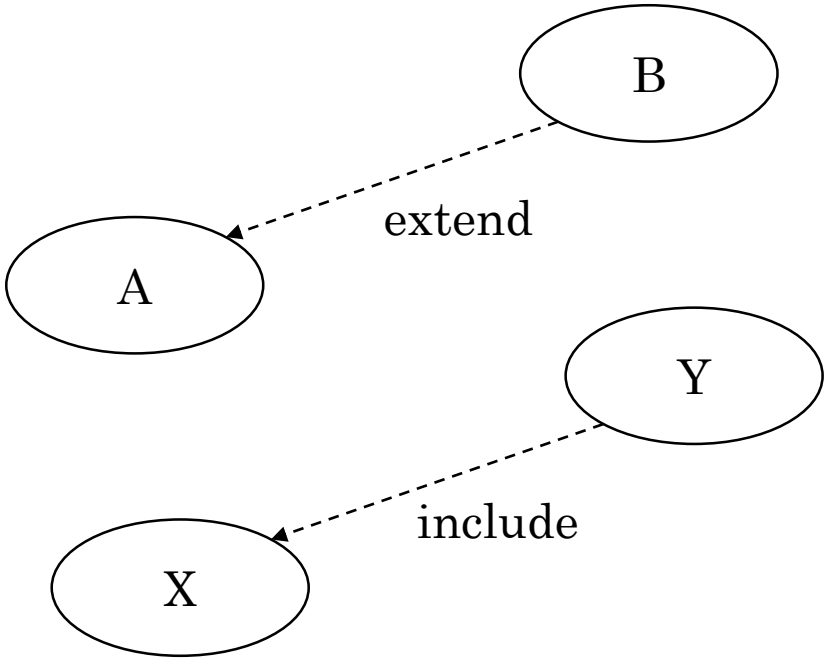


Subjects



Use cases may be grouped in subjects.

Relationships between use cases



extend: the behavior of A can be extended by the behavior of B at the extension points specified in A. A is meaningful independently of B.

For example, A may be extended by an *optional help use case*.

include: the behavior of X is included in the behavior of Y. Y is incomplete without X.

Description of borrowBook

To borrow a book, members must open a loan. First, they get a list of books by specifying subject and authors, then they choose an item from the list.

The following scenario may be defined

1. User inputs search parameters (topic and authors).
2. System displays a list of available books.
3. User chooses a book.
4. System validates the user is eligible to get the book.
5. System opens a new loan.

What is the right structure of a scenario?

The scenario rewritten

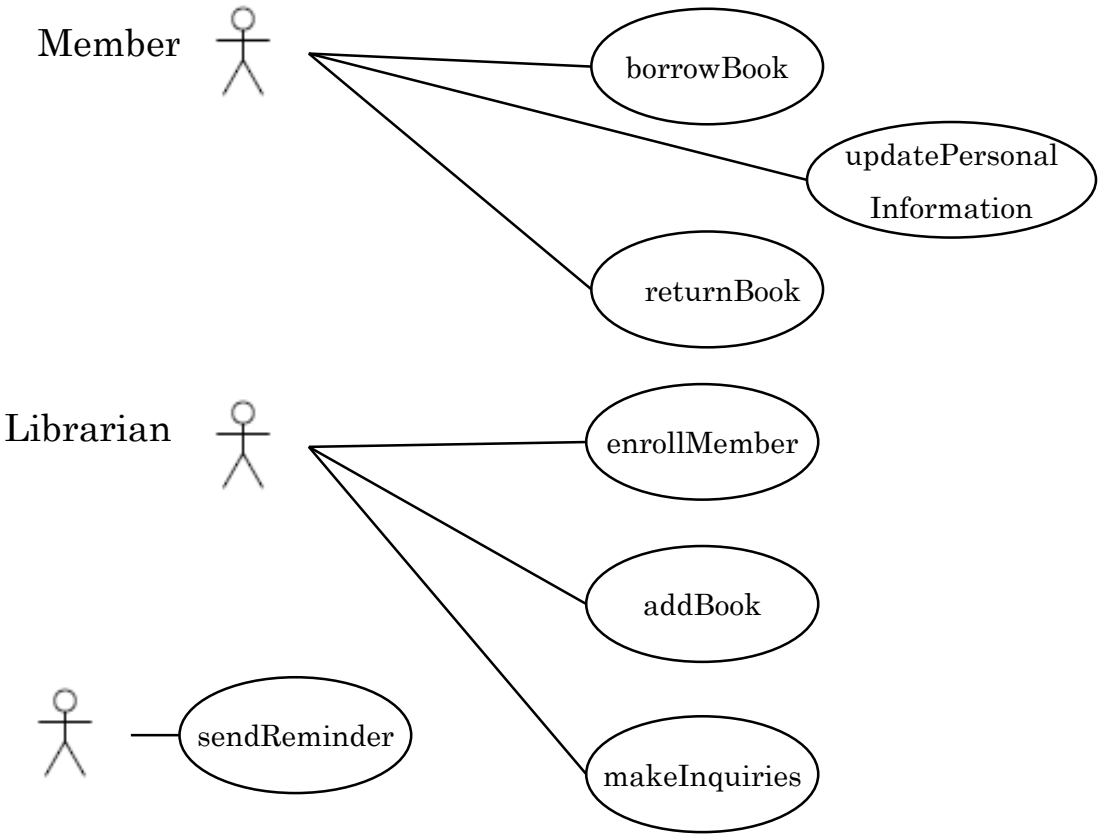
1. User inputs search parameters (title and authors).
2. System displays a list of available books.
3. User chooses a book.
4. If all constraints are met, System opens a new loan and the use case ends; otherwise User is sent back to step 1.

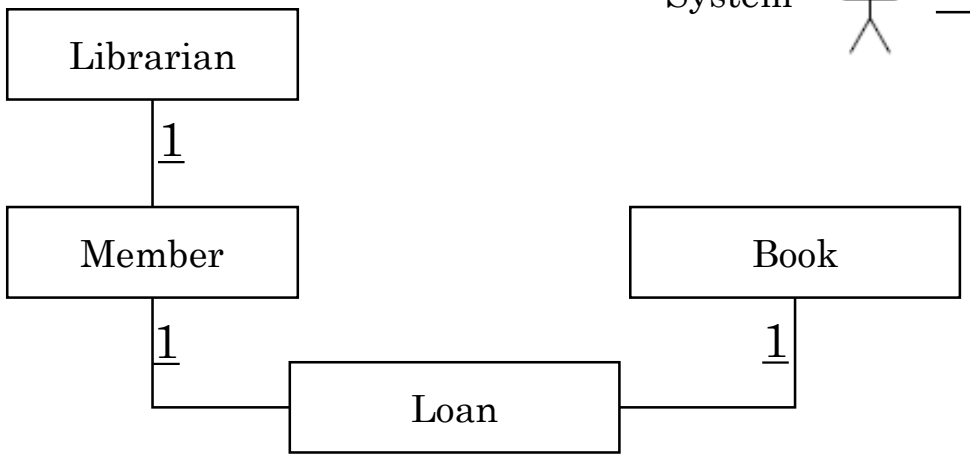
Flow chart written in structured natural language.

Focus on the "happy" path; error situations may be added.

Integrazione tra modello informativo e casi d'uso.

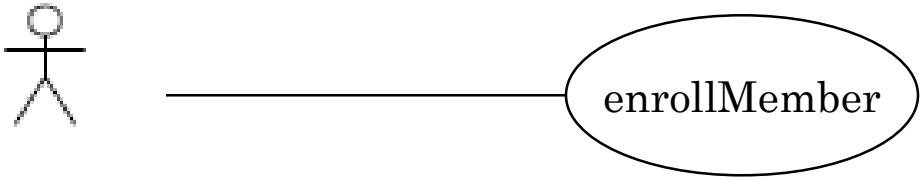
Definizione delle precondizioni e postcondizioni dei casi d'uso.





Modello informativo + attributi (non mostrati per mancanza di spazio)

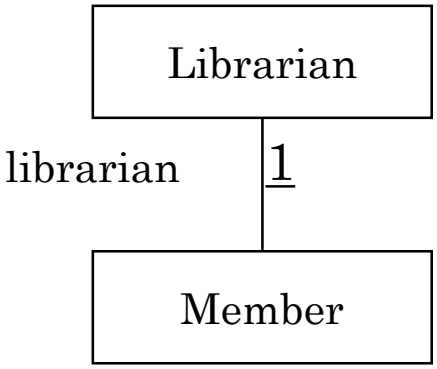
Use case enrollMember



post: **new** Member.

There is a new member after the completion of the use case.

The effect is to add a new member entity to the information system with all its required attributes set to appropriate values. In particular, the associative attribute librarian points to the librarian entity representing the librarian who performed the use case.



Member: String id, String name, String address, String email, Date enrolmentDate.

Note

Dati gli attributi (informativi e associativi) necessari,

post: new Member

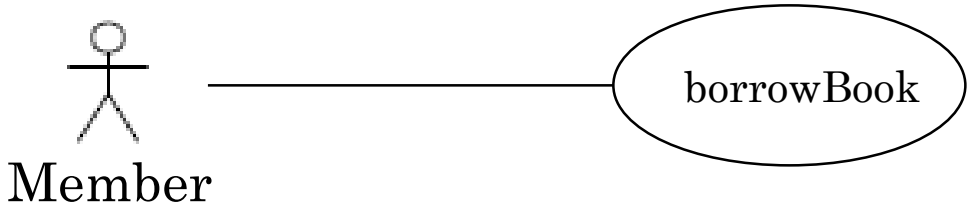
significa che è aggiunto un nuovo member con tutti gli attributi informativi necessari valorizzati e con il collegamento al librarian che ha eseguito il caso d'uso.

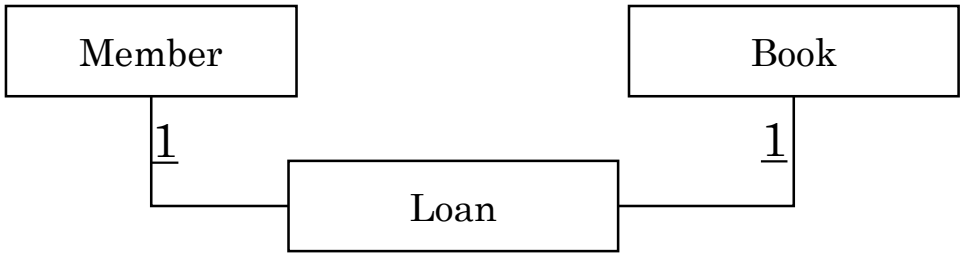
post esplicita: new Member m, m.id def, m.name def, m.address def, m.email def, m.enrolmentDate def, m.librarian == librarian.

Nella post esplicita **librarian** indica l'esecutore del caso d'uso: si usa il nome del ruolo con l'iniziale minuscola.

def significa defined (definito)

Use case borrowBook



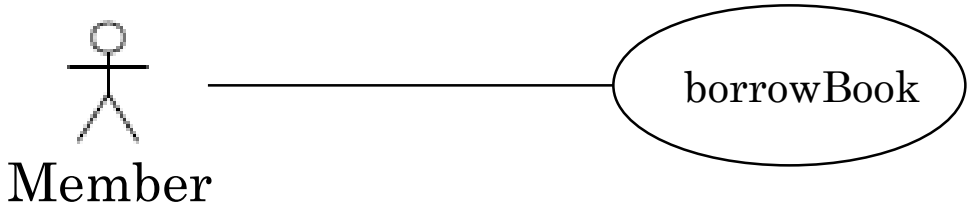


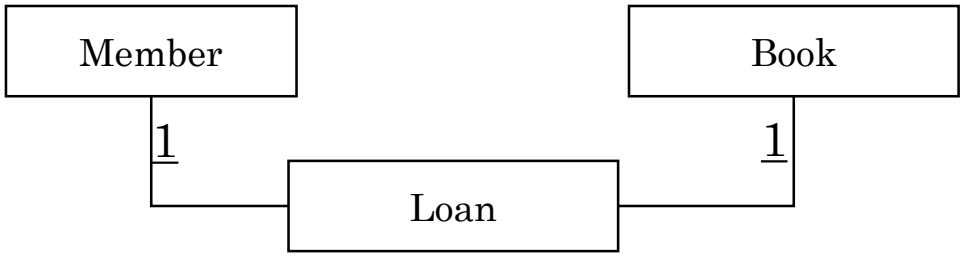
Il risultato è un nuovo prestito (loan) collegato all'esecutore (un member) e al libro che ha scelto durante l'esecuzione del caso d'uso.

Occorre accertare che l'esecutore non abbia dei prestiti scaduti e che il libro sia disponibile; inoltre occorre definire la scadenza del prestito.

L'invariante garantisce che l'esecutore non abbia già 6 libri in prestito.

Use case borrowBook





with book // il libro scelto

pre: [member.loans(overdue)] == 0 and book.available

post: new Loan l, l.dueDate == today + 14 days // def della dueDate

post esplicita: new Loan l, l.dueDate == today + 14 days, l.member == member, loan.book == book.

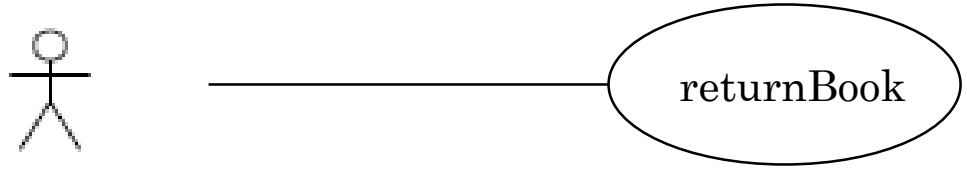
Use case borrowBook

Complete precondition (without the invariant)

pre: [member.loans(overdue)] == 0 and [member.loans (ongoing)] <= 5

and book.available

Use case returnBook



Member

L'effetto è di chiudere il prestito relativo al libro (che deve essere in prestito all'esecutore del caso d'uso)

with book

pre: book in member.loans(ongoing).book

post: book.currentLoan.endDate == today.

oppure

with book, book in member.loans(ongoing).book

post: book.currentLoan.endDate == today.

Specify the remaining use cases

updatePersonalInformation: address **modified** and/or email **modified**

addBook: new Book

Remark: **modified** means that the attribute has been modified.

sendReminder: to be performed daily // timing constraint

post (informal): a reminder has been sent to

loans (l, l.overdue and today == l.dueDate + 1 week).member.

Nota: il sollecito è mandato una volta sola, il giorno in cui è trascorsa una settimana dalla scadenza, ed è mandato agli utenti dei prestiti scaduti da una settimana.

Variazioni sulla biblioteca

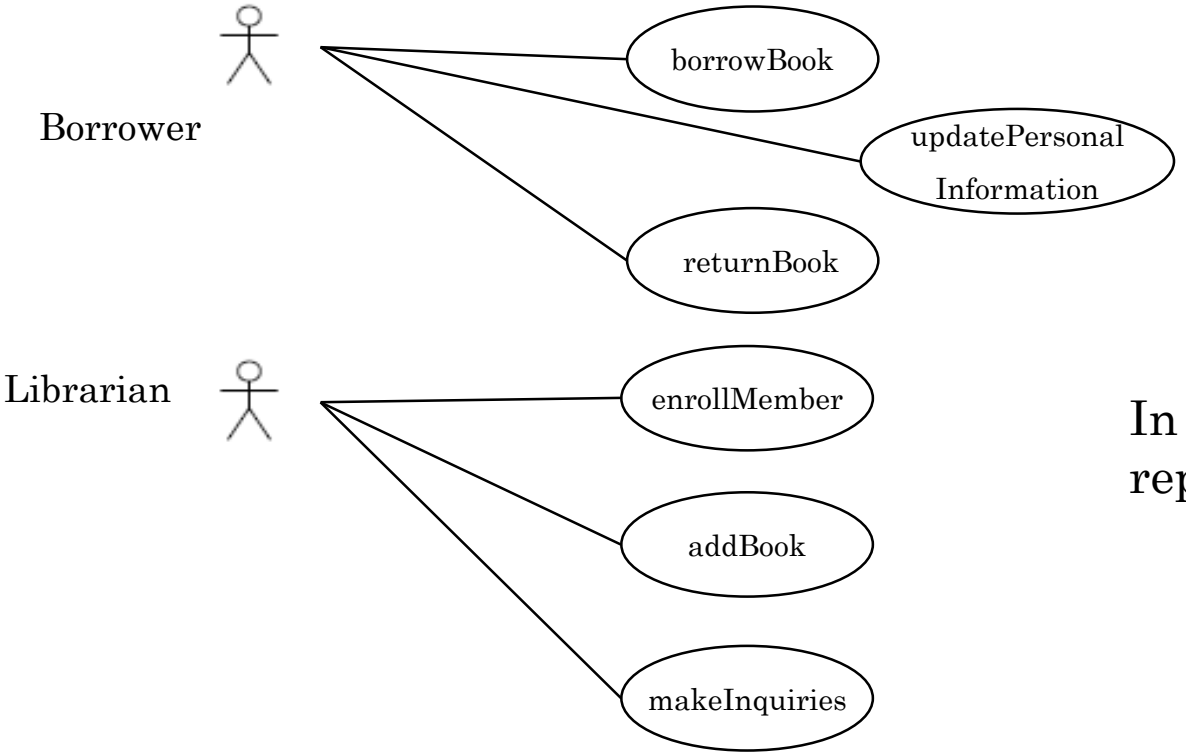
I bibliotecari possono prendere libri in prestito?

Sì e non serve la registrazione.

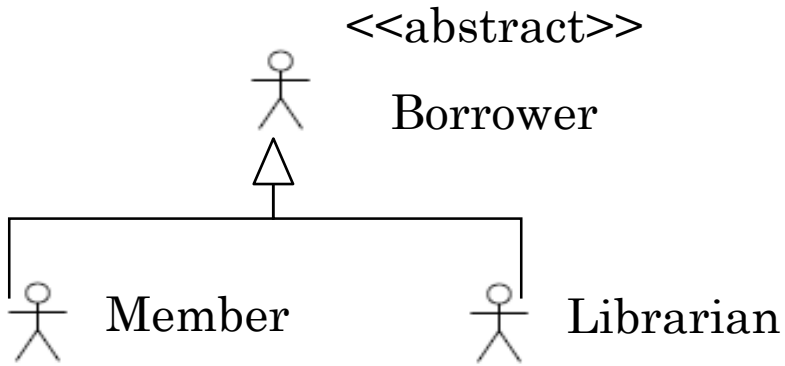
Come appare il nuovo modello di uses cases?

→ Borrower entity type

Use cases

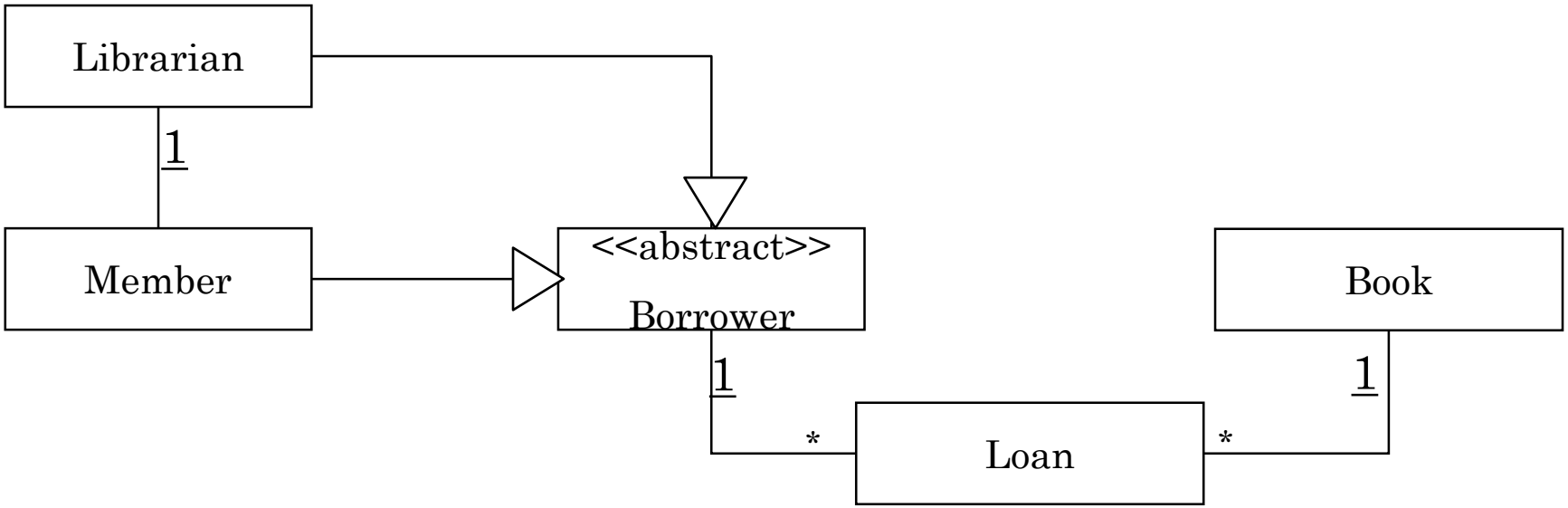


In the definitions of the use cases replace Member with Borrower.



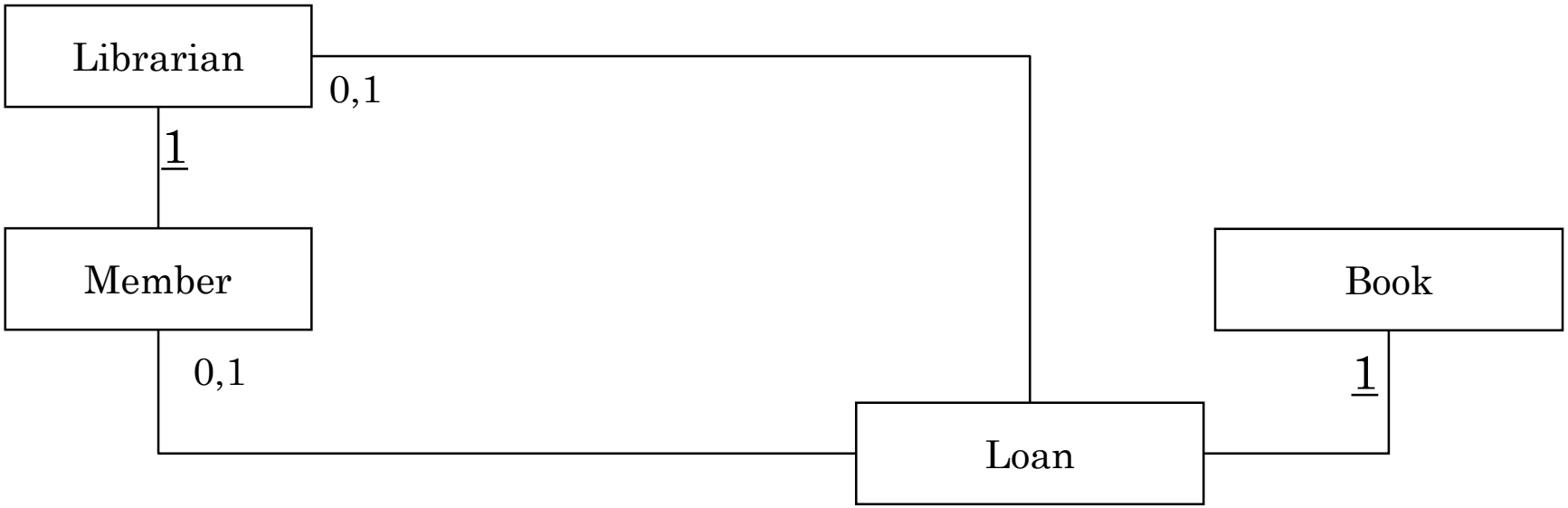
La relazione di inheritance è automaticamente definita nei modelli use cases come in tutti i modelli UML.

The revised domain model



Perché serve introdurre Borrower?

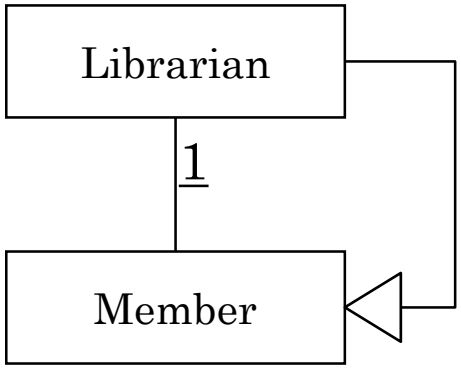
Simplified domain model



La classe Loan potrebbe essere collegata sia a Member sia a Librarian, ma un prestito è in carico ad un solo utente.

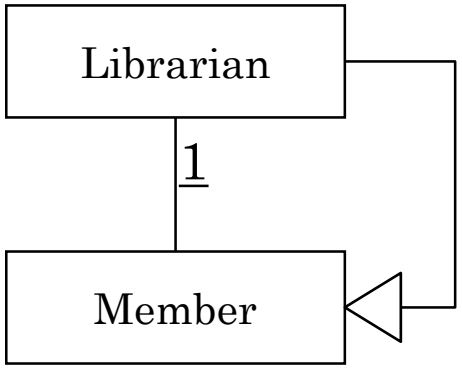
invariante: [loan.member] + [loan.librarian] == 1.

Variante



Questa soluzione è accettabile?

Variante



Questa soluzione è accettabile?

No, perché se un librarian fosse un member avrebbe un collegamento al librarian che l'ha iscritto, ma un librarian non necessita di alcuna iscrizione.

Business processes

Process (from Oxford Advanced Learner's Dictionary): a series of actions or tasks performed in order to do, make or achieve something.

In a business context, a (business) process (BP) is *a structured ordering of work steps across time and place (Davenport)*, to be carried out by users playing specific roles or by automated services.

UML activity diagrams

Un'attività è un flow chart con vari tipi di nodi: action nodes, control nodes e object nodes. In questi modelli le attività sono sinonimi di processi.

Gli *action nodes* corrispondono ai task (unità di lavoro) e possono essere automatici o umani.

La determinazione degli esecutori dei task umani non è trattata nello standard. Negli esempi, i task umani sono accompagnati dai ruoli degli esecutori.

La precedenza dei task dipende dal control flow che è definito mediante *control nodes* (tra cui scelte, confluenze, fork, join)

Il data flow è subordinato al control flow; i dati possono essere rappresentati mediante *object nodes*.

I processi sono gestiti da un process engine. Il process engine ha il compito di istanziare ed eseguire i processi.

Control flow and data flow

Control flow: stabilisce i vincoli di precedenza tra i task. Quando un task è completato, il process engine attiva i task che lo seguono, tenendo conto degli eventuali control nodes.

Se un task è automatico, è eseguito direttamente mediante un servizio. Se un task è umano, è assegnato ad un esecutore (o partecipante) sulla base del ruolo associato al task.

Data flow: gli input di un task sono gli output di uno o più task precedenti. Il data flow mostra le dipendenze dei task dai dati.

Esempio di processo

Il processo **HandleClientOrder** permette ad un fornitore di trattare gli ordini provenienti dai clienti.

Quando il fornitore riceve un ordine da un cliente, il process engine genera un'*istanza* del processo e le passa come parametro di input l'ordine del cliente.

Un ordine ha una descrizione e un importo.

Se l'importo è <= 1000 €, l'ordine è accettato automaticamente altrimenti è valutato dall'account manager associato al cliente.

L'ordine ha uno stato che può essere accepted o rejected.

Il processo informa il cliente dello stato dell'ordine.

Nel sistema informativo del processo sono registrati i clienti e gli account manager (staff). Le relazioni con un cliente sono gestite da un accountMgr che può trattare vari clienti.

Information model del fornitore

ClientOrder:

String description; double amount; state (accepted, rejected);

boolean highAmount = amount > 1000.

Activity diagram

Questo processo ha un parametro di input di tipo ClientOrder. Il parametro diventa una variabile d'istanza che è accessibile agli altri nodi.

control nodes

initial node

final node

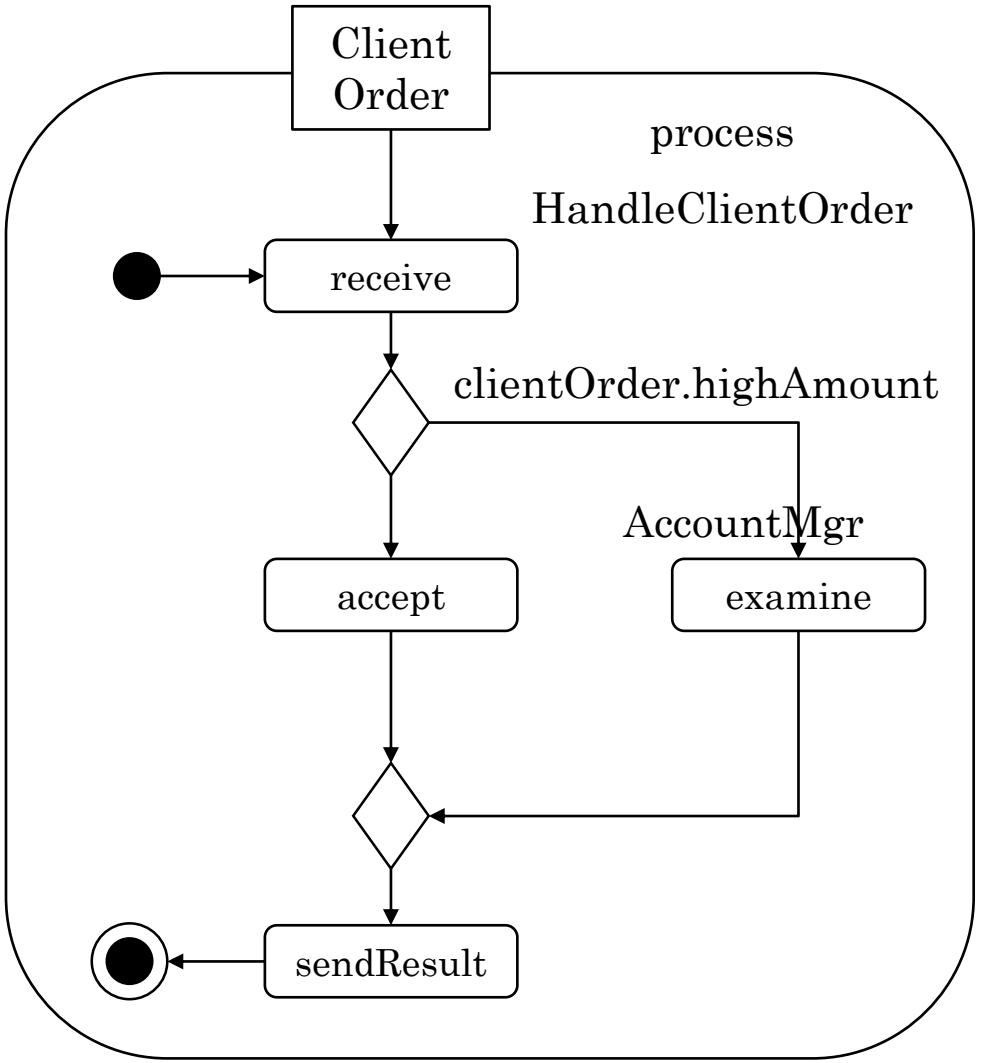
decision/merge node

action node (task)

name

object node

Client Order



var: ClientOrder clientOrder.

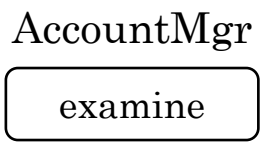
Post-condizioni

receive: this.clientOrder == clientOrder;

// copia il parametro nel riferimento locale

accept: clientOrder.state == accepted.

examine: clientOrder.state def.

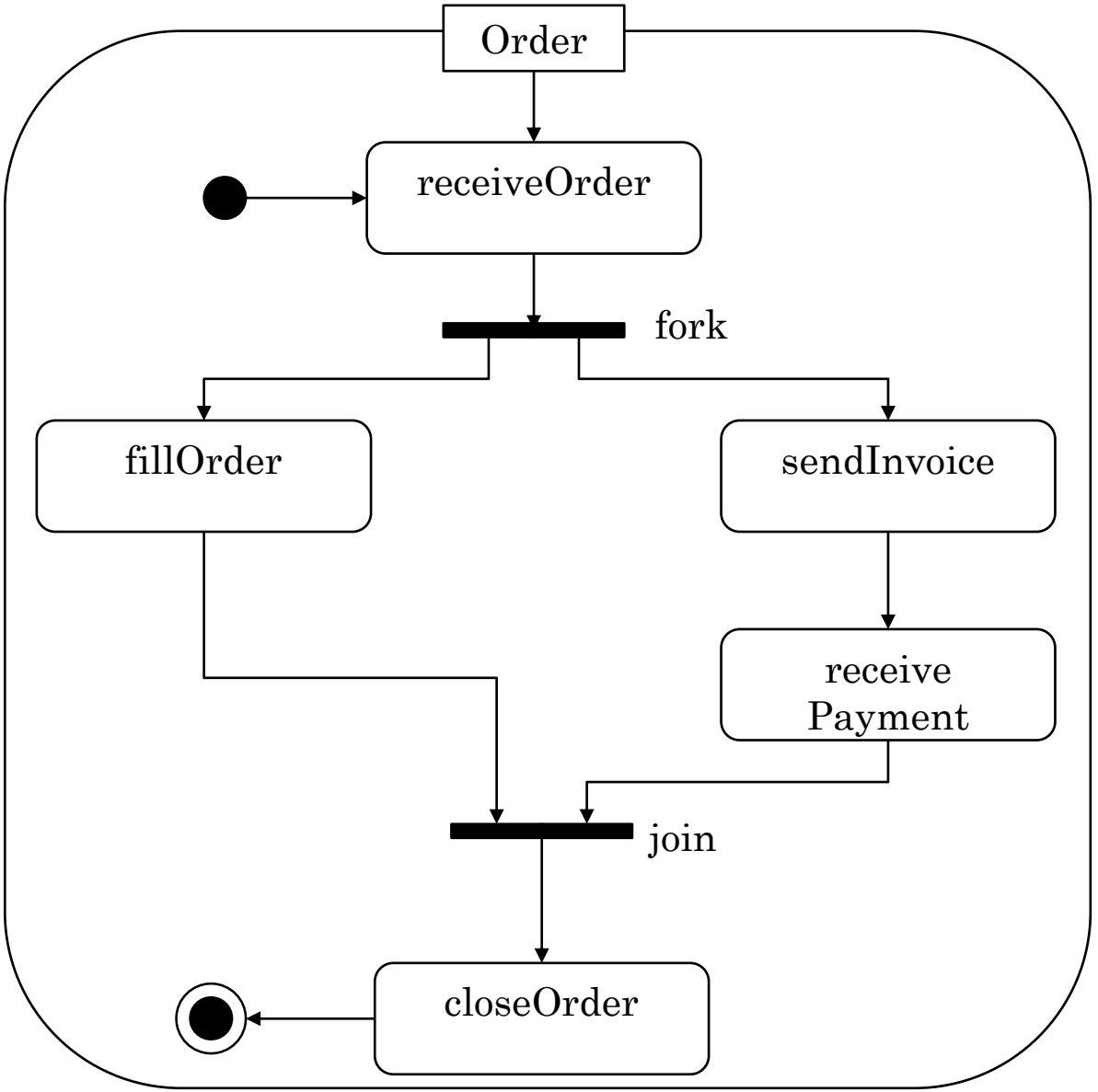


Al task examine è associato il ruolo AccountMgr.

Come si determina l'esecutore del task examine?

Con la regola: performer = clientOrder.client.accountMgr.

Parallel control-flow constructs: fork and join



Two parallel paths

Dataflow models

The process is represented as a network of activities producing output data from input data (functional requirements).

The diagram is called dataflow diagram or DFD.

The complexity of the model is handled by means of top-down decomposition: the first level shows the external actors and the external flows.

The structure of data is defined in the data dictionary.

The logic of activities is described in structured English.

What, rather than how

Old technique: informal mapping between inputs and outputs

Top-down decomposition

Case study

From: C. Gane and T. Sarson, Structured systems analysis: tools and techniques. Prentice-Hall, 1979.

The CBM (Computer Books by Mail) company receives customer orders (by mail) for books from customers, place publisher orders with appropriate publishers at a discount, and fills the customer orders on receipt of the books.

Requirements 1

Incoming orders are checked against a list of available books and against a customer file to see if the customer is in good credit standing (TBD, to be defined). Valid orders are put in a container of pending orders. If the checks fail, the order is rejected.

External actors: customers and publishers.

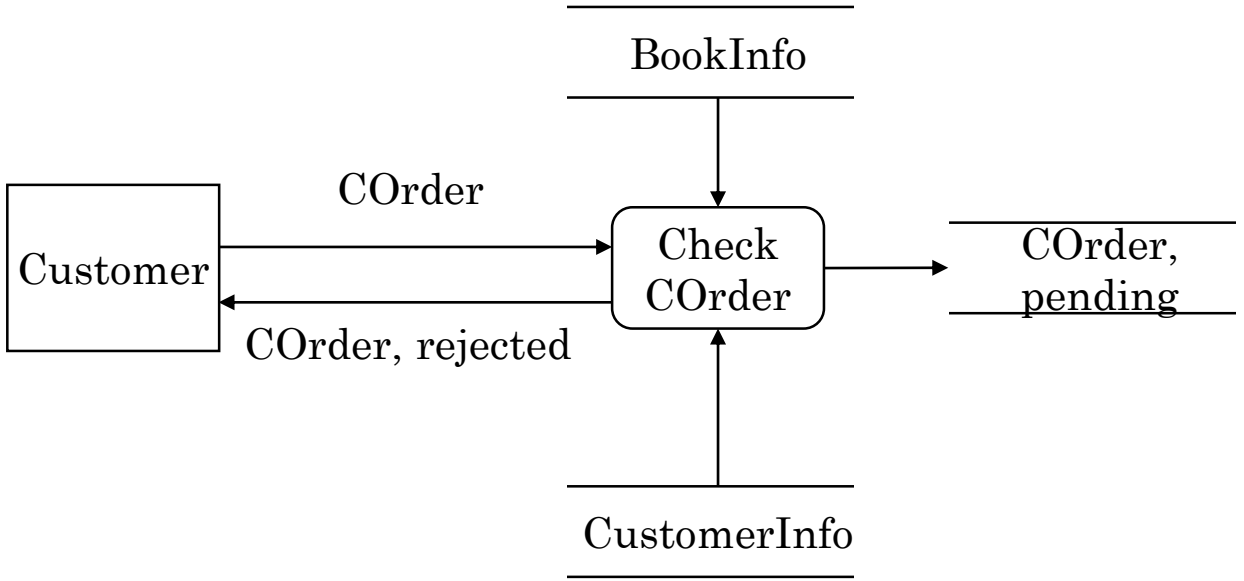
External data flows: customer orders, rejection notifications.

Activities: check customer order (resulting in valid order or rejection notification).

Remark: customer in good credit standing is a condition to be defined.

Exceptions are not considered in this preliminary analysis.

Model 1



COrder = Customer order

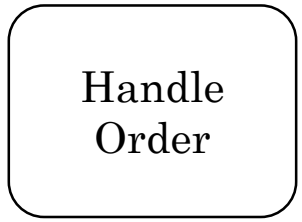
Comments:

The second label of a data flow indicates the meaning of the items passing through the data flow. The second label of a data store indicates the meaning of the items contained in the data store.

The two outputs of CheckCOrder are mutually exclusive; this fact has to be written separately as the notation provides no means to express it.

The elements of a dataflow diagram

An **activity** transforms input data into output data (physically or logically). It can be simple or compound. Its label is a verb + an object noun.

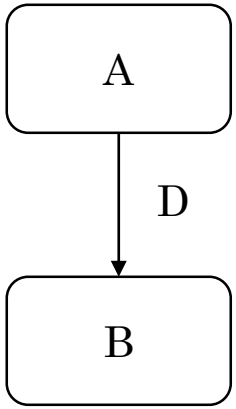


A **dataflow** is a path or conduit by which information is conducted from one element to another.

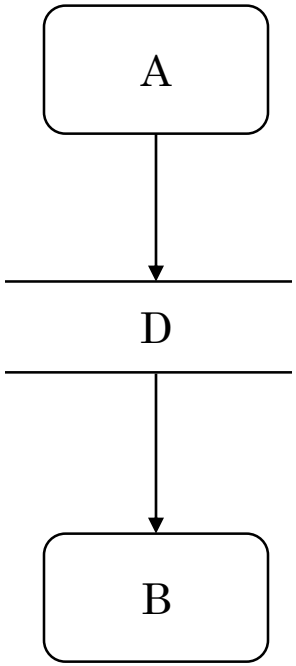
A **datastore** is a temporary holder of data. It allows items to be retrieved in an order that may be different from the order in which they were introduced. *4 meanings* (reading, extraction, update, insertion) which are not shown in the model.

External actors provide and/or consume external dataflows.

Data flows



What A produces is taken by B in the same order (pipeline).



Datastore D enables B to take (or read) data in the order they need.

Constraints:

no dataflows between actors

no dataflows between datastores

What is a customer order?

Description in the data dictionary (the order as a document); hierarchical description.

An order is made up of 3 sections, i.e. header, customerDetails, booksDetails.

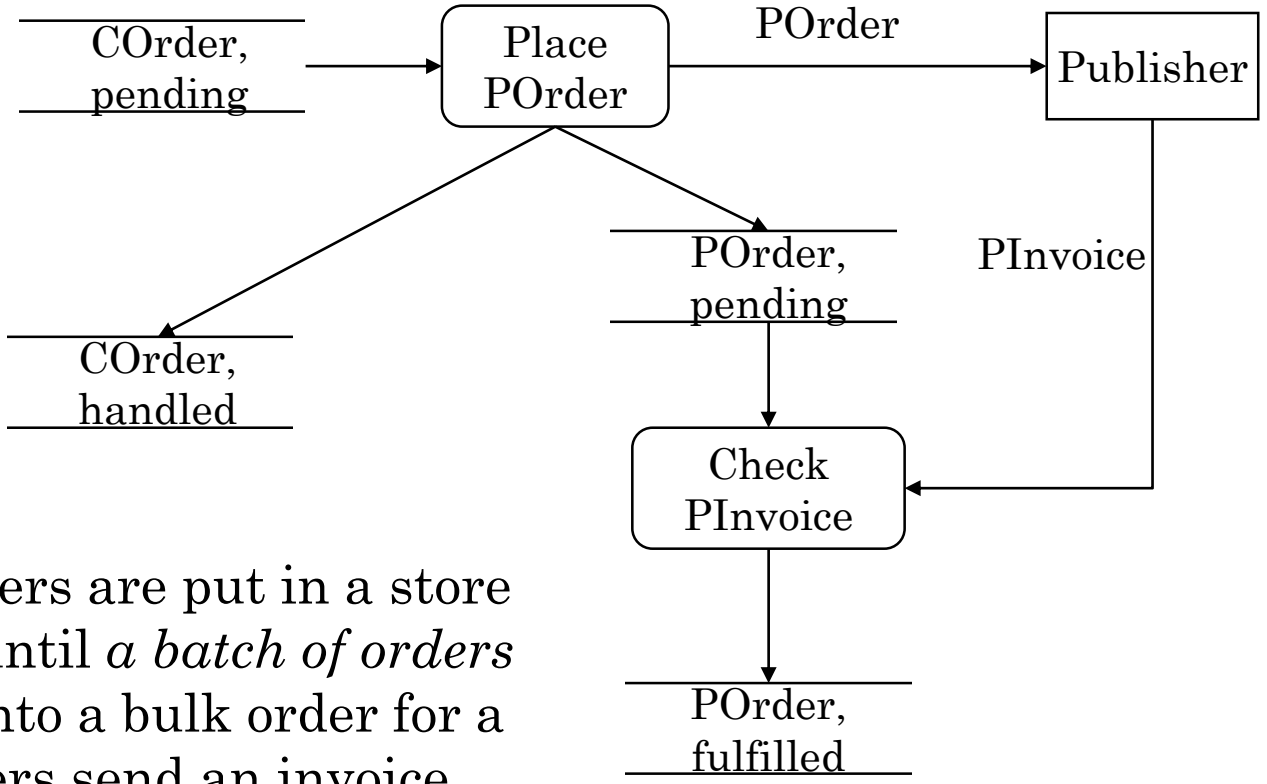
- header = order-date, [orderNum]
- customerDetails = organizationName, address
- booksDetails = {bookDetail}
- bookDetail = {authorName}, title, publisherName, nOfCopies.

[] means optionality

{} means repetition

In other terms, an order is made up of lines, each line referring to a book and indicating the number of copies needed.

Requirements 2



Valid customer orders are put in a store of pending orders until *a batch of orders can be assembled* into a bulk order for a publisher. Publishers send an invoice with each shipment, detailing its contents; it *must be compared* with the corresponding publisher order.

POrder = Publisher order

PInvoice = Publisher invoice

What is a publisher order?

Examples of cOrders

cOrder 100 from customer 1 includes

2 copies of book 1

1 copy of book 2.

cOrder 101 from customer 2 includes

3 copies of book 1

1 copy of book 3.

book 1 and book 2 are published by publisher 1, book 3 by publisher 2.

The following pOrders match the cOrders above

pOrder 50 to publisher 1:

5 copies of book 1

1 copy of book 2.

pOrder 51 to publisher 2:

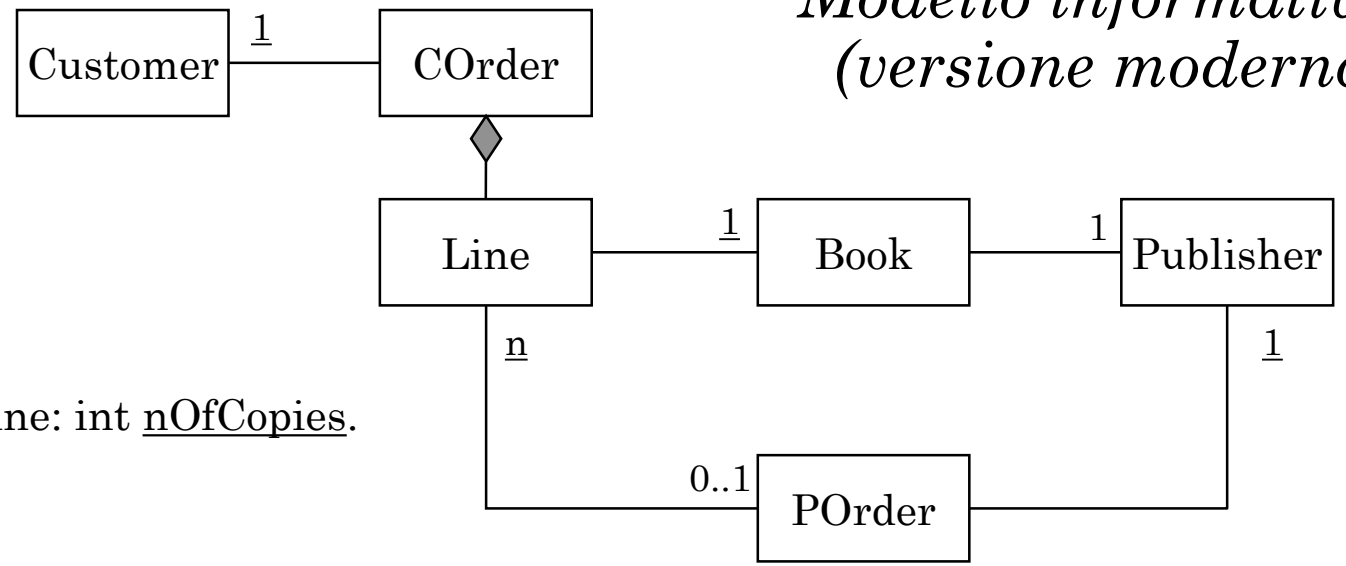
1 copy of book3

A publisher order includes a group of lines referring to books published by the publisher the order is to be sent to.

100, 101, 50 and 51 are order identifiers.

Modello informativo (versione moderna)

Una linea è associata ad un ordine cliente e può anche essere associata ad un ordine editore.



Invariante:

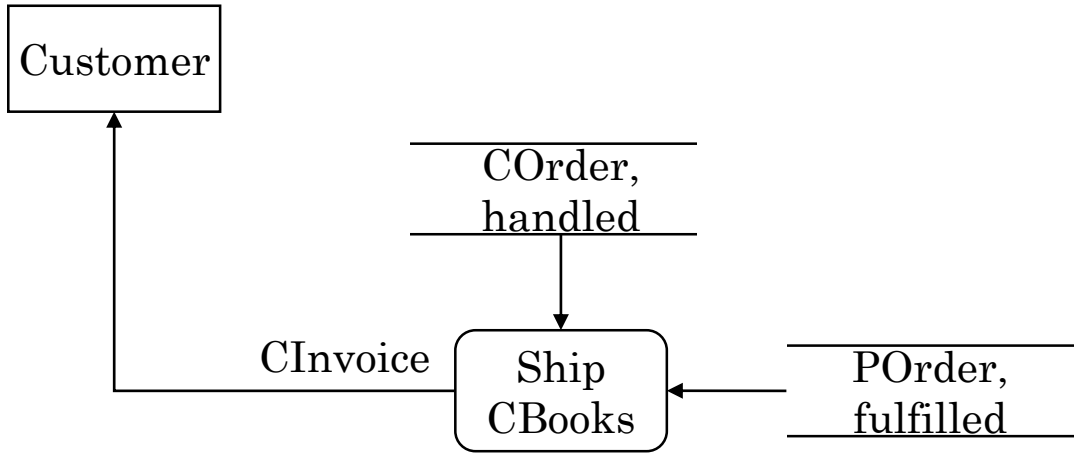
pOrder.publisher == pOrder.lines.book.publisher

Nota: una linea non è collegata a nessun pOrder se il cOrder a cui appartiene è stato respinto.

Si può introdurre la relazione derivata COrder – POrder (m,n).

in COrder: Set<POrder> pOrders = lines.pOrder.

in POrder: Set<COrder> cOrders = lines.cOrder.



Requirements 3

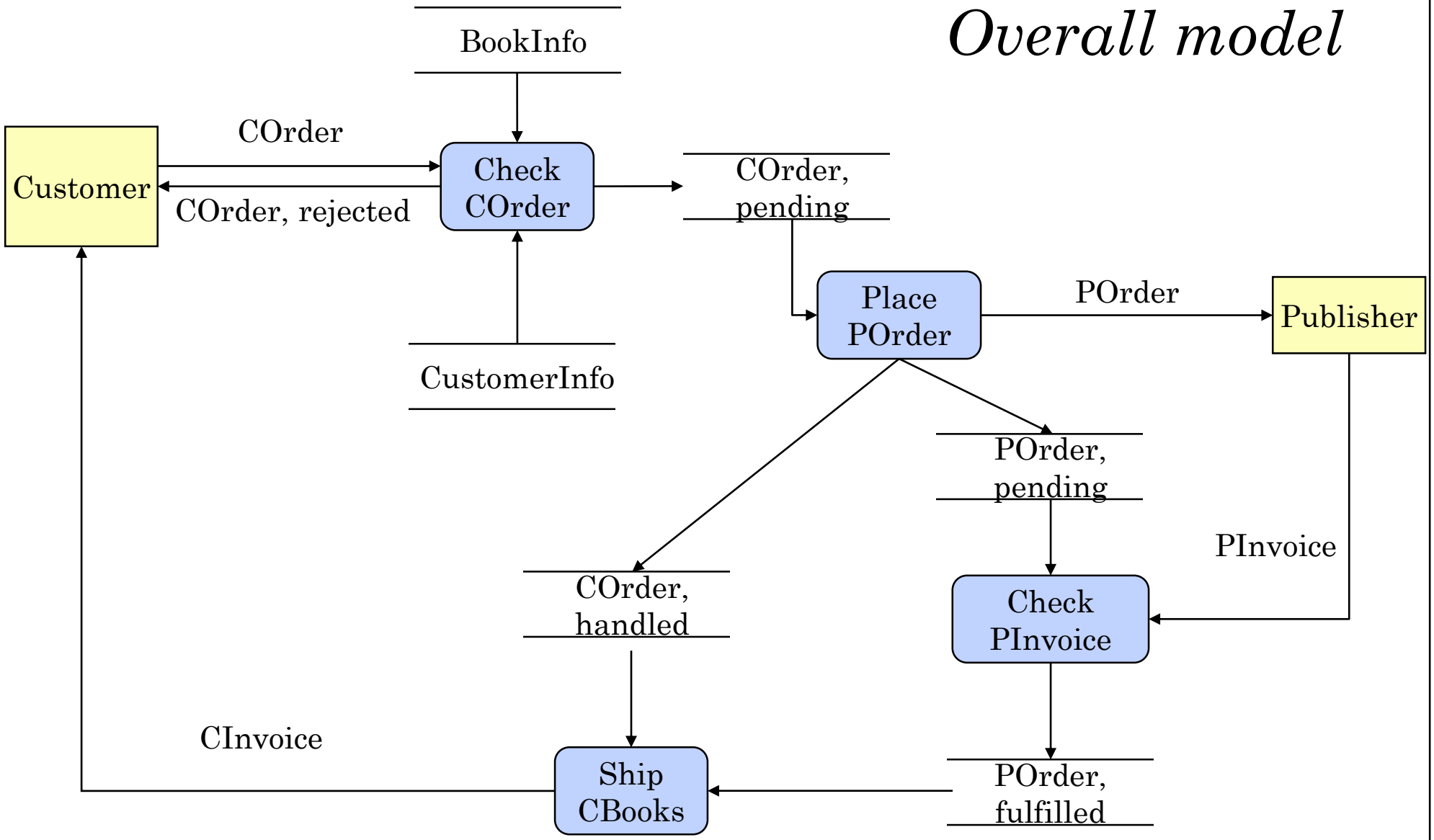
When all the books included in a customer order have been received, they are sent to the customer along with an invoice.

Comment:

When datastore POrder (fulfilled) contains all the pOrders associated with a cOrder in datastore COrder (handled), then ShipCBooks removes the cOrder from the datastore and produces a customer invoice related to the order. The books are sent to the customer along with the invoice.

ShipCBooks also removes a pOrder from datastore POrder (fulfilled), when there are no more cOrders in datastore COrder (handled) that are associated with it.

Overall model

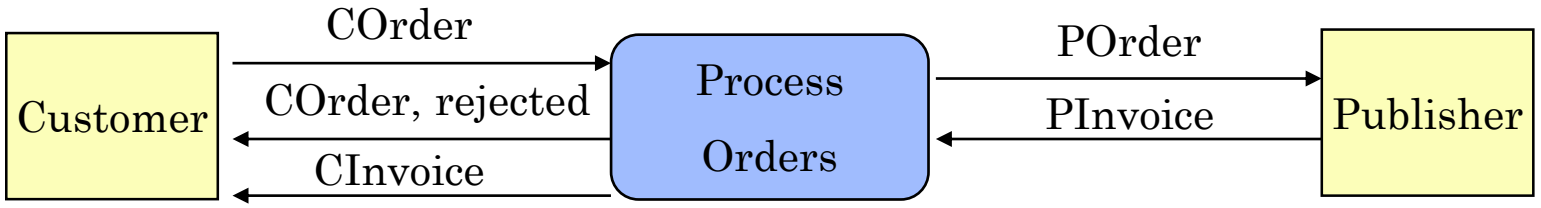


Singleton process

Il processo è **singleton**, cioè ha un'unica istanza in grado di trattare tutti gli ordini cliente.

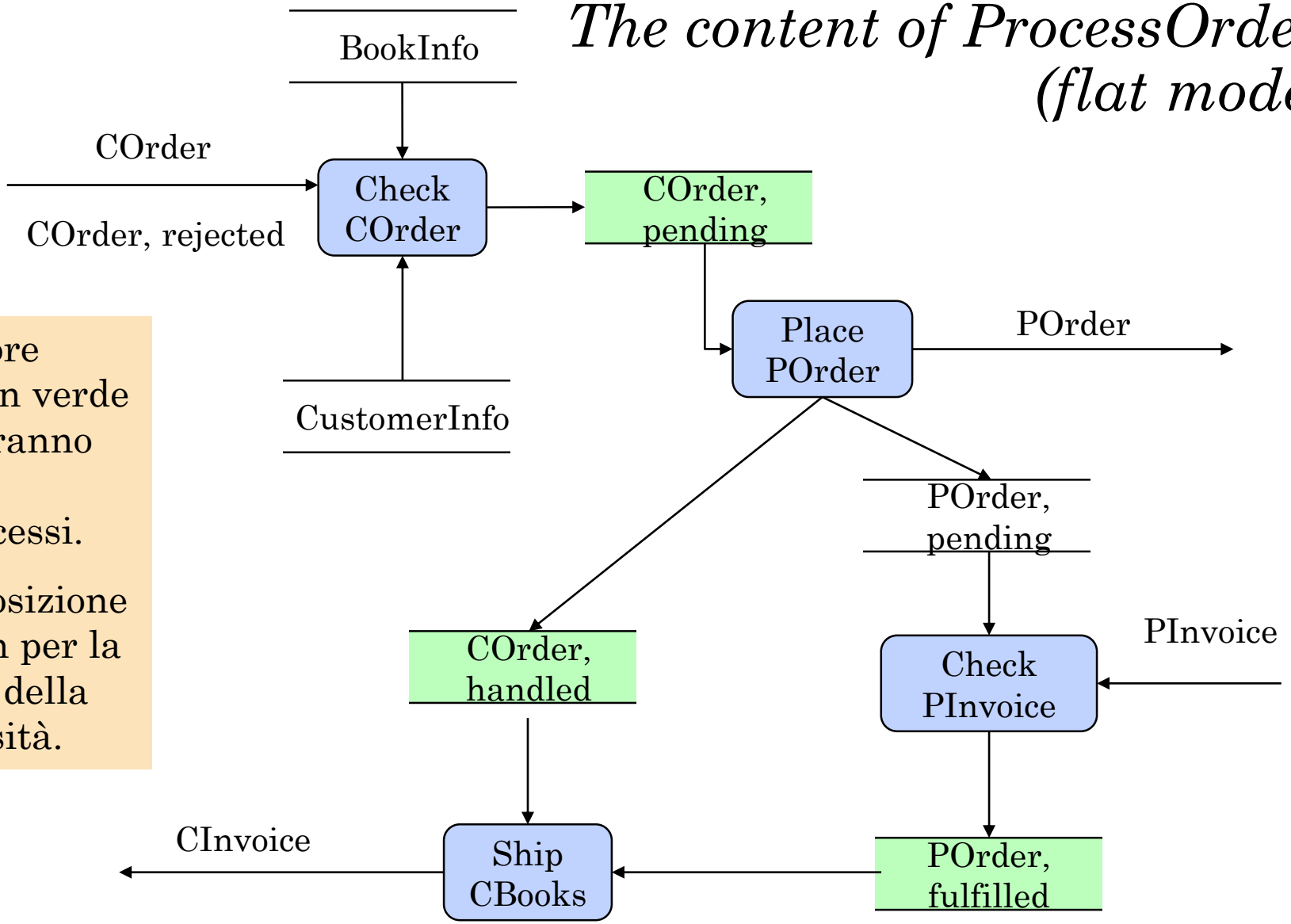
Gli ordini cliente non sono trattati separatamente perché i loro libri possono comparire in vari ordini fornitore. In generale, la corrispondenza tra ordini cliente e ordini fornitore è molti a molti (m,n).

Top-down decomposition: the context diagram



1. the process (the top level compound activity)
2. the external actors
3. the external flows

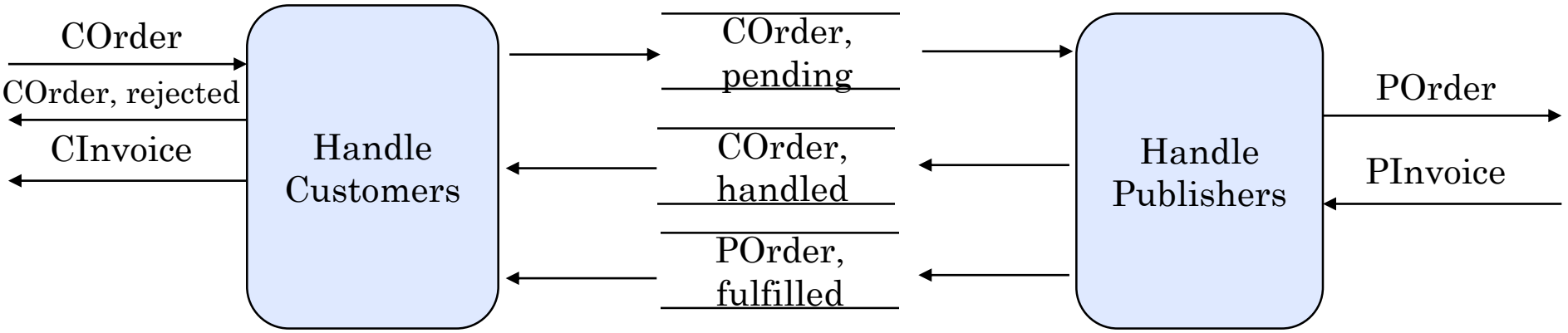
The content of ProcessOrders (flat model)



I datastore segnati in verde compariranno in due sottoprocessi.

Decomposizione top-down per la gestione della complessità.

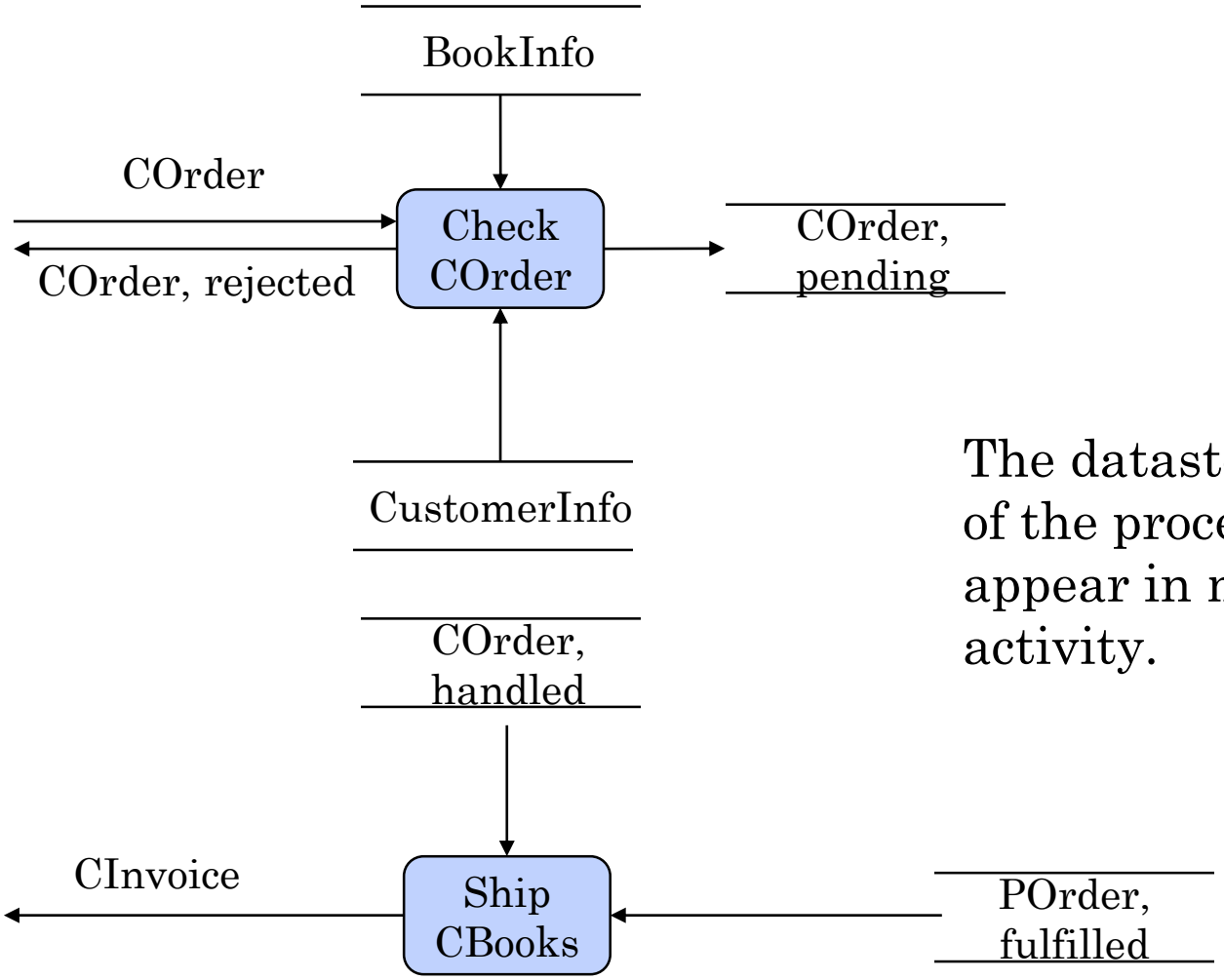
The hierarchical model of ProcessOrders



Two subprocesses, one handling customers and the other handling publishers.

Principle of flow balancing between outer views and inner ones.

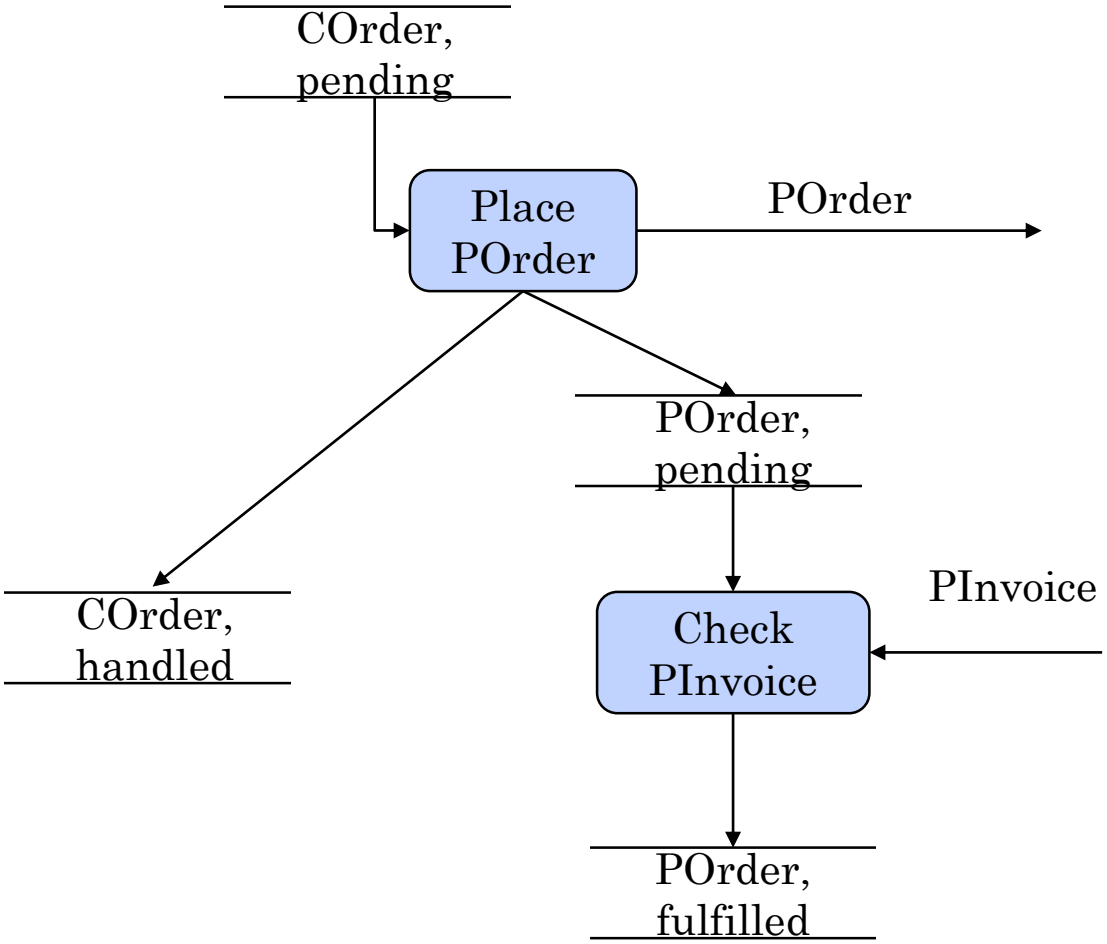
Handle Customers



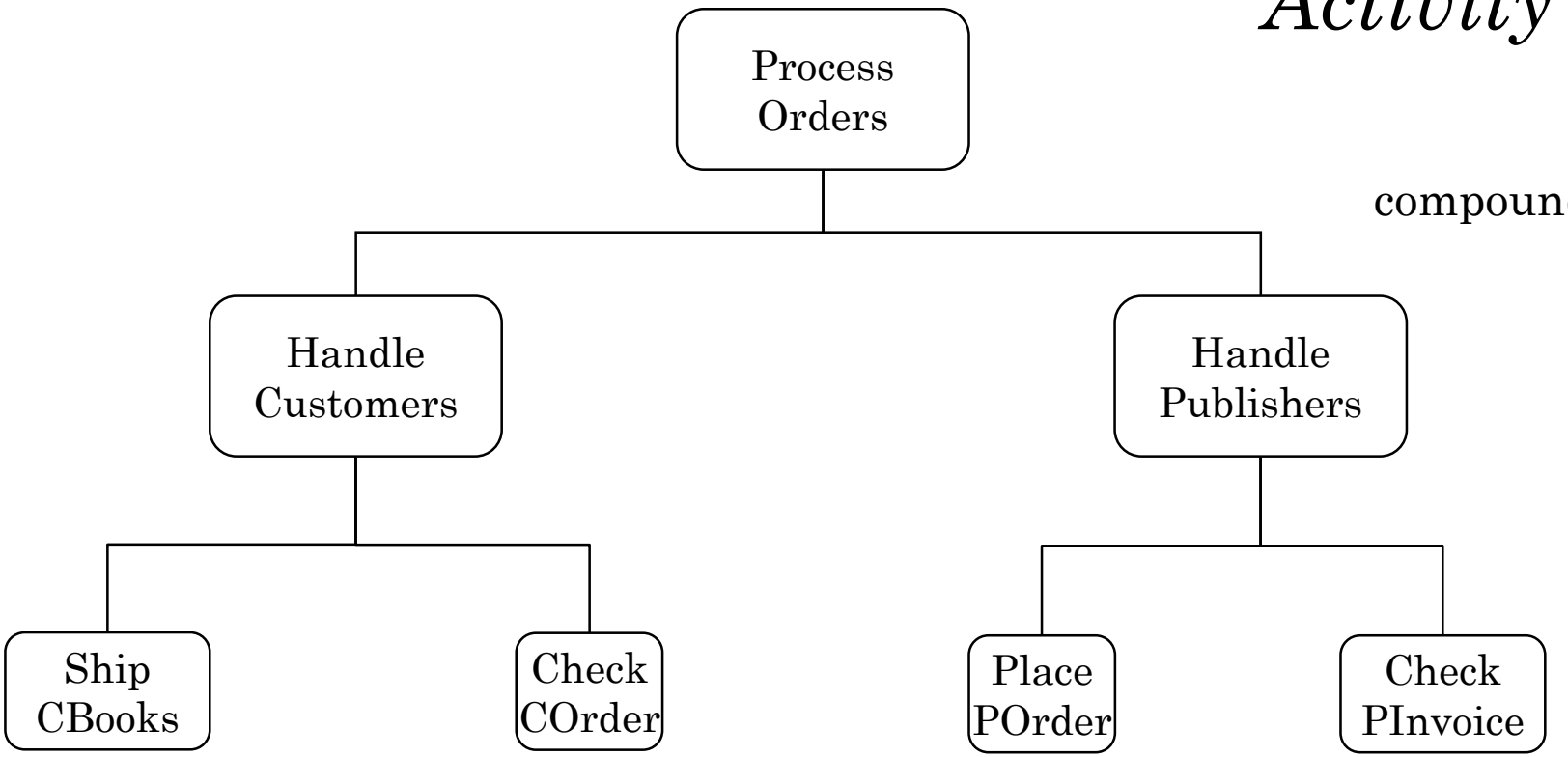
The datastores are global entities of the process and then they may appear in more than one compound activity.

Top-down decomposition: how many activities in a compound activity?

HandlePublishers



Activity tree



The leaves of the tree are simple activities.

Pros / cons

Pros

- Dataflow described
- Suitable for presenting the process with a top-down approach

Cons

- The precise dynamics is not apparent from the diagram and must be described separately
- Top-down decomposition driven by topological considerations (number of internal activities, minimizing the number of connections between compound activities).

State models

State models

The life cycle of an object encompasses a number of stages (or states).

The life cycle is driven by *external events*. The object is normally waiting for something to happen. Waiting takes place in specific states, as in each state only a subset of all the possible events are expected.

A state model shows the states and the *transitions* leading from each state to the subsequent ones.

Transitions are event-driven and can be associated with actions (activities) to be performed when they are traversed.

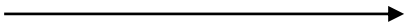
A simple state model

When button P is pressed the lights are alternately turned on and off.

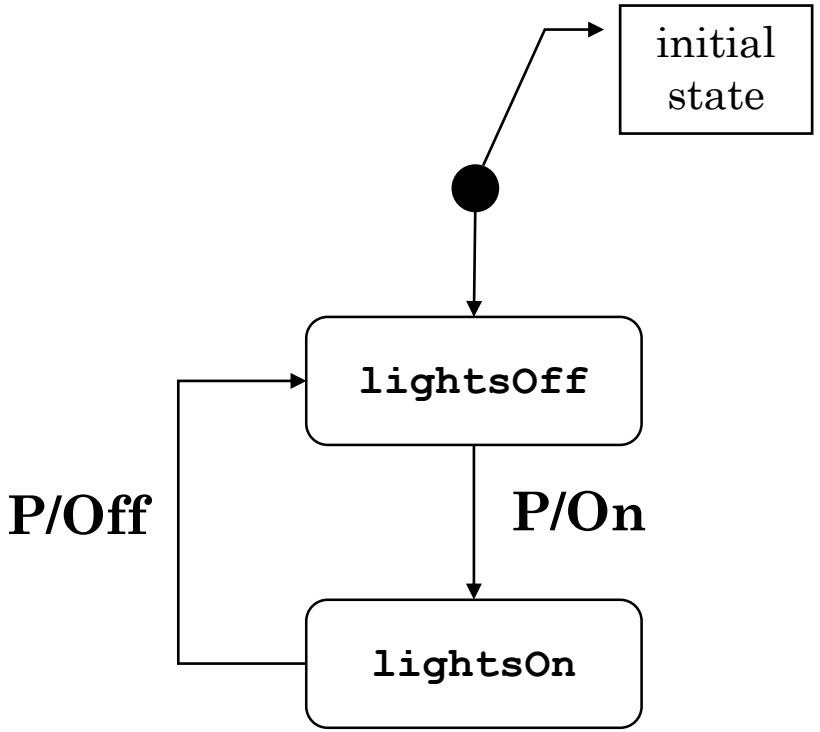
Events: P (button pressed)

Commands: on, off

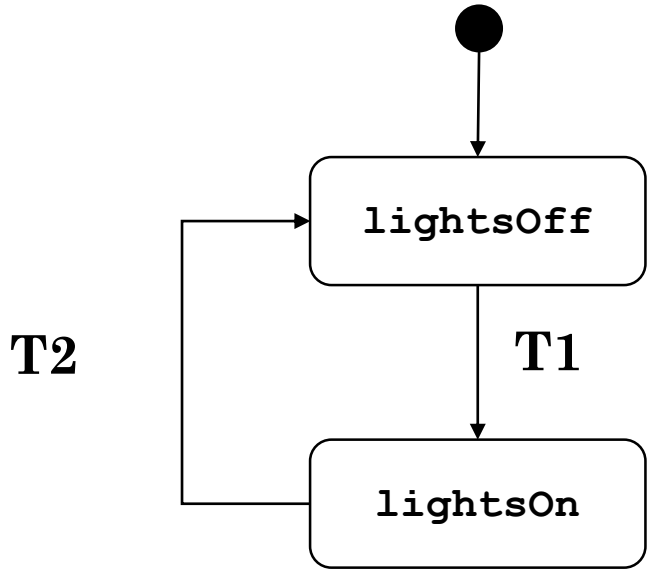
event/command(s)



transition



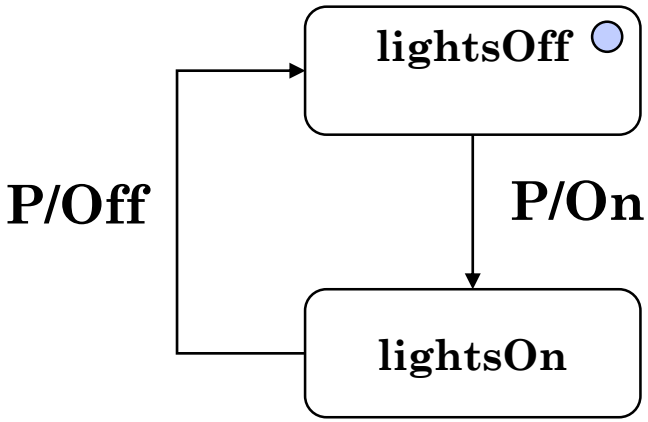
Notation for transitions



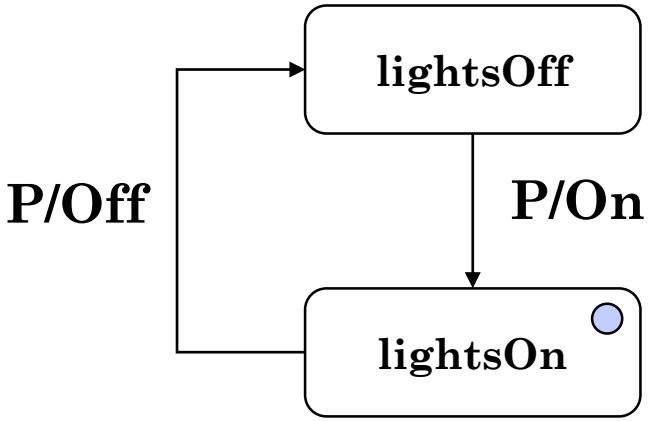
T1: P/on

T2: P/off

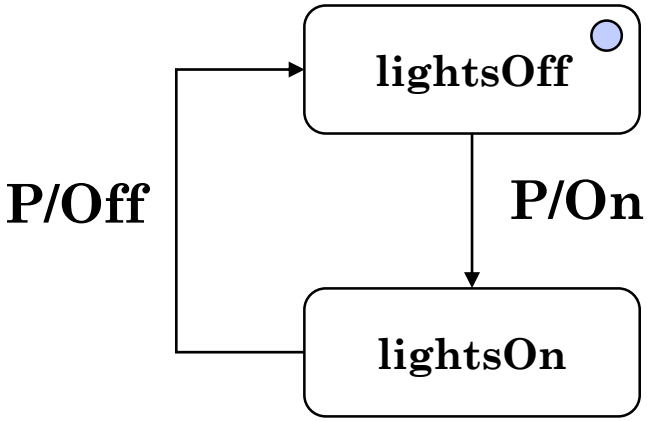
Current state



token



after the first P



after the second P

Remarks

The same event cannot label two transitions originating from the same state.

Events are processed one at a time as they are received.

If an event does not trigger any transition in the current state, it will be discarded.

A reaction controller

Components:

tank

vessel

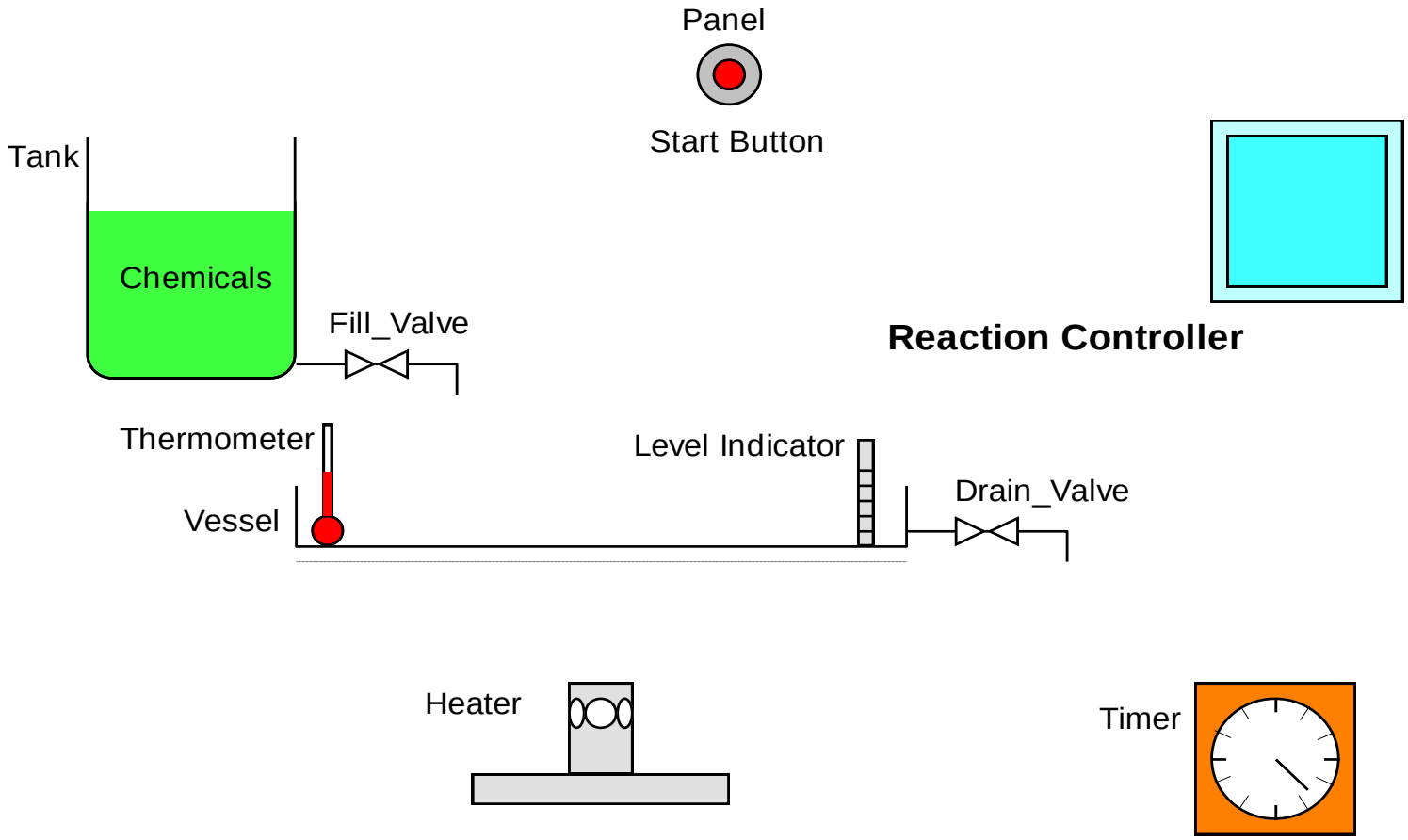
heater

timer

panel

reaction controller

The chemical plant



A chemical reaction controller

When the operator presses the start button, the fill valve is opened and the reagent flows from a tank into the reaction vessel. When the vessel is full, the fill valve is closed, the heater is swiched on and the timer is activated.

When the vessel temperature goes out of range, the heater is switched on (if below the lower bound) or off (if above the upper bound). When the timer goes off, the heater is switched off (if is is on) and the drain valve is opened. When the vessel is empty, the drain valve is closed and a new reaction can be started.

Events and commands

Events

Panel: Start_Filling

Vessel:

Vessel_Full, Vessel_Empty, Temperature_High, Temperature_Low

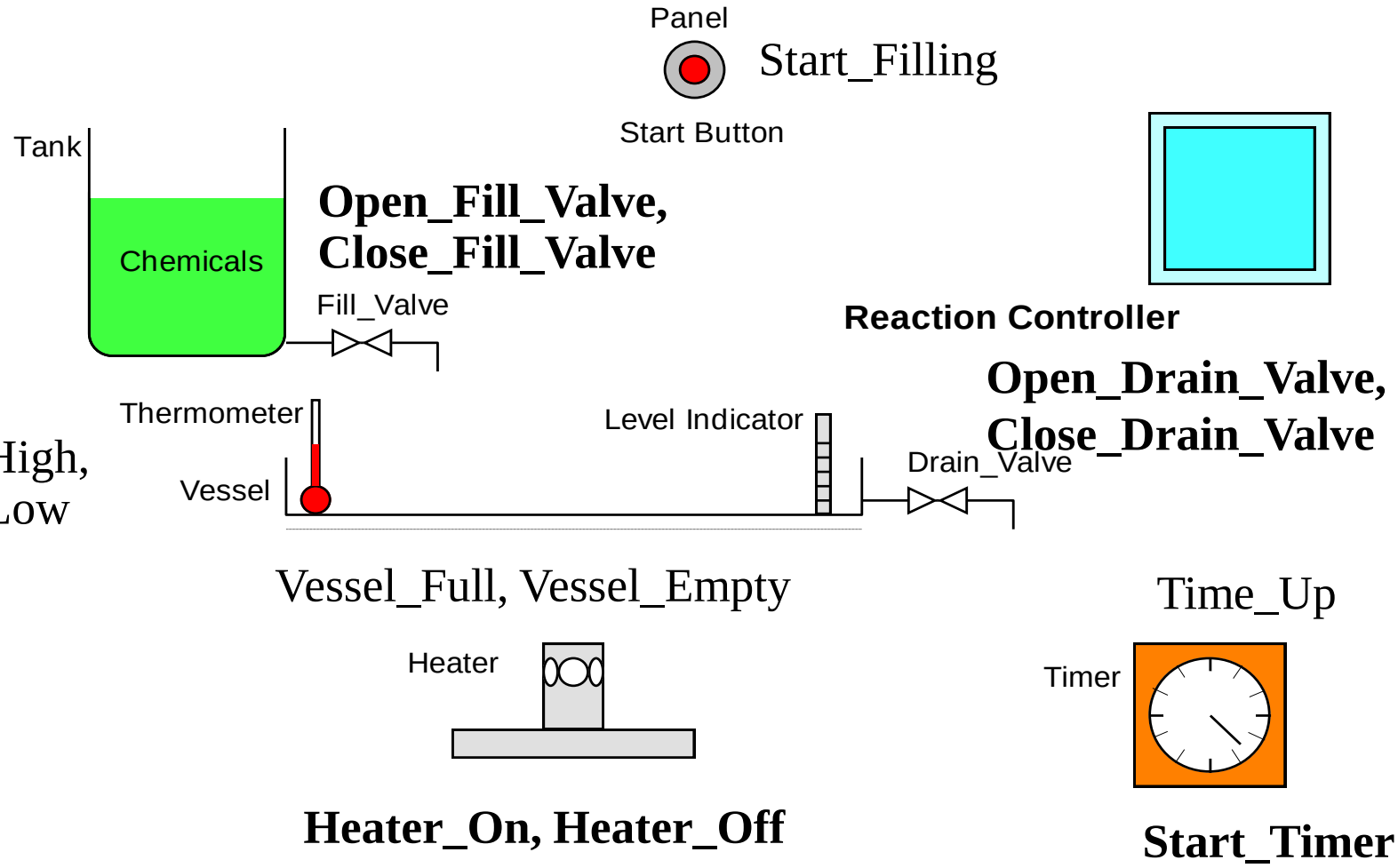
Timer: Time_Up

Commands

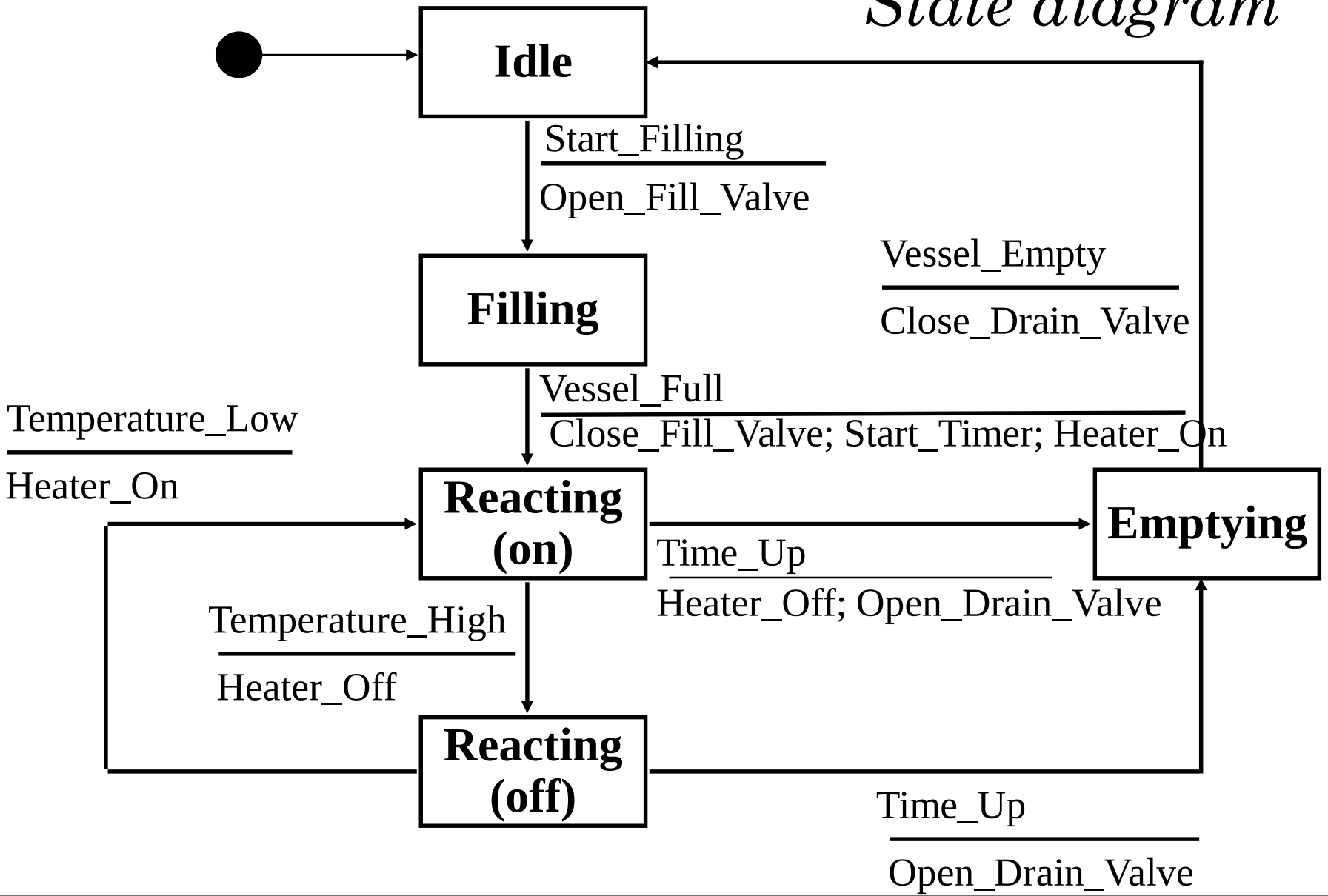
Fill_Valve: Open_Fill_Valve, Close_Fill_Valve
Drain_Valve: Open_Drain_Valve, Close_Drain_Valve
Timer: Start_Timer
Heater: Heater_On, Heater_Off

Events and ***commands***

The chemical plant



State diagram



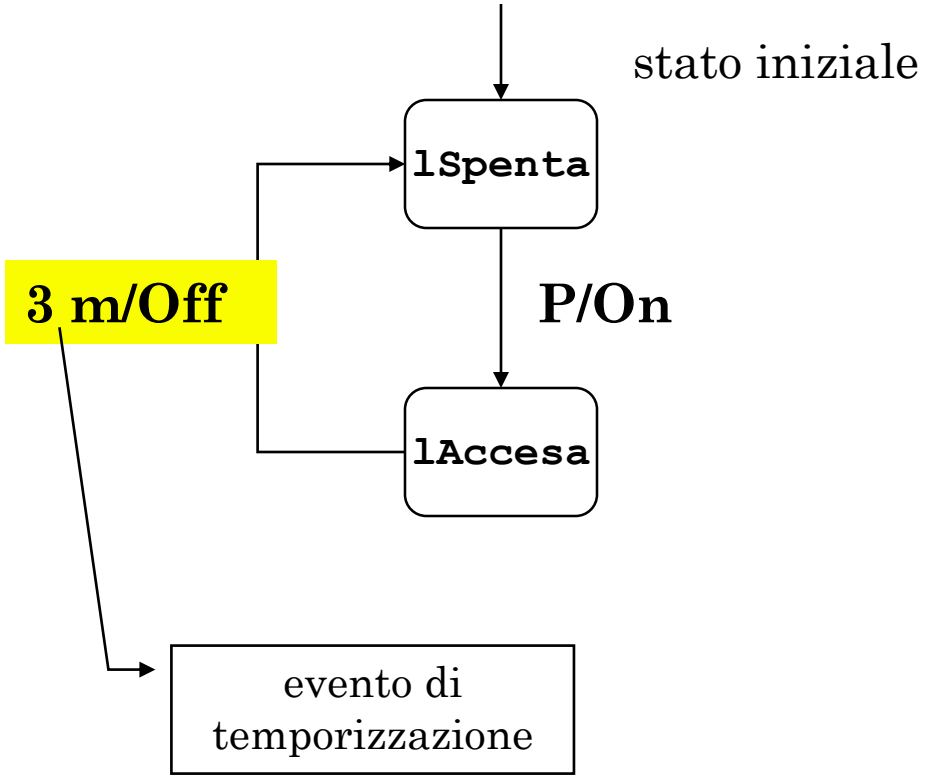
Transizioni temporizzate

Premendo il pulsante P si accende la lampadina che illumina la cantina; dopo 3 minuti la lampadina si spegne automaticamente.

Evento: P

Comandi: On, Off

La transizione temporizzata scatta dopo che il sistema è rimasto per 3 minuti nello stato lAccesa. Se nel frattempo scattasse un'altra transizione (diretta ad un altro stato) la temporizzazione verrebbe cancellata.



Operational State Machine

Si tratta di una calcolatrice che esegue somme e sottrazioni.

Un evento è un numero intero (n) oppure un operatore (+,-) o il terminatore ";".

Il valore del numero si trova nell'attributo **val** (payload dell'evento).

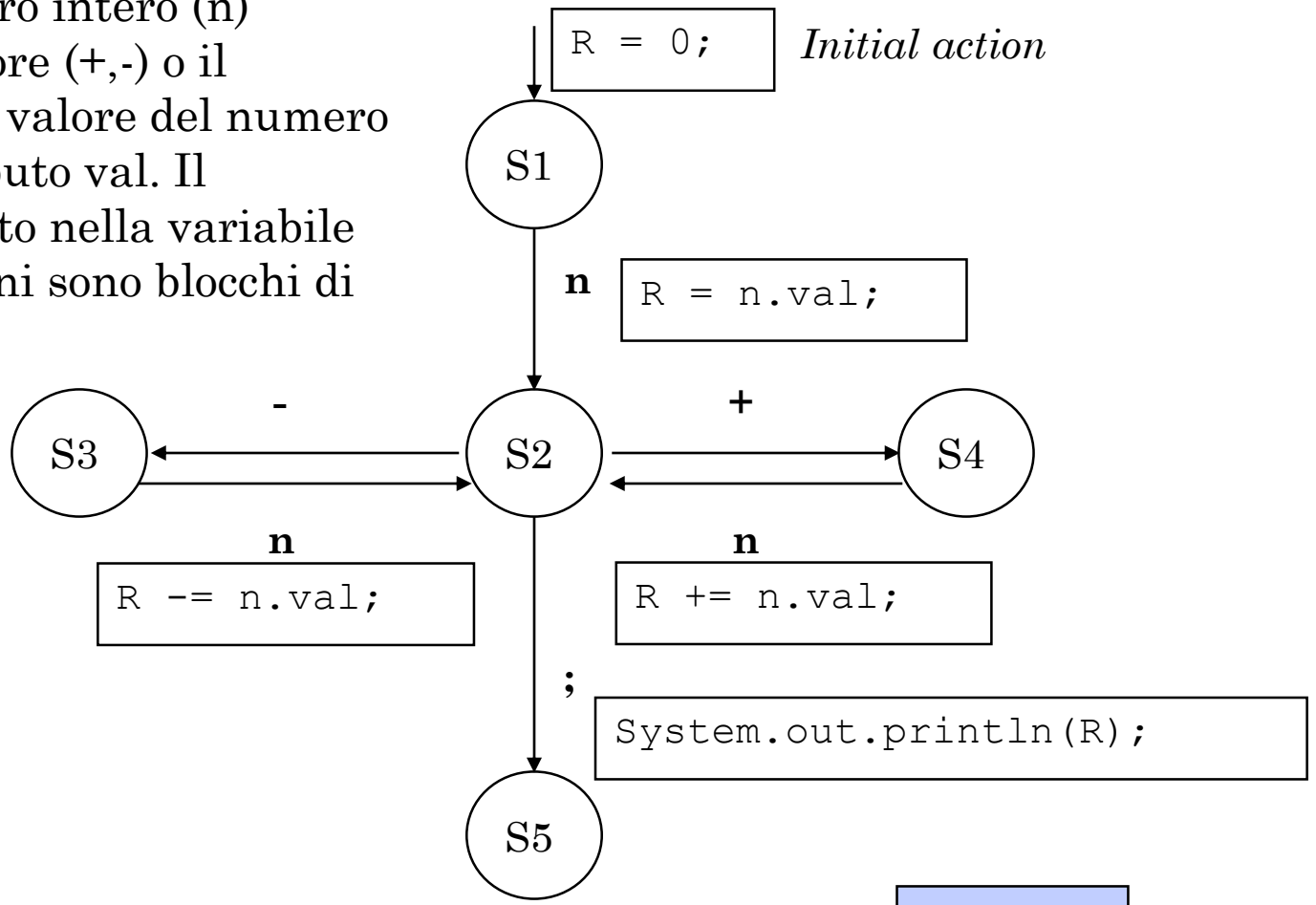
Il risultato è costruito nella variabile globale R.

Le azioni sono blocchi di codice.

Nota: si introducono variabili globali, cioè accessibili da tutte le transizioni.

Operational State Machine

Si tratta di una calcolatrice che esegue somme e sottrazioni. Un evento è un numero intero (n) oppure un operatore (+,-) o il terminatore ";". Il valore del numero si trova nell'attributo val. Il risultato è costruito nella variabile globale R. Le azioni sono blocchi di codice.




```
int R;
```

Esempio:

5+3-2;

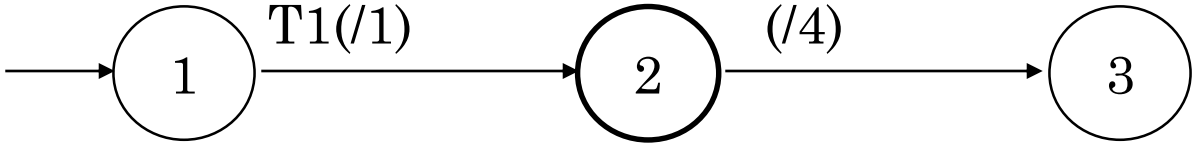
Hierarchical State Machines

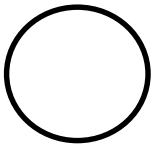
Compound states

Group transitions

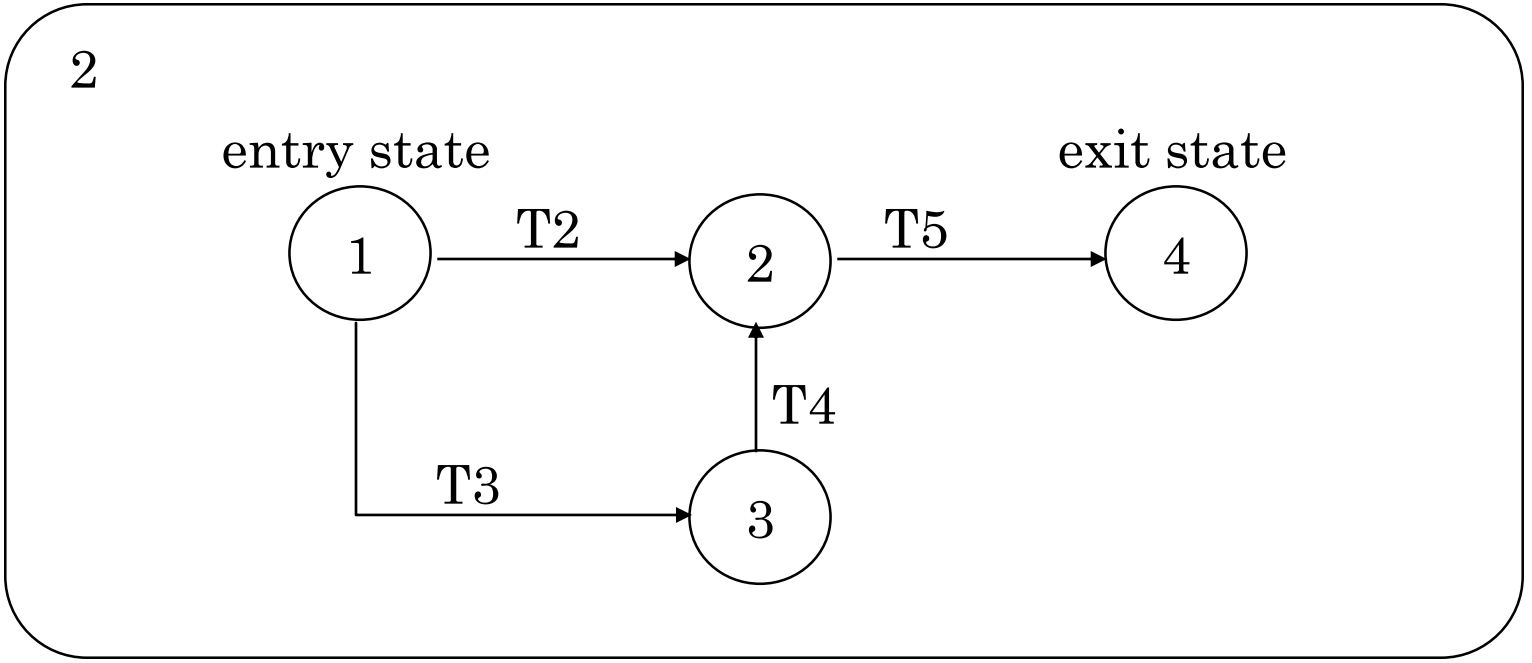
History transitions

Stati composti

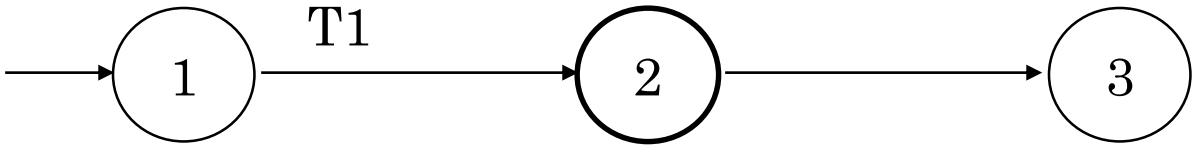




the icon of a compound state



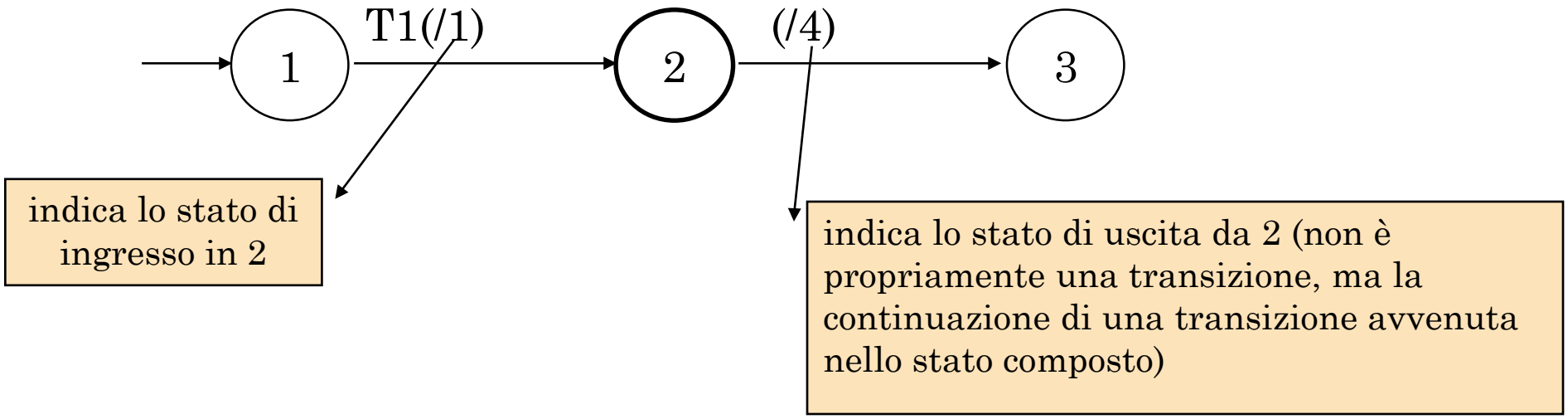
con un solo entry state
e un solo exit state in 2

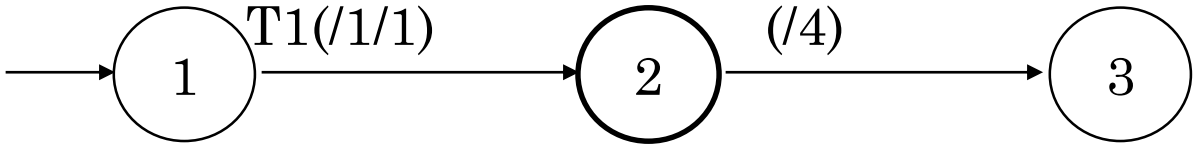


Note

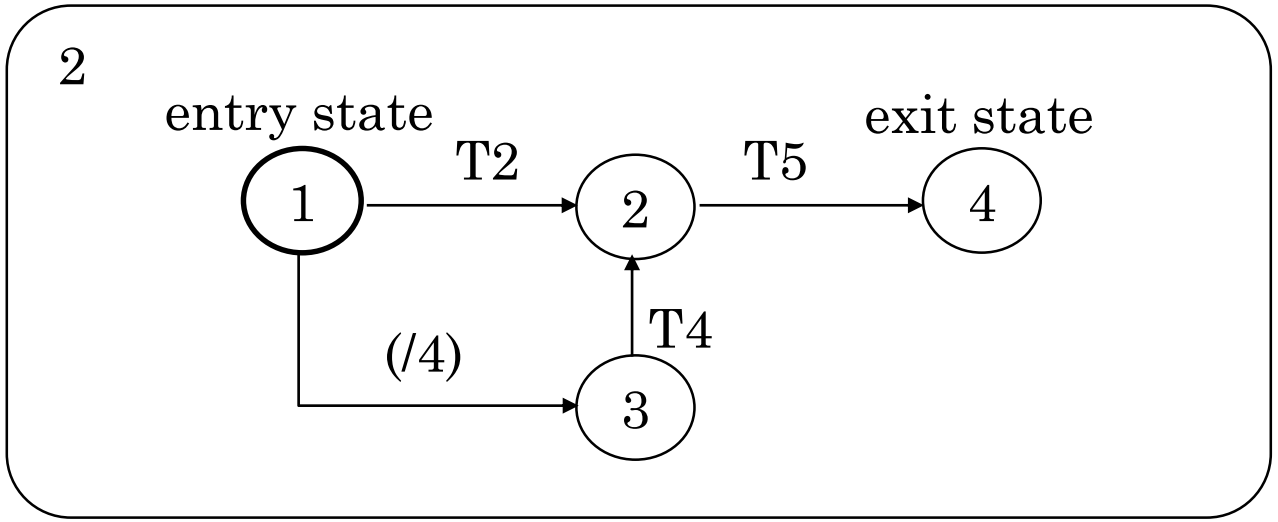
Uno stato composto può avere uno o più entry states e uno o più exit states.

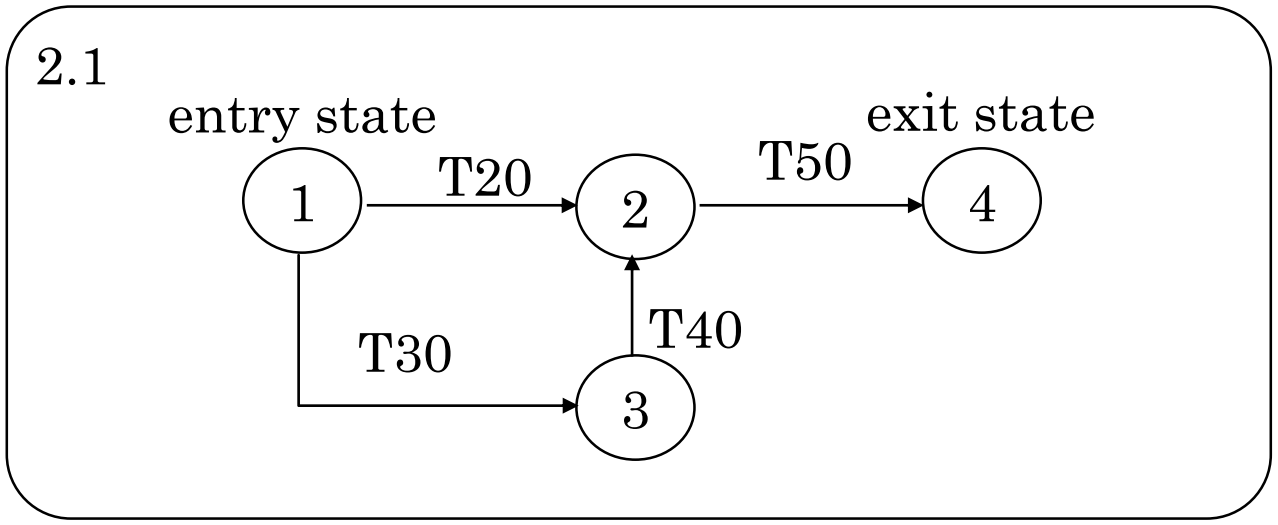
T_i ⟶ Le transizioni hanno nomi globali, a cui corrispondono coppie evento/azione





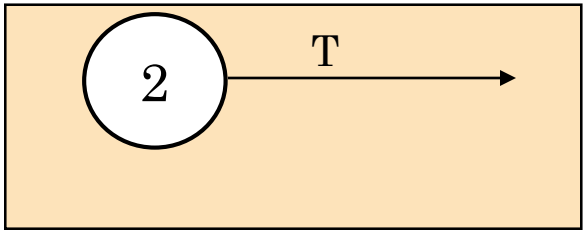
Composizione a più livelli

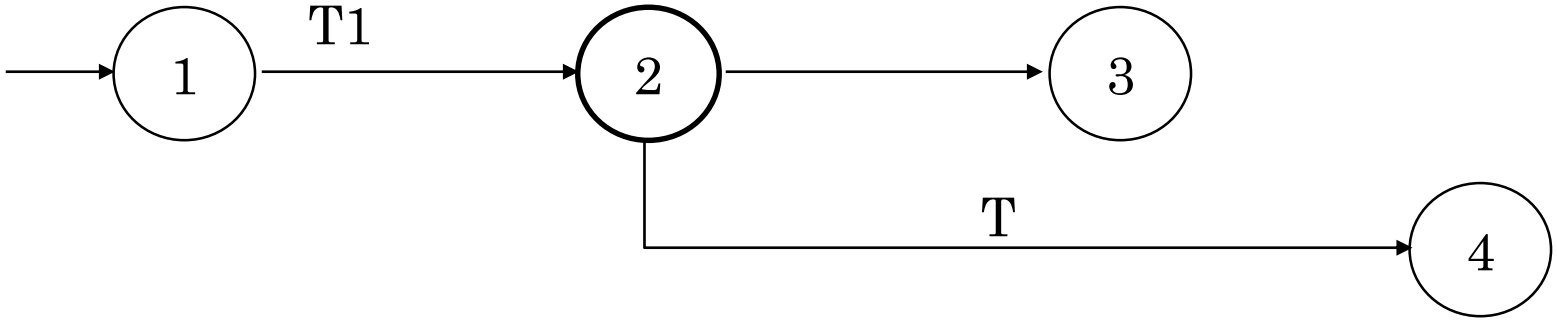


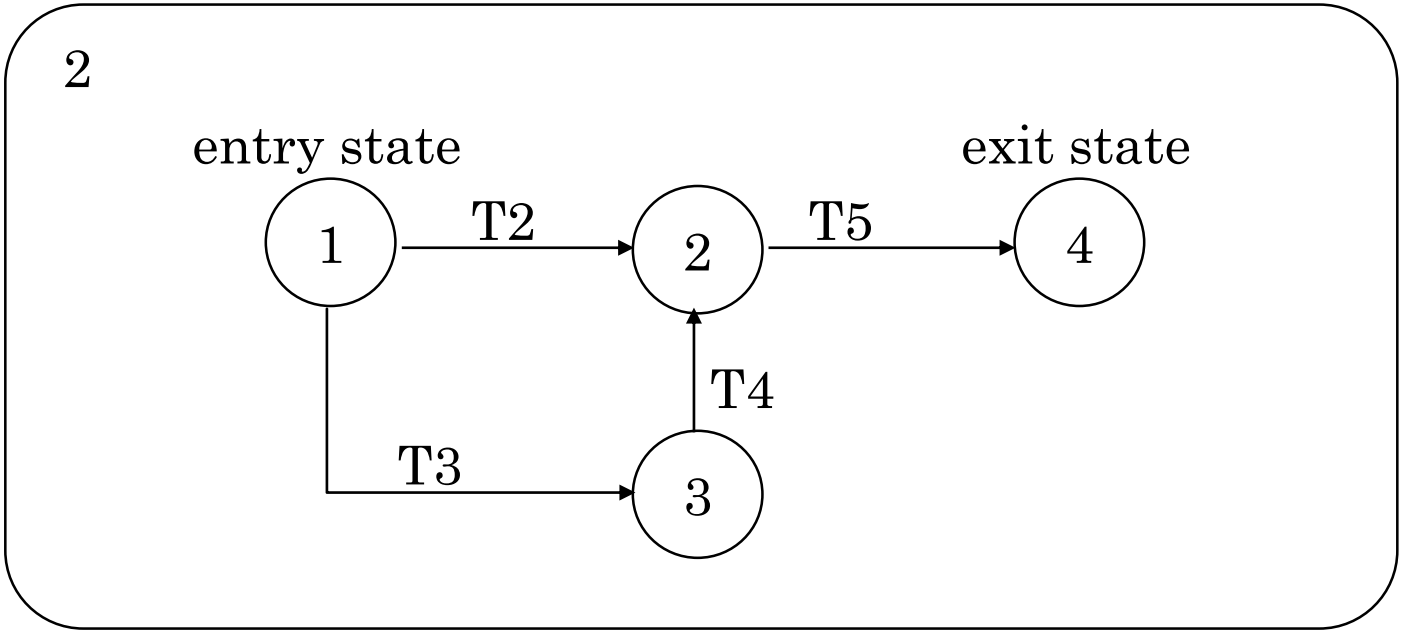


- Un entry state può essere composto, un exit state no.
- (/1/1) è un pathname relativo allo stato d'arrivo.
- T2 è una transizione di gruppo.

Transizione di gruppo

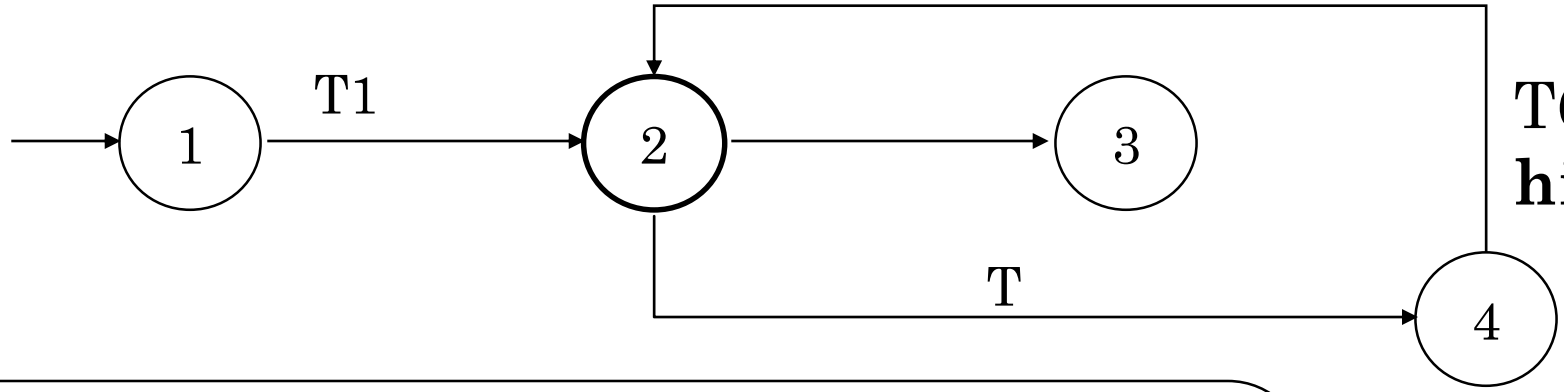


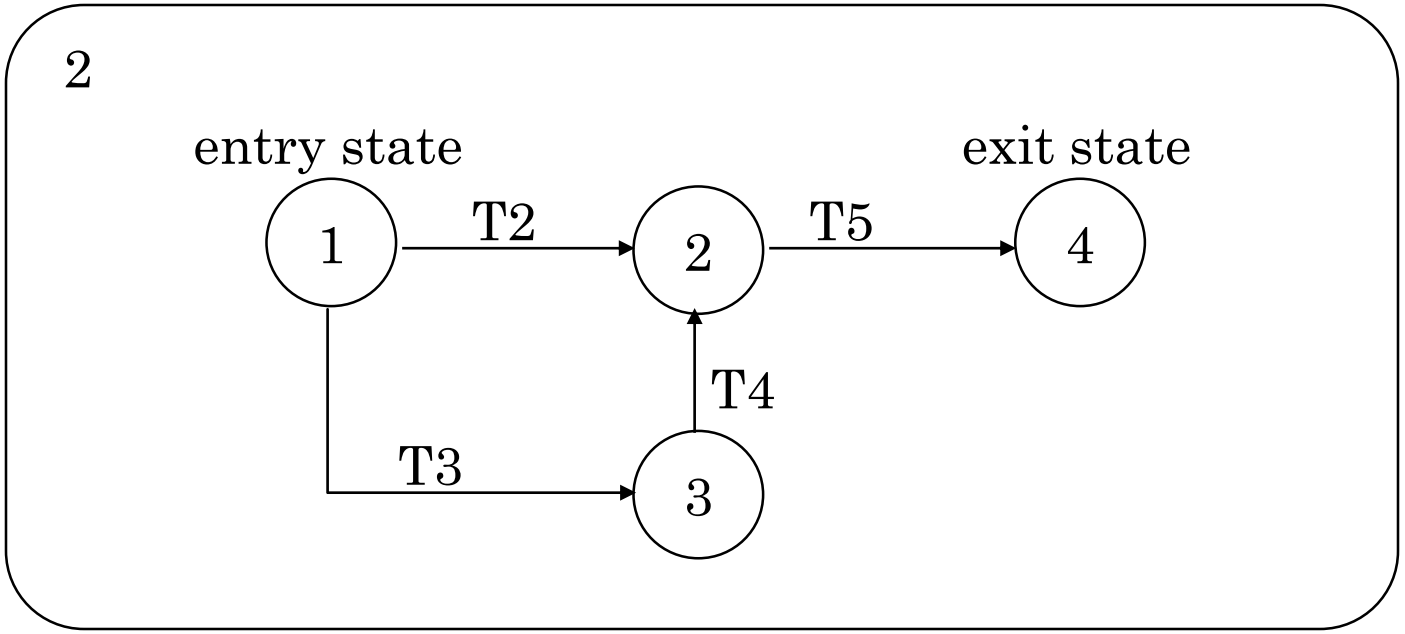




T causa la transizione allo stato 4 da un qualsiasi sottostato (a qualsiasi livello) dello stato 2.

Transizione di storia





T6 (t. di storia) porta nel sottostato di 2 lasciato dalla transizione di gruppo T. Una t. di storia è utile nel caso in cui un'anomalia sia stata trattata e il processo possa riprendere dal punto in cui è stato interrotto.

Moore and Mealy machines

In una **Mealy** machine le azioni sono associate alle transizioni, mentre in una **Moore** machine sono associate agli stati.

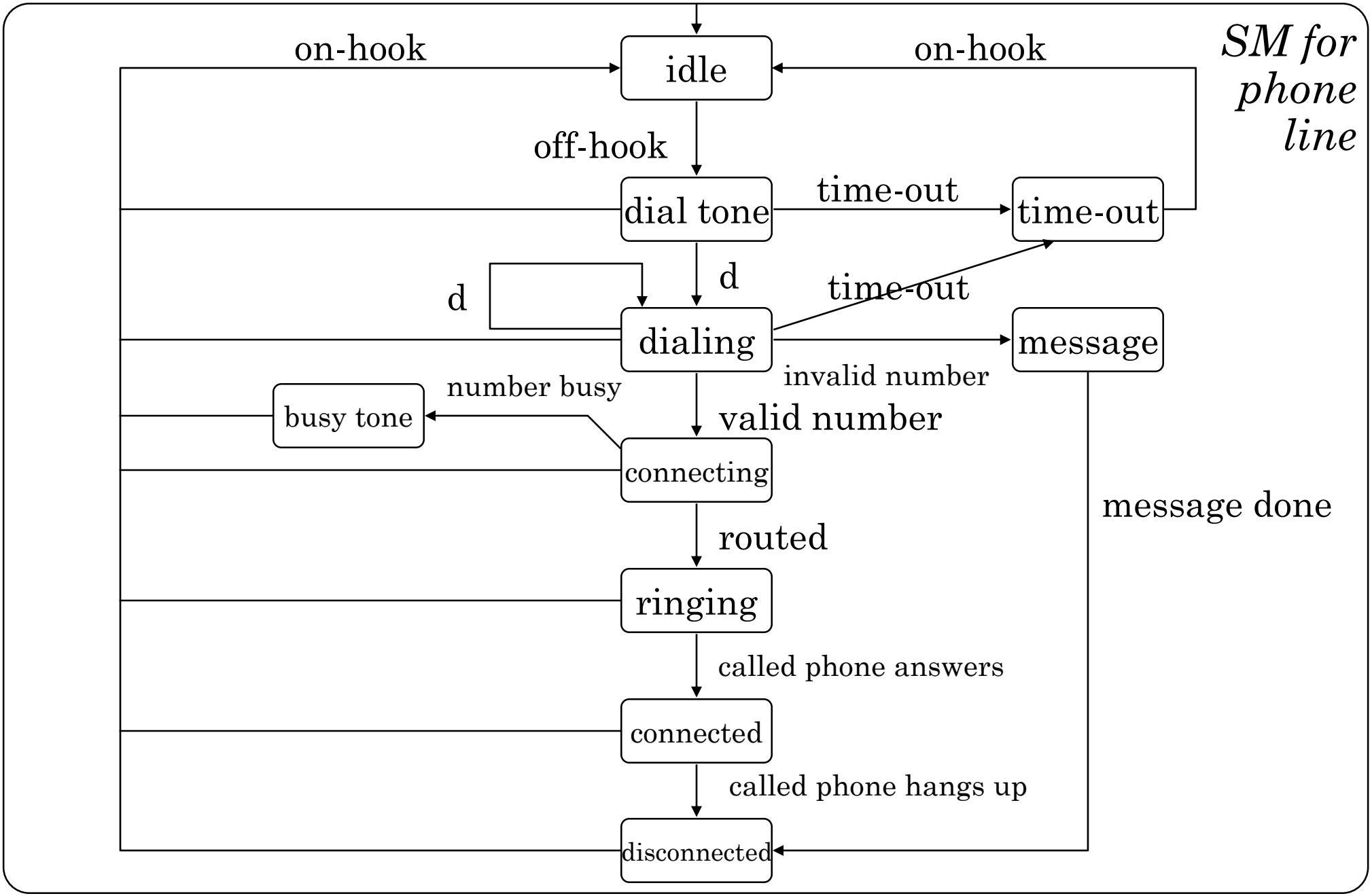
I due modelli sono equivalenti.

Nelle implementazioni le azioni si possono trovare associate sia alle transizioni sia agli stati.

Esempi di Moore machines:

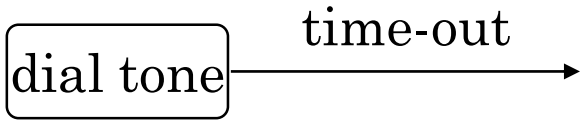
SM for phone line

Cruise control

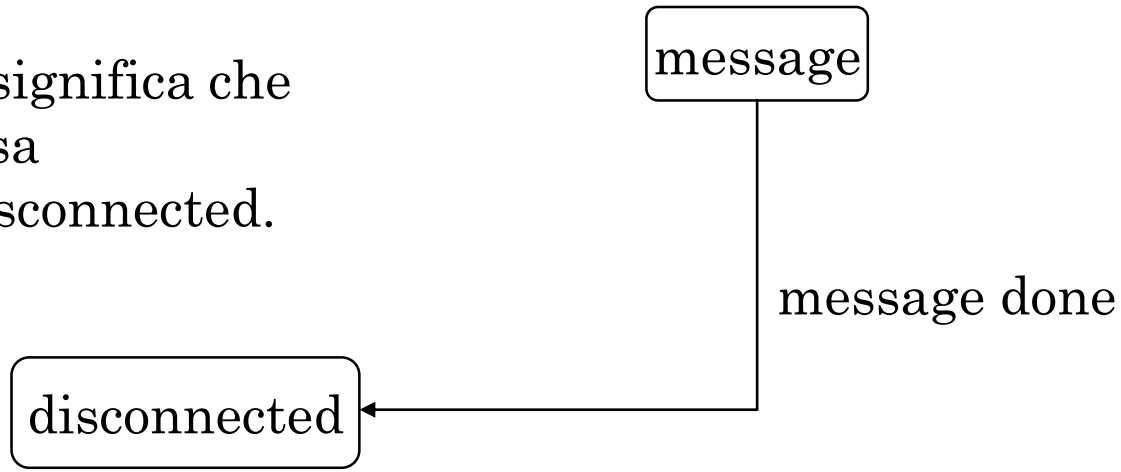

SM for phone line
idle
on-hook
on-hook
off-hook
dial tone
time-out
time-out
d
d
time-out
dialing
message
invalid number
number busy
busy tone
valid number
connecting
message done
routed
ringing
called phone answers
connected
called phone hangs up
disconnected

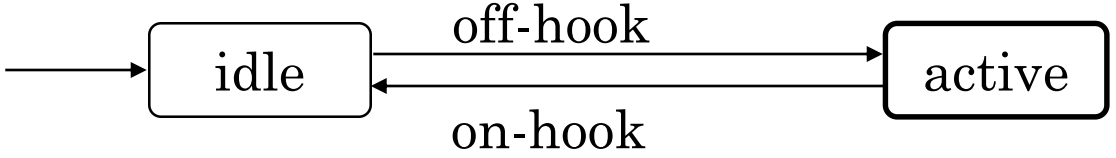
Note

Il significato dell'azione è dato dal nome dello stato. L'azione genera il dial tone e quando termina la durata (senza che sia inserita la prima cifra del numero), emette l'evento time-out.

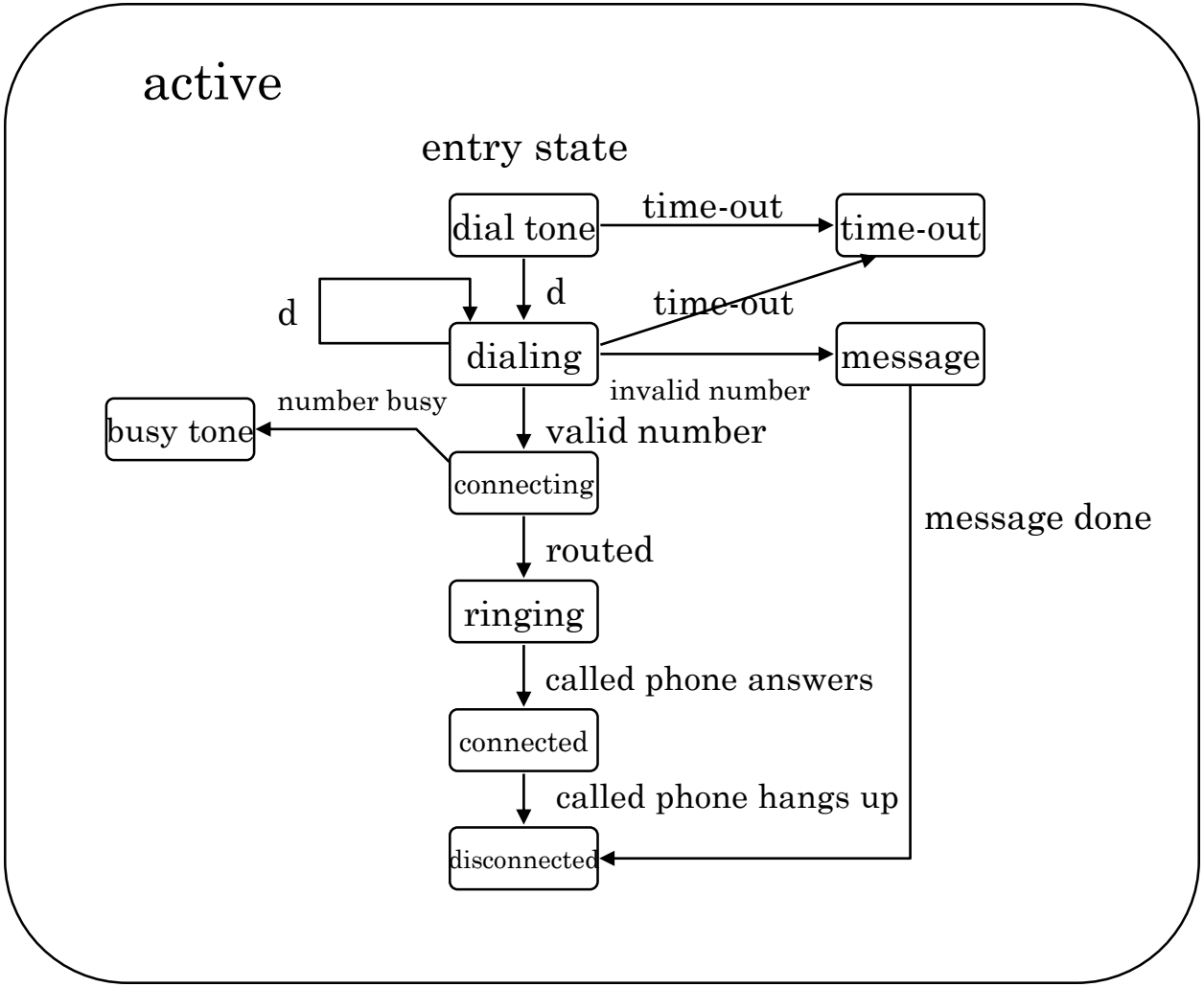


La transizione *message done* significa che terminato il messaggio si passa immediatamente allo stato disconnected.





HSM for phone line

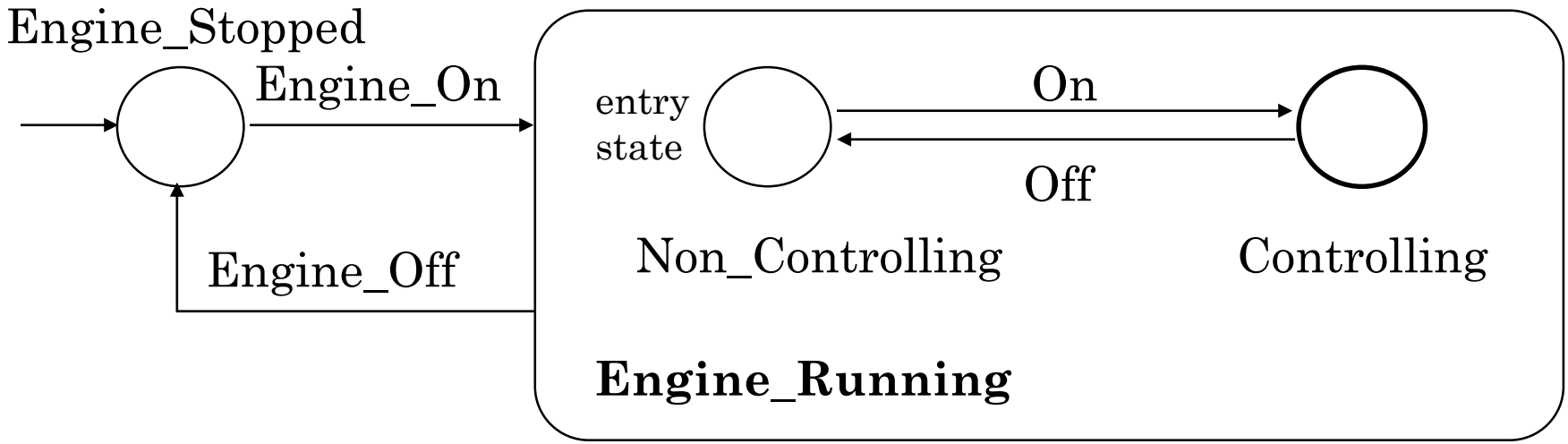


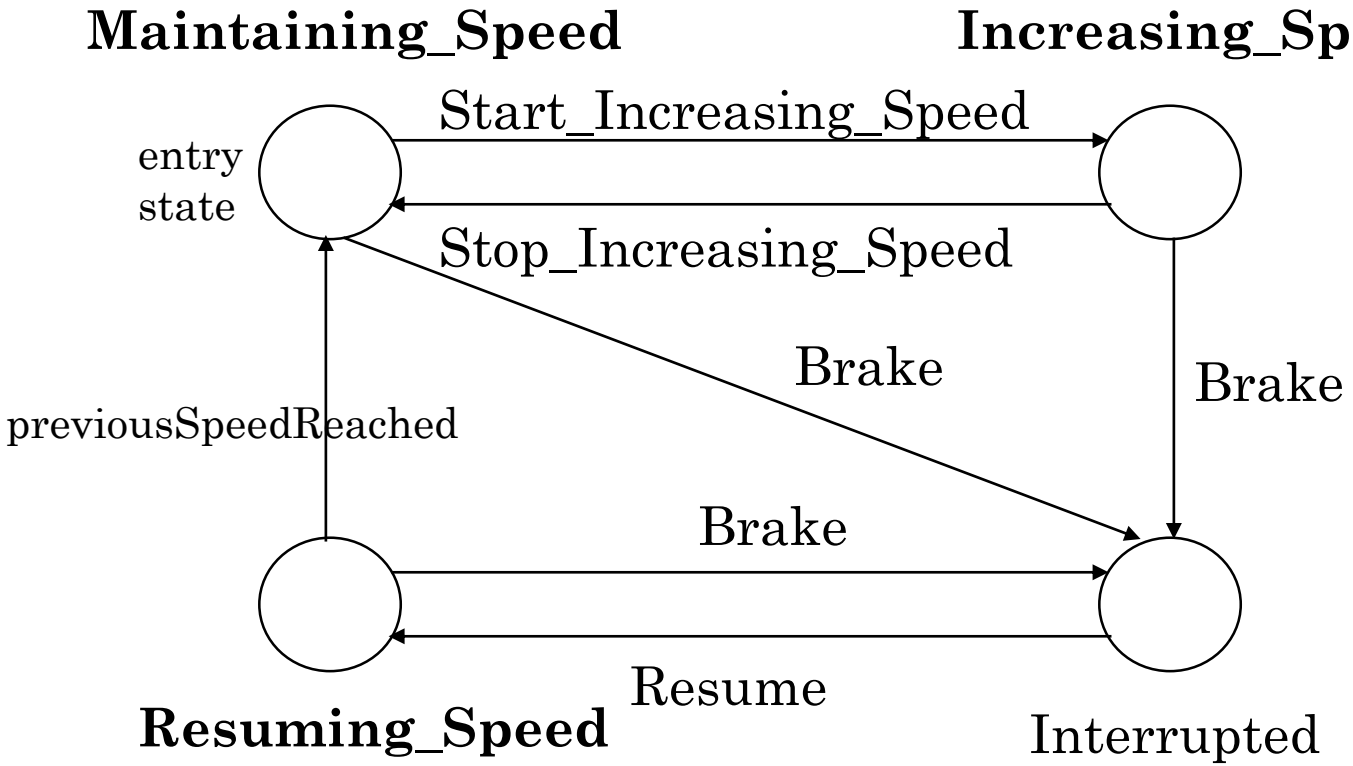
Cruise control

Il dispositivo è attivo quando il motore è acceso. Quando il guidatore preme il pulsante On, il dispositivo entra in funzione: registra la velocità corrente e la mantiene. In caso di frenata (Brake) il controllo è sospeso; il pulsante Resume consente di tornare alla velocità impostata in precedenza.

Due ulteriori pulsanti (Start_Increasing_Speed e Stop_Increasing_Speed) permettono l'accelerazione automatica.

Cruise control





Nei tre stati principali devono essere eseguiti periodicamente tre algoritmi: il mantenimento della velocità corrente (algoritmo 1), l'incremento della velocità (algoritmo 2), la ripresa della velocità precedente (algoritmo 3).

Note

Uno stato può avere

una *entry action*, che viene eseguita dopo una transizione entrante,
una *exit action*, che viene eseguita prima di una transizione uscente,
un'*azione interna*, che può anche essere eseguita periodicamente.

int targetSpeed; // velocità di crociera

Azioni degli stati

Maintaining_Speed

entry action: copia la velocità corrente in targetSpeed.

L'azione interna esegue periodicamente l'algoritmo 1.

Increasing_Speed

L'azione interna esegue periodicamente l'algoritmo 2.

Resuming_Speed

L'azione interna esegue periodicamente l'algoritmo 3.

Se la velocità corrente è uguale alla velocità target (o la supera), l'azione interna termina dopo aver emesso l'evento *previousSpeedReached*.

Strutture

Definiscono ad alto livello la *struttura* delle *notazioni* grafiche.

Con le regole di *validazione* si può definire se un modello (struttura) è valido (rispetto ai requisiti) oppure no.

Argomenti

activity diagrams

dataflow diagrams

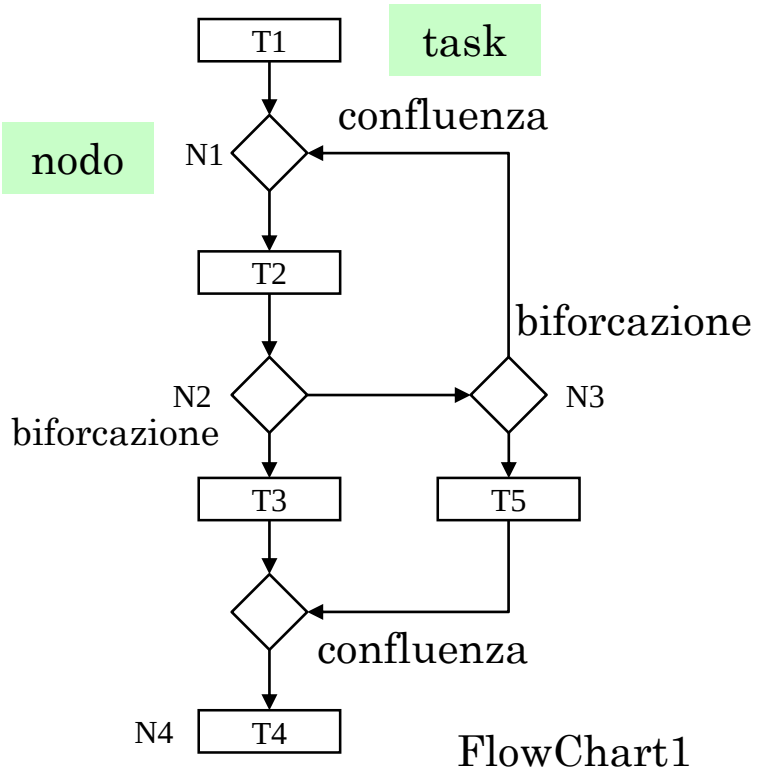
state models

Struttura di un activity diagrams (versione semplificata)

Si definisca una struttura per diagrammi flowchart di cui è dato un esempio in figura. Un diagramma ha un nome e contiene elementi di due tipi, **task** e **nodi**. Ogni elemento ha un nome. Gli elementi sono collegati mediante link orientati.

Valgono i vincoli seguenti.

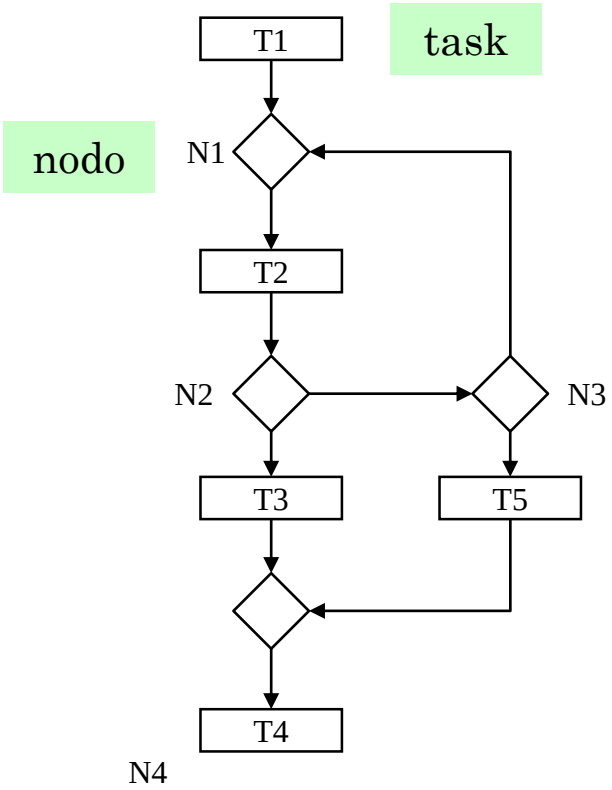
Di norma un task ha 1 input e 1 output; il task iniziale come T1 ha solo 1 output. Un task finale come T4 ha solo 1 input. Il diagramma contiene 1 solo task iniziale e 1 o più task finali. Non ci sono task privi sia di input sia di output. I nodi hanno 1 o più input e 1 o più output e almeno 3 tra input e output.



Note: si tratta di un grafo con 2 tipi di elementi.

Se un nodo ha due o più input e due o più output è sia una confluenza sia una biforcazione.

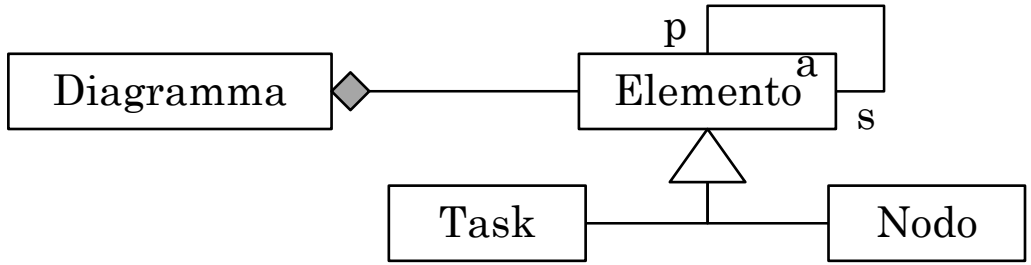
Analisi



FlowChart1

Si tratta di un *grafo orientato* composto da due tipi di elementi, nodi e task.

Conviene introdurre una classe astratta con precedenze.



p = predecessori, s = successori

Elemento è una classe astratta; lo stereotipo <<a>> è indicato semplicemente con a.

Task e Nodo rappresentano oggetti Task e oggetti Nodo.

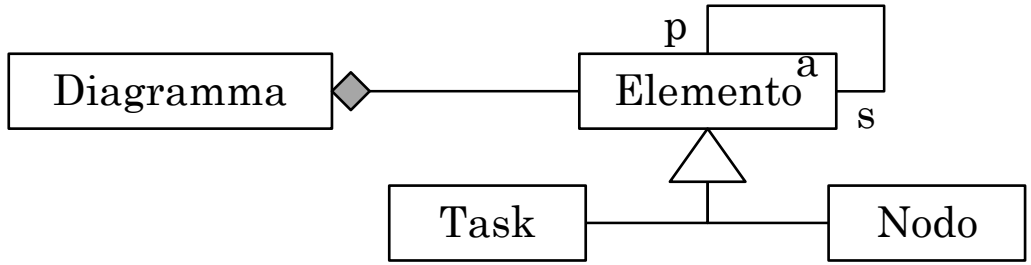
Attributi

Elemento: String name;

Diagramma: String name;

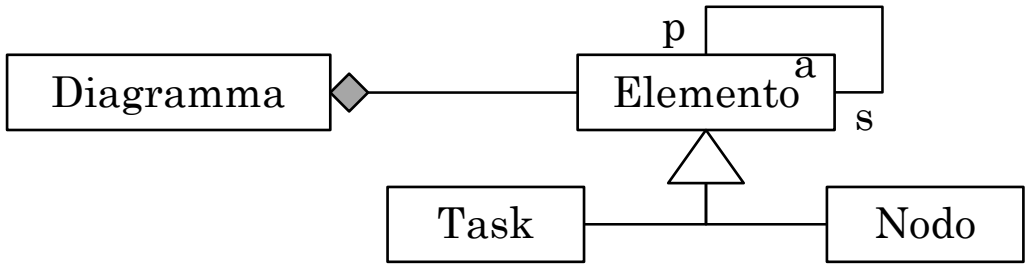
Valgono i vincoli seguenti.

1. Di norma un task ha 1 input e 1 output; il task iniziale come T1 ha solo 1 output. Un task finale come T4 ha solo 1 input. Non ci sono task privi sia di input sia di output.
2. Il diagramma contiene 1 solo task iniziale e 1 o più task finali.
3. I nodi hanno 1 o più input e 1 o più output e almeno 3 tra input e output.



Regole di validazione

1. [task.s] in 0..1. [task.p] in 0..1. [task.s] + [task.p] > 0.
2. [diagramma.tasks([p] == 0)] == 1. [diagramma.tasks([s] == 0)] >= 1.
3. [nodo.s] > 0. [nodo.p] > 0. [nodo.s] + [nodo.p] > 2.



Navigazioni nei modelli con inheritance

[diagramma.tasks([p] == 0)] == 1. [diagramma.tasks([s] == 0)] >= 1.

L'espressione diagramma.tasks è una semplificazione della seguente:

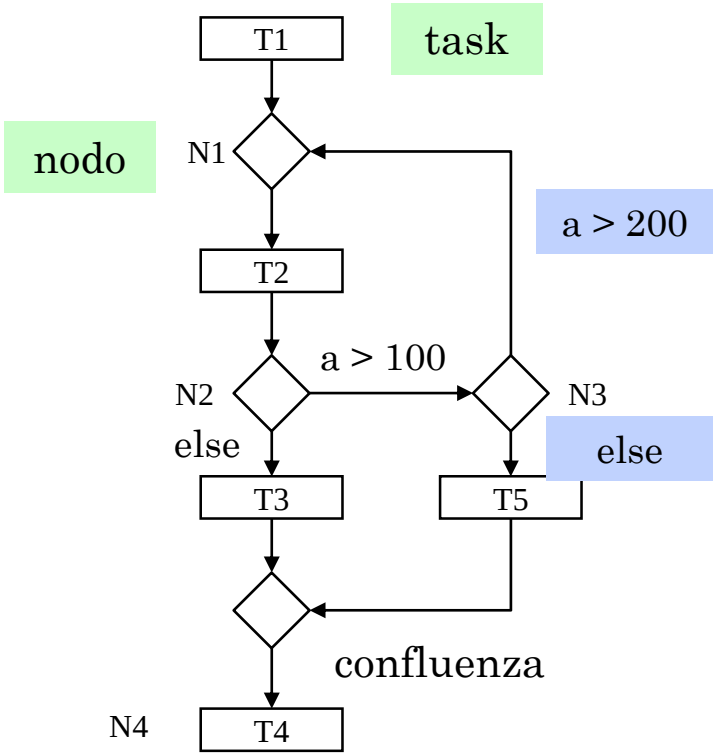
diagramma.elementi (e, e instanceof Task)

cioè tra gli elementi contenuti nel diagramma si considerano soltanto i task.

Estensione - struttura v.2

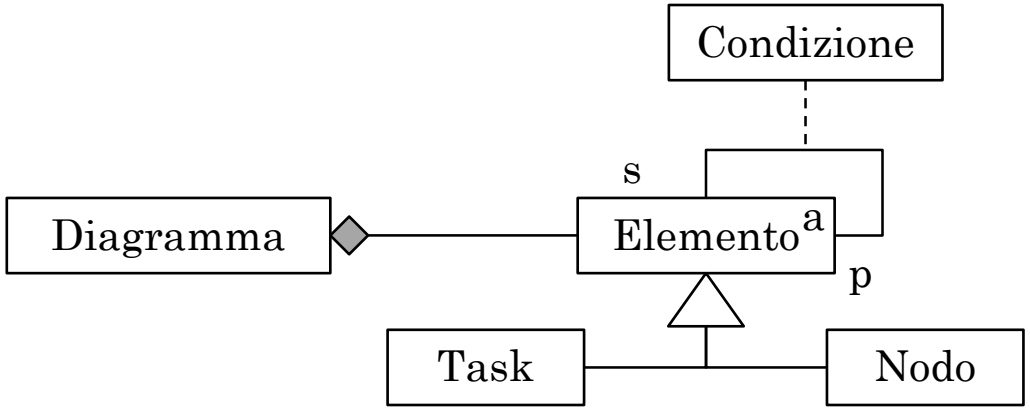
Se un nodo ha due o più archi di uscita, ciascuno di questi archi ha una condizione (String); *else* è assimilata ad una condizione.

Occorre introdurre gli attributi degli archi e quindi occorre introdurre gli archi.



FlowChart1

Struttura v.2



Attributi

Condizione: String c.

Regole di validazione

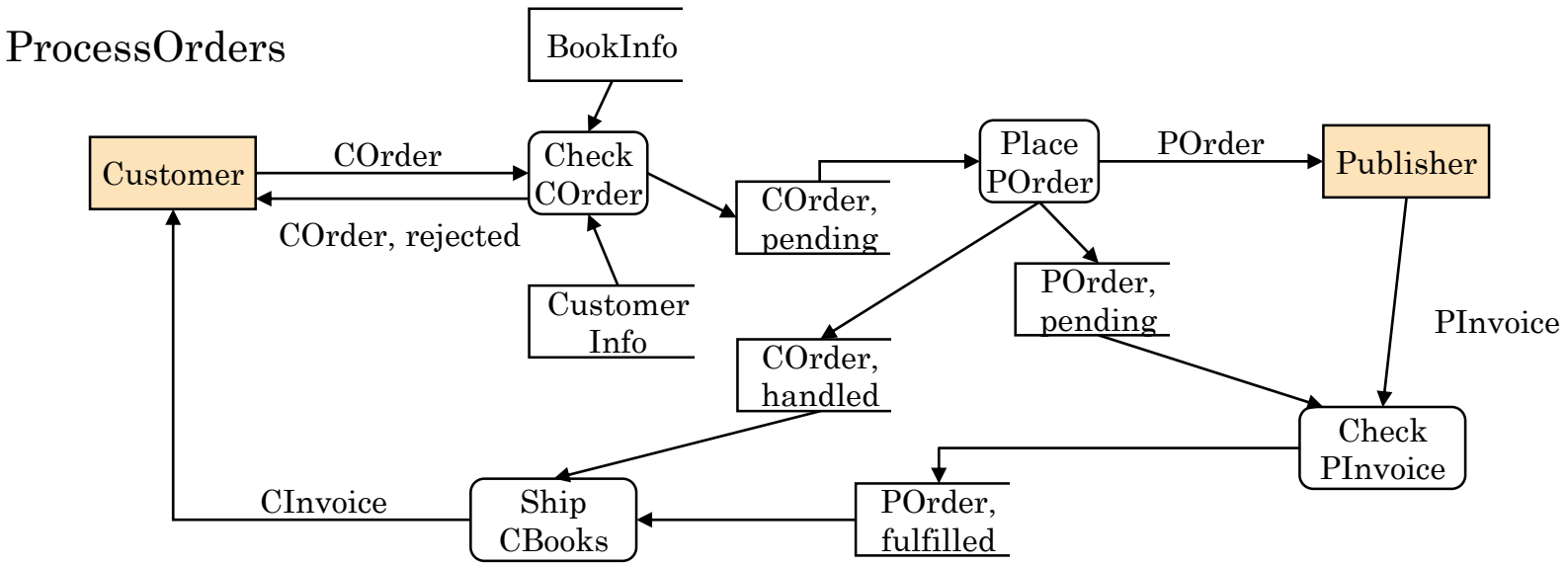
1. [task.s] in 0..1. [task.p] in 0..1. [task.s] + [task.p] > 0.
2. [diagramma.tasks([p] == 0)] == 1. [diagramma.tasks([s] == 0)] >= 1.
3. [nodo.s] > 0. [nodo.p] > 0. [nodo.s] + [nodo.p] > 2.

Regole sulla presenza delle condizioni

[nodo.s] > 1 → nodo.condizioni.c **def**

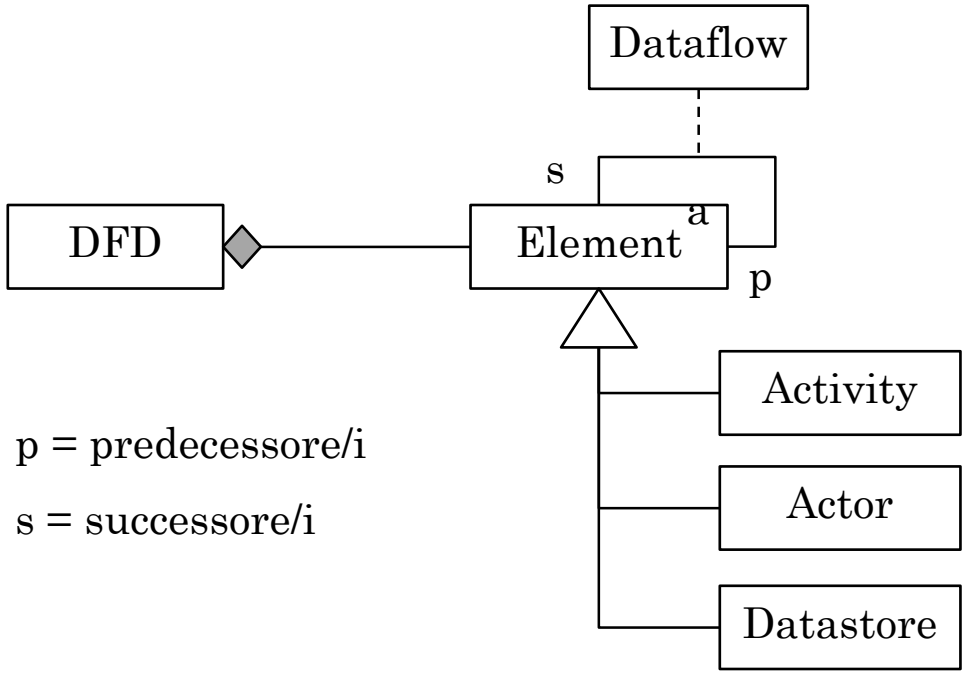
[nodo.s] == 1 → nodo.condizione.c == null oppure nodo.condizione == null

Struttura dei dataflows



Un DFD ha un nome e contiene attori esterni, actitivies, datastores e dataflows.
I collegamenti non ammessi sono i seguenti: attore → attore, attore → datastore, datastore → datastore, datastore → attore.
Un attore ha almeno 1 collegamento, un datastore ha almeno 1 collegamento, una activity ha almeno 1 collegamento di input e 1 di output.
Attori e attività hanno un nome, i datastore hanno un nome e uno stato (opzionale), i dataflow tra attività e quelli tra attori e attività hanno due etichette (tipo e stato) di cui la seconda può essere nulla.

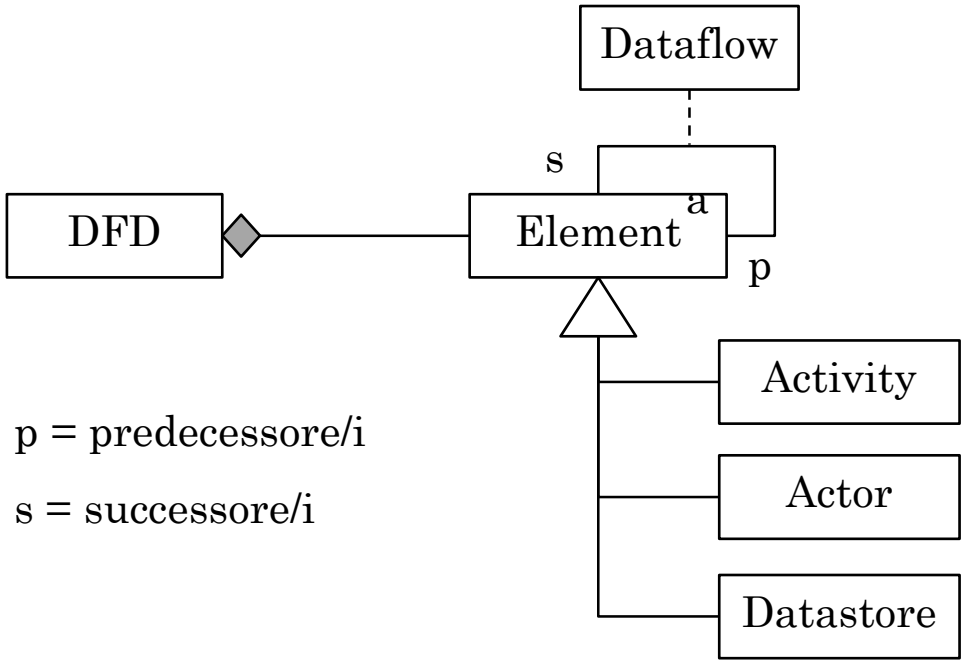
Struttura



Attributi
DFD: String name.
Element: String name.
Datastore: String state.
Dataflow: String type, String state.

Le classi Activity, Actor e Datastore rappresentano oggetti Activity, Actor e Datastore, rispettivamente.

Vincoli sui collegamenti



I collegamenti non ammessi sono i seguenti:

attore → attore, attore → datastore,

datastore → datastore,

datastore → attore.

In breve, attori e datastore sono collegati soltanto ad attività.
actor.s *instanceof Activity*; actor.p instanceof Activity; idem per datastore;
instanceof Activity è un predicato: actor.s instanceof Activity è vero se i successori dell'attore sono attività.

Un attore ha almeno 1 collegamento, un datastore ha almeno 1 collegamento, una activity ha almeno 1 collegamento di input e 1 di output.
[actor.p] + [actor.s] > 0; idem per datastore; [activity.p] > 0; [activity.s] > 0;

Vincoli sulle etichette

//un datastore ha i dataflow entranti e uscenti privi di etichette

datastore.dataflows.type == null; datastore. dataflows.state == null.

Significato: i tipi e gli stati dei dataflow entranti sono nulli.

//i dataflow di un actor hanno almeno l'etichetta di tipo

actor.dataflowstype def.

//un dataflow che collega due attività ha almeno l'etichetta di tipo

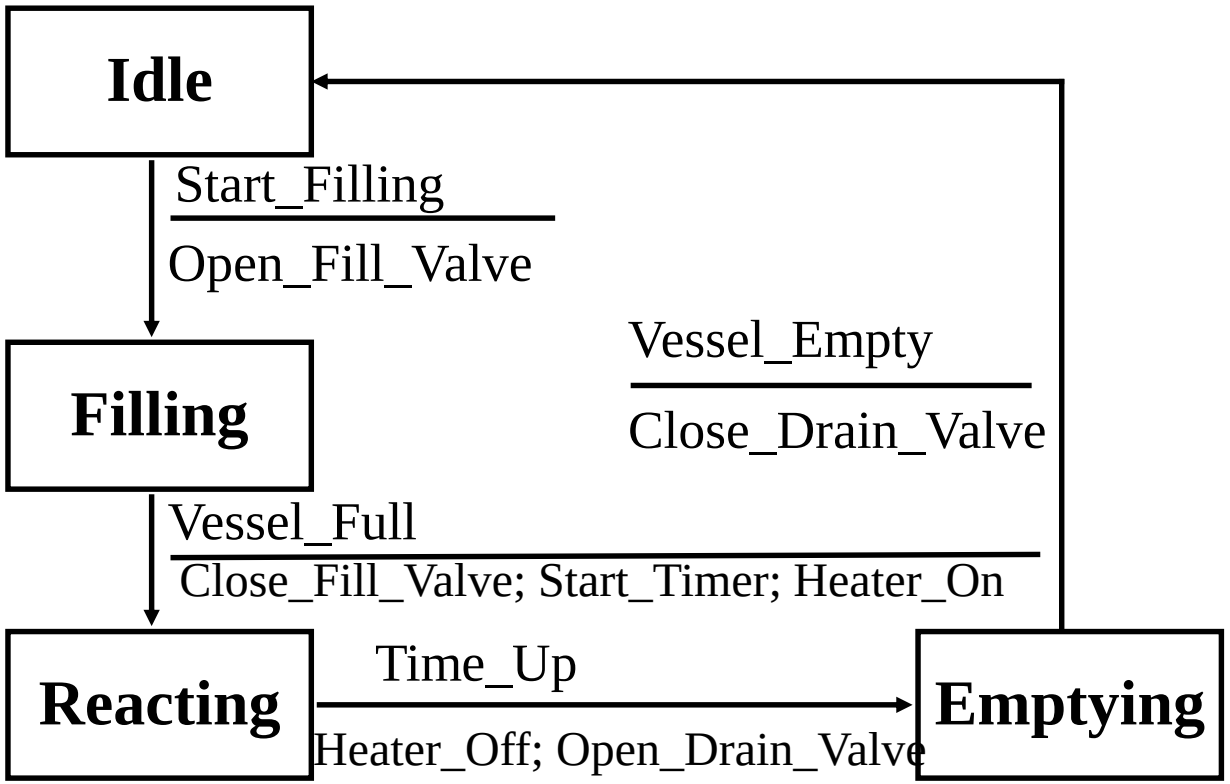
dataflow.s instanceof Activity and dataflow.p instanceof Activity → dataflow.type def.

Struttura dei modelli di stato

Si definisca la struttura dei modelli di stato con le caratteristiche seguenti.

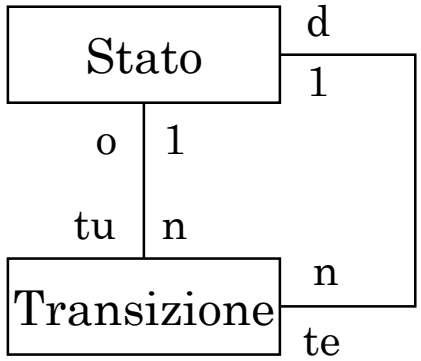
Gli stati hanno un nome, delle transizioni entranti (almeno una) e delle transizioni uscenti (almeno una).

Gli eventi delle transizioni relative allo stesso stato devono avere nomi distinti.



Struttura

La classe Stato rappresenta oggetti Stato, la classe Transizione rappresenta oggetti Transizione.



Attributi

Stato: String nome;

Transizione: String evento, String comando.

Regole

Stato: [te] > 0 and [tu] > 0.

tu = transizioni uscenti
te = transizioni entranti
o = origine
d = destinazione

Gli eventi delle transizioni relative allo stesso stato devono avere nomi distinti.

stato.tu.evento distinct

Patterns

Patterns

Reusable solutions to recurring problems (Christopher Alexander)

Software patterns were motivated by Christopher Alexander's work on patterns in architecture.

A PATTERN LANGUAGE, Oxford University Press, New York, 1977.

"Each 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".

The book presents 253 patterns for regions, towns, houses, gardens and rooms.

Design patterns

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides ("Gang of Four").

Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

"It's a book of design patterns that describes simple and elegant solutions to specific problems in object-oriented software design.

Design patterns capture solutions that have developed and evolved over time. ..

Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems. Our goal is to capture design experience in a form that people can use effectively."

Creational patterns: factory method, builder, prototype, singleton …

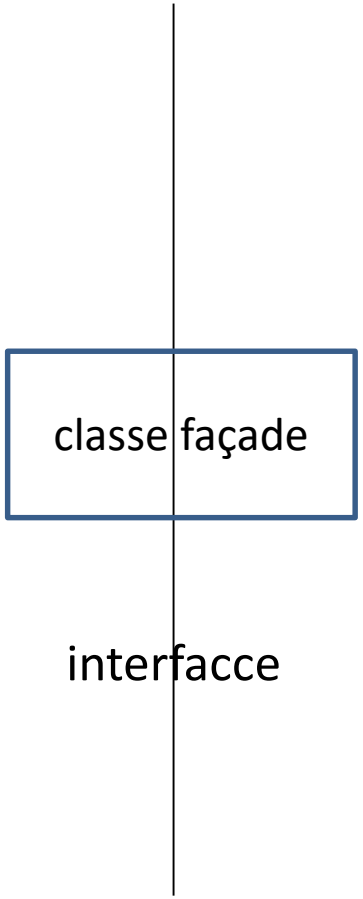
Structural patterns: façade, composite …

Behavioral patterns: iterator, command, observer …

Façade

package applicativo

Le classi di questo package agiscono sugli oggetti di libreria in modo controllato (attraverso la classe di facciata).



package di libreria

Gli oggetti sono generati dai metodi (factory) della classe principale (facciata).
La classe principale contiene le collezioni degli oggetti di libreria.

Factory method

Obiettivo: generare un nuovo oggetto senza chiamare direttamente un costruttore.

Inserimenti
La biblioteca può avere più copie (volumi) dello stesso libro.
Il metodo **addLibro** (titolo, nVolumi, autori) inserisce un libro e i volumi corrispondenti.

Il metodo **addUtente** (nome, maxPrestiti, durata) inserisce un utente con il nome, il n. max di prestiti che può avere nello stesso periodo, la durata (massima) in giorni dei suoi prestiti.

Builder

Intent: to separate the construction of a complex object from its final representation; e.g. StringBuilder and String.

Example
A car has a number of options, such as the number of seats, the presence of a trip computer/a bluetooth kit.

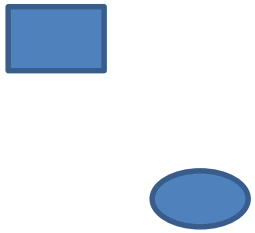
Class **CarBuilder** enables you to configure a car by choosing the options.
When you have finished, you get the configuration of the car with the price.

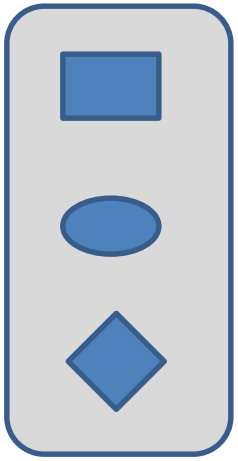
CarBuilder carBuilder = new CarBuilder();
carBuilder.setNofSeats(5); carBuilder.setTripComputer();
Car car = carBuilder.getResult();

This pattern is frequently used to build products made up of components.

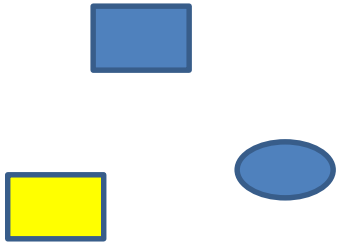
Prototype

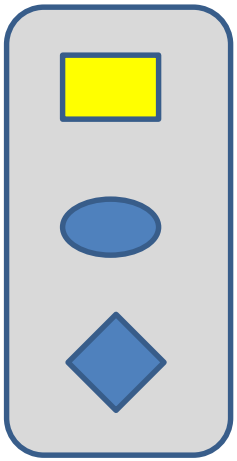
Intent: to get a new object by making a copy of a prototypical instance. The class of the prototype implements the **clone** method.





A diagram is made up of objects similar to those included in a palette.
The objects in the palette are prototypes. If the user modifies the properties of a prototype the new instances will be different from the previous ones.





Singleton pattern

Garantisce che di una classe esista un'unica istanza.

Serve, ad esempio, per la gestione di dispositivi.

Esempio

```
public class Dispositivo {
private Dispositivo() {} // nota: costruttore privato
private static Dispositivo dispositivo = new Dispositivo();
public static Dispositivo getDispositivo() {return dispositivo;}
private boolean attivo = false;

public void start () {
        if (! attivo) {System.out.println("start"); attivo = true;}
}
public void stop () {
        if (attivo) {System.out.println("stop"); attivo = false;}
}
}
```

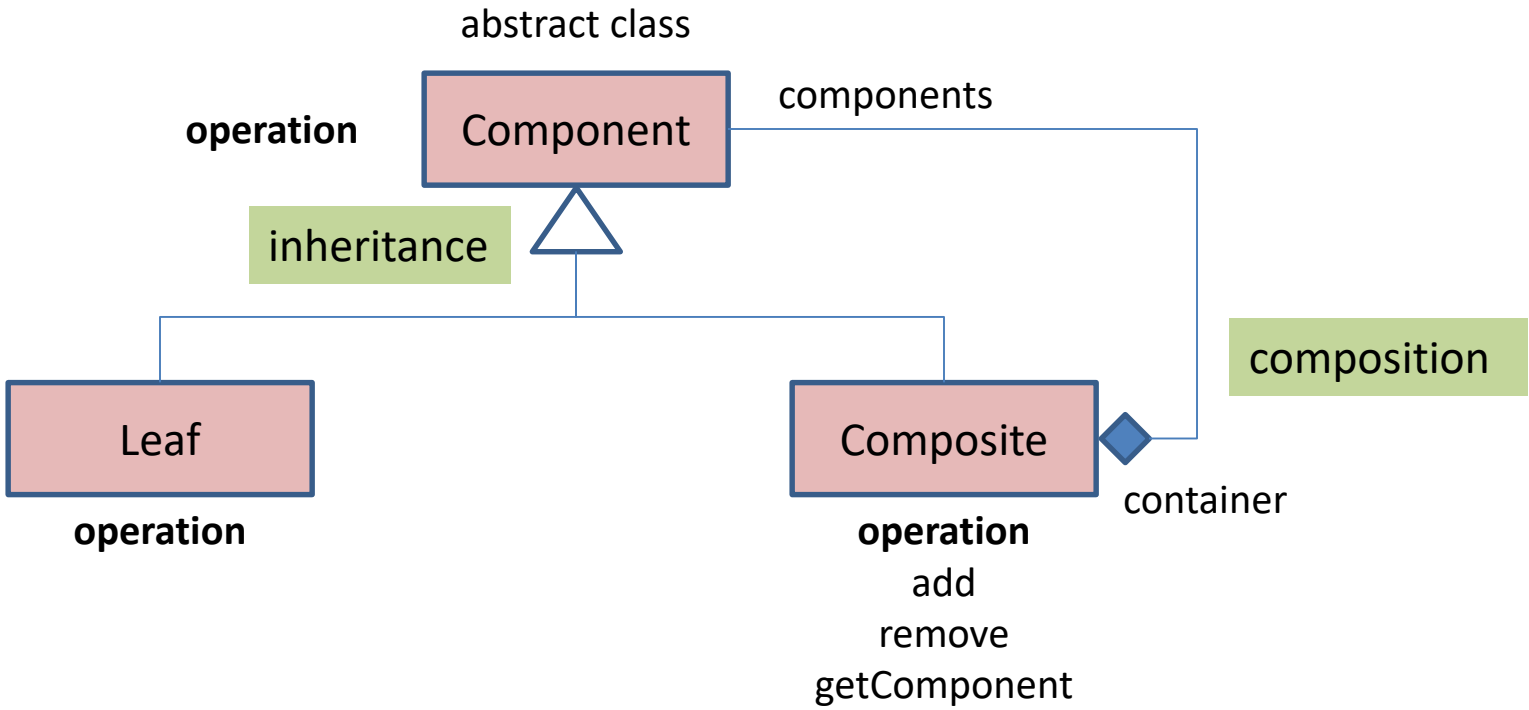
Esempio

```
public static void main(String[] args) {
Dispositivo d = Dispositivo.getDispositivo();
d.start(); d.start(); d.stop(); d.stop();
d.start(); d.stop();
}
```

```
start
stop
start
stop
```

Composite

Intent: to handle part-whole hierarchies by treating simple objects and compound ones uniformly.



Iterator

Intent: to provide a way to access the elements of a collection sequentially without exposing the underlying representation.

Implemented in Java by Iterator and ListIterator objects.

Command

Intent: to encapsulate the request for an operation into an object.

Example
A client generates a command and passes it to an invoker; the invoker sets the command performer and passes it the command.
In alternative, the performer may be set by the client.
In addition, the client may be notified with the result.

Observer

Intent: to enable a number of observers to be notified when a subject changes state.
Also known as *publish/subscribe*.

The subject provides a method to add/remove observers.
The subject knows the observers through an interface containing the notify method.

In java swing observers are called *event listeners*.

Esempio d'uso del pattern Observer

Ci sono agenzie d'informazione e utenti. Le agenzie pubblicano notizie e le inviano ai loro iscritti (utenti). Per ricevere notizie gli utenti si devono iscrivere presso le agenzie.

Le classi sono Subject e Observer.

Gli observer sono noti ai subject tramite l'interfaccia ObserverI.

```
public interface ObserverI {
        void notify (String news);
}
```

Class Observer

```
public class Observer implements ObserverI {
        private String name;
        public Observer (String name) {
                this.name = name;
        }
        public void notify (String news) {
                System.out.println(name + ": " + news);
        }
}
```

Class Subject + main

```
public class Subject {
        ArrayList<ObserverI> observerList = new ArrayList<>();
        public void addObserver(ObserverI observer) {
                observerList.add(observer);}
        public void publish(String news) {
                observerList.forEach(o -> {o.notify(news);});}

public static void main(String[] args) {
Observer john = new Observer("john");
Observer mary = new Observer("mary");
Subject agency = new Subject();
agency.addObserver(john); agency.addObserver(mary);
agency.publish("breaking news");
}}
```

```
john: breaking news
mary: breaking news
```

Domanda

Perché conviene definire l'interfaccia ObserverI?

Domanda

Perché conviene definire l'interfaccia ObserverI?

Per poter avere observers di classi diverse.

Livelli dei pattern

Da specifici a generici

programmazione

come risolvere un particolare problema; ad es. leggere tutto un file testuale linea dopo linea.

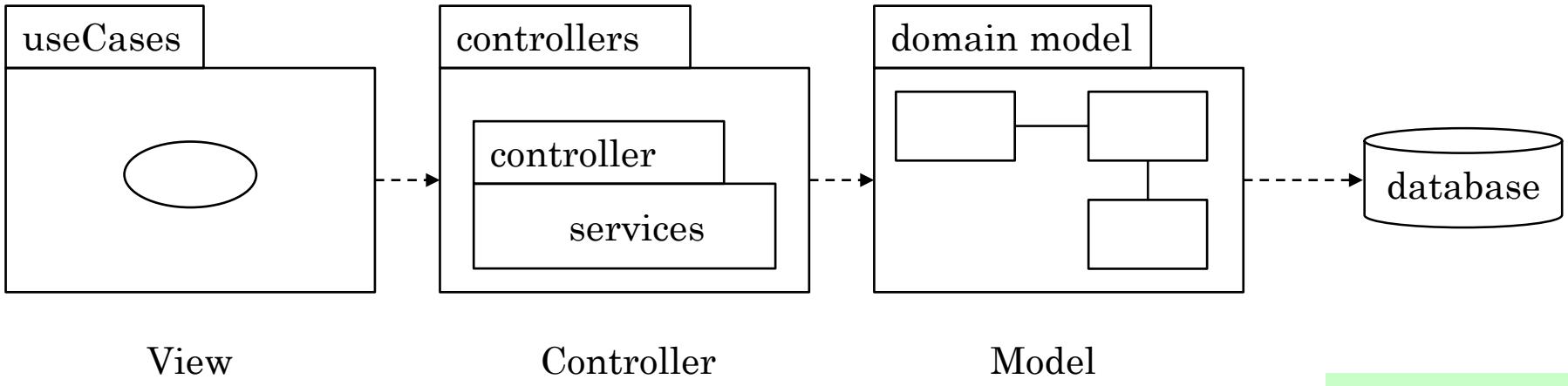
progetto (design)

schema di classi e interfacce per ottenere un certo comportamento; ad es. porre in relazione fonti di notizie con entità interessate a riceverle.

architettura

schema di sottosistemi e intercomunicazioni per ottenere un certo comportamento; ad es. il pattern MVC (Model View Controller) per applicazioni con interfacce grafiche.

Pattern Model View Controller (MVC)



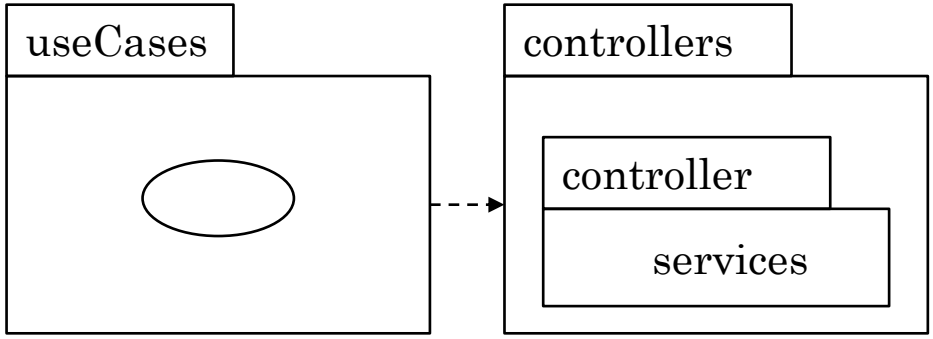
Schema logico

L'implementazione del *model* gestisce la persistenza dei dati e agisce sul database con operazioni CRUD.

I *controllers* verificano la correttezza delle richieste e le smistano ai servizi idonei.

La *view* rappresenta l'interfaccia utente.

Package e dipendenze (UML)



package

dipendenza