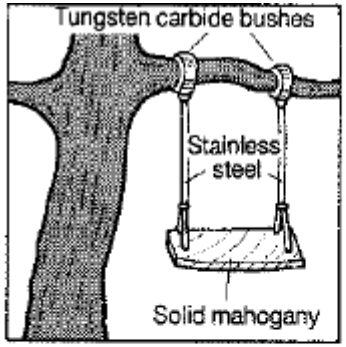
Requisiti

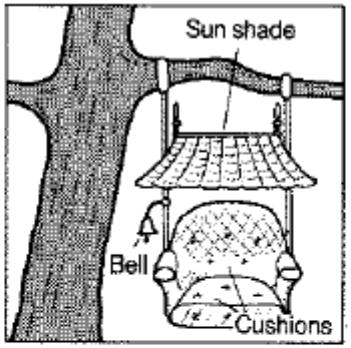
Ingegneria del software

Vincenzo Bonnici
Corso di Laurea in Informatica
Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Università degli Studi di Parma

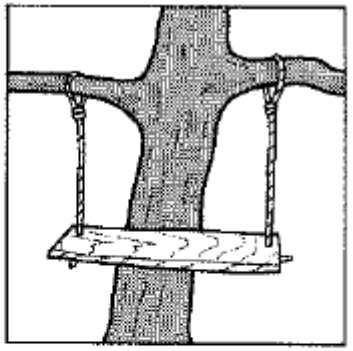
2025-2026



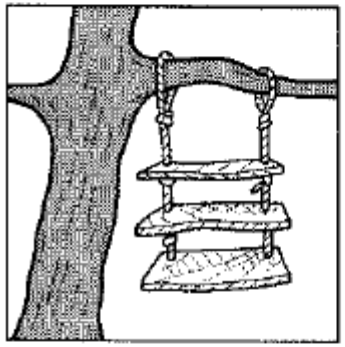
What Product Marketing specified



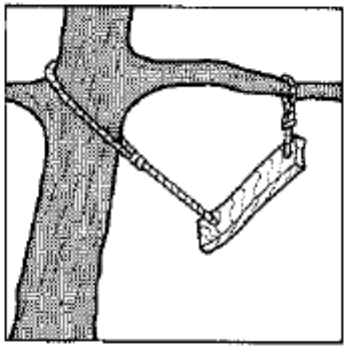
What the salesman promised



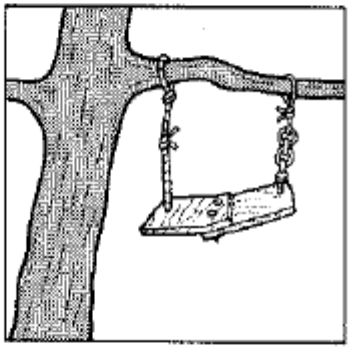
Design group's initial design



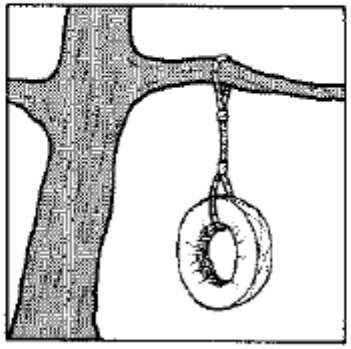
Corp. Product Architecture's modified design



Pre-release version



General release version



What the customer actually wanted

Analisi e specifica dei requisiti

- **L'analisi dei requisiti** è il processo di comprensione di ciò che è richiesto a un sistema.

- L'analisi dei requisiti comporta lo studio del dominio applicativo, in modo che gli sviluppatori di software possono comprendere la relazione del software con il suo ambiente.

- Una **specifica dei requisiti** è il risultato dell'analisi dei requisiti processo, cioè un documento che fornisca una descrizione precisa e completa del fabbisogno.

- Il processo di analisi si traduce nella costruzione di un modello, cioè di una astrazione degli aspetti rilevanti del sistema e del suo ambiente.

- L'utilizzo di un metodo di analisi specifico aiuta a realizzare questo modello e le sue sottostanti ipotesi più **esplicite** e **verificabili**.

- **Definizioni**
  - Una capacità del software, che l'utente necessita per risolvere un problema o per ottenere un risultato
  - Una capacità che il software deve avere per soddisfare un contratto, uno standard, una specifica

- Un requisito va da una descrizione con alto livello di astrazione di un servizio o di un vincolo ad una specifica funzionale dettagliata matematicamente

- I requisiti possono servire
  - Come base per una offerta per un contratto
    - Devono essere sufficientemente astratti per non indicare una soluzione predefinita
  - Come base per il contratto
    - Devono essere sufficientemente dettagliati
  - Per descrivere ciò che è richiesto agli sviluppatori
    - Devono essere molto dettagliati

Tipi di requisiti

- Si dividono in **funzionali** e **non-funzionali**

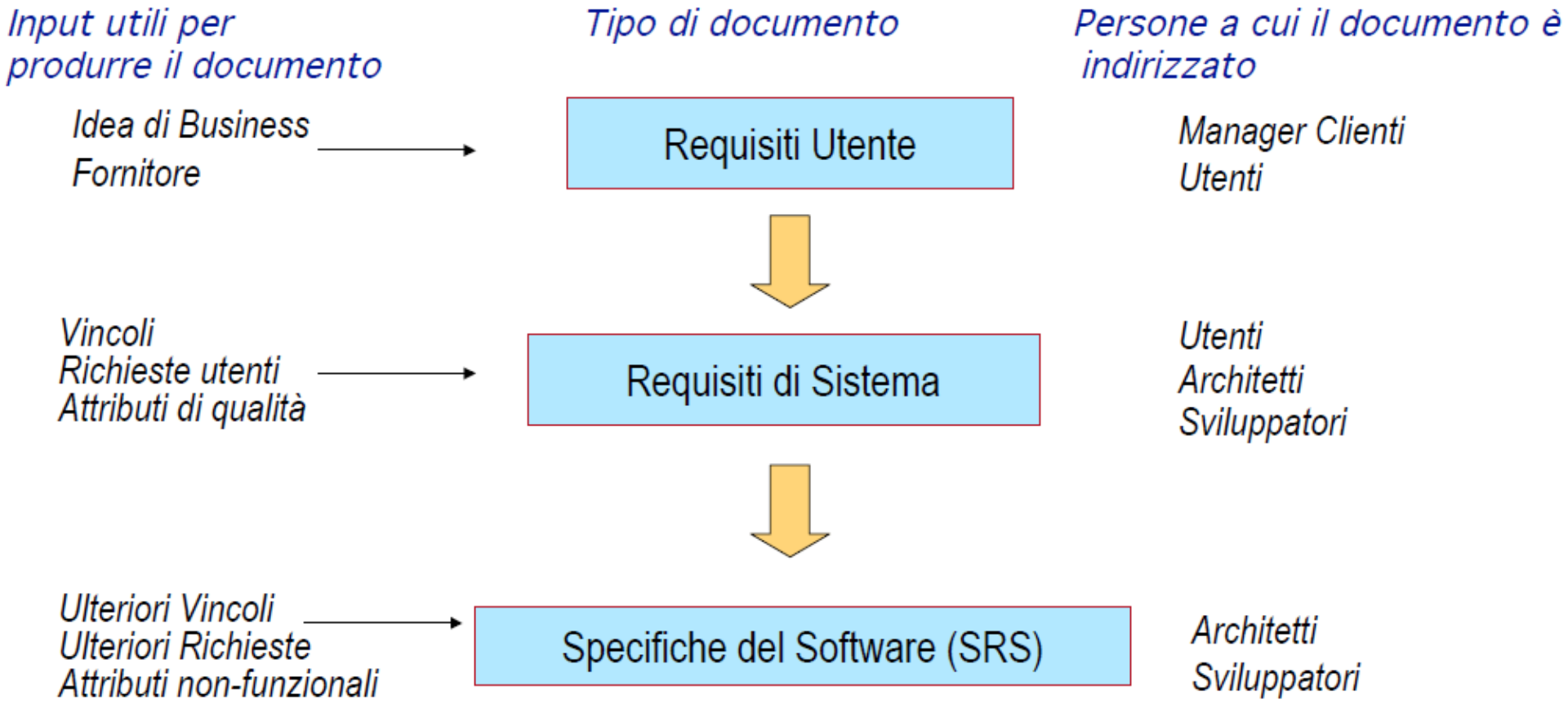
- Esempi di requisiti **funzionali**
  - Ad ogni ordine corrisponderà un unico identificatore che l'utente potrà usare per accedere ad un ordine
  - Il sistema fornirà strumenti per la visualizzazione degli ordini immagazzinati

- I **non funzionali** possono essere più critici di quelli funzionali
  - Se non sono soddisfatti, il sistema è inutile
  - Di prodotto: specificano il comportamento del prodotto, es. velocità, affidabilità, etc.
  - Organizzativi: quelli derivanti da prassi organizzative, es. metodi di design o linguaggi usati, etc.
  - Esterni: derivanti da fattori esterni, es. interoperabilità con altri sistemi, aderenza alle leggi, fattori etici, etc.

Tipi di requisiti

- I requisiti devono essere completi e consistenti
  - **Completi**: includere la descrizione di tutto ciò che è richiesto
  - **Consistenti**: non ci devono essere contraddizioni nella loro descrizione
  - In pratica è difficile ottenere un documento con entrambe le caratteristiche

Livelli di requisiti

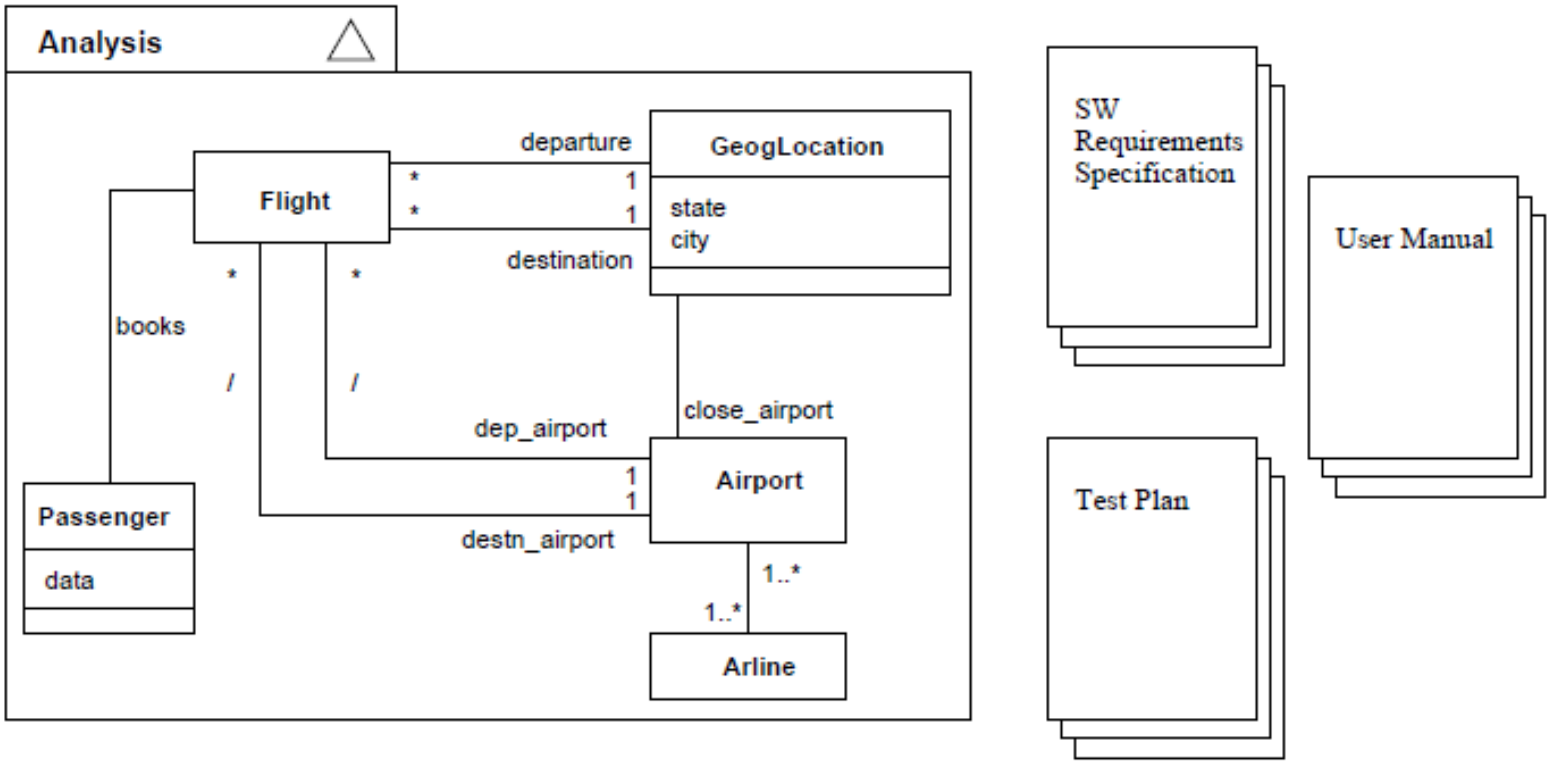


Tipi di requisiti

- Requisiti **utente**
  - Descrizioni in linguaggio naturale e diagrammi dei servizi che il sistema fornisce.
  - Scritto per i clienti
    - Il software dovrà fornire il modo per rappresentare ed accedere file esterni creati con altri tool

- Requisiti **di sistema**
  - Un documento strutturato che dettaglia i servizi del sistema. Scritto come contratto tra cliente e fornitore
    - L'utente dovrebbe essere fornito di strumenti per definire il tipo di file esterni
    - Ogni tipo di file esterno può avere associato un tool che può essere applicato al file
    - Ogni tipo di file esterno può essere rappresentato da una icona specifica
    - La selezione dell'icona provoca l'applicazione del tool al file rappresentato

- **Specifiche del software** (SRS: Software Requirements Specification)
  - Descrizione dettagliata del software che serve come base per il design o l'implementazione. Scritto per gli sviluppatori



La fase di analisi produce un modello di analisi.

Il modello di analisi è definito dal documento Software Requirements Specification (specifica dei requisiti del software).

Altri risultati includono il piano di test e i manuali utente.

- Un **requisito** è un'affermazione su un aspetto rilevante di un sistema che deve essere sviluppato: un servizio che deve fornire, una proprietà o un vincolo che deve soddisfare, e così via.

- Alcuni requisiti possono riguardare il processo di sviluppo in sé, piuttosto che il sistema; Ad esempio, potrebbe essere necessario che il sistema sia sviluppato seguendo determinate procedure standard.

- I requisiti **funzionali** descrivono i servizi offerti dal sistema in termini di relazioni tra input e output, o tra stimoli e risposte (sistemi reattivi).

- Un sistema che soddisfa i suoi requisiti funzionali si dice **corretto**

- I requisiti **non funzionali** descrivono le proprietà o i vincoli del sistema o del processo

- Alcuni requisiti non funzionali sono l'affidabilità, la robustezza, la sicurezza e le prestazioni.

Requisiti - tracciabilità

- I requisiti devono essere **riconducibili** alla progettazione; il design dovrebbe essere **tracciabile** al codice; I requisiti, la progettazione e il codice devono essere **riconducibili** ai test.
- La **tracciabilità** dovrebbe essere mantenuta quando vengono apportate **modifiche**.
- La tracciabilità dovrebbe avvenire nella direzione opposta, per garantire che non siano state create **funzioni indesiderate**.
- Ogni requisito e ogni caratteristica di progettazione devono essere espressi in modo tale che possa essere effettuato un **test** per determinare se tale caratteristica è stata implementata correttamente.
- Sia i requisiti funzionali che quelli non funzionali dovrebbero essere **testabili**.
- I risultati dei test devono essere riconducibili ai requisiti associati.

IAEA Safety Guide Software for Computer Based Systems Important to Safety in Nuclear Power Plants, NS-G-1.1

Vincoli

- I **vincoli** sono requisiti **non funzionali** che impongono limitazioni alle scelte di progettazione.

- I vincoli di tempo sono particolarmente importanti nei sistemi concorrenti, come i sistemi di controllo, i sistemi di transazione (database distribuiti), ecc.

  - I vincoli di **sincronizzazione** impongono un certo ordinamento tra gli eventi, senza specificare limiti sugli intervalli di tempo tra gli eventi.
    - Es.: "La valvola B non deve essere aperta prima della valvola A".

  - I vincoli in **tempo reale (real-time)** impongono limiti agli intervalli di tempo tra gli eventi.
    - Es.: "La valvola B deve essere aperta tra 10 e 20 secondi dopo la valvola A".

Si noti che un sistema in tempo reale non è necessariamente un sistema veloce. La prevedibilità è più importante della velocità.

Requisiti non funzionali

- **sicurezza** (safety): Capacità di funzionare senza arrecare danni a persone o cose (piú precisamente, con un rischio limitato a livelli accettabili). Si usa in questo senso anche il termine innocuità.

- **riservatezza** (security): Capacità di impedire accessi non autorizzati ad un sistema, e in particolare alle informazioni in esso contenute. Spesso il termine sicurezza viene usato anche in questo senso.

- **robustezza**: Capacità di funzionare in modo accettabile anche in situazioni non previste, come guasti o dati di ingresso errati.

- **prestazioni**: Uso efficiente delle risorse, come il tempo di esecuzione e la memoria centrale.

- **disponibilità**: Capacità di rendere disponibile un servizio continuo per lunghi periodi.

- **usabilità**: Facilità d'uso.

- **interoperabilità**: Capacità di integrazione con altre applicazioni.

Requisiti non funzionali

- La **dependability**, o **affidabilità** in accezione informale, è un requisito che combina requisiti piú specifici, fra cui:
  - sicurezza;
  - riservatezza;
  - robustezza.
- N.B.: in senso stretto, l'affidabilità (**reliability**) è la probabilità che non avvengano malfunzionamenti entro determinati periodi di tempo.

- **Prevenzione dei guasti**: la qualità del software è la prima linea di difesa.
  - Processo di sviluppo rigoroso.
  - Test, verifica e convalida.
  - Competenza ed esperienza.

- **Tolleranza ai guasti**: le disposizioni specifiche progettate nel software sono la seconda linea di difesa, contro errori di progettazione (o codifica) non rilevati e incidenti imprevisti.
  - Conformità ai requisiti di sicurezza e progettazione a livello di sistema.
  - Gestione delle eccezioni a livello di sistema.
  - Scelta di meccanismi tolleranti ai guasti (ridondanza, diversità, schemi di voto…).
  - Problemi specifici dei sistemi informatici (interrupt, gestione della memoria, orologi…).

Convalidare i requisiti

- Ha lo scopo di mostrare che i requisiti definiscono il sistema che il cliente vuole
- Gli errori nei requisiti costano e quindi la convalida è importante
  - Aggiustare un errore nei requisiti dopo la consegna del sistema costa fino a 100 volte il costo di aggiustare errore nell'implementazione

- **Validità** Il sistema fornisce le funzioni che meglio supportano le necessità delle varie classi di utenti?
- **Consistenza** Ci sono conflitti tra i requisiti, o descrizioni differenti della stessa funzione?
- **Completezza** Sono incluse tutte le funzioni richieste dagli utenti?
- **Realismo** I requisiti possono essere implementati con il budget, le tecnologie e nel tempo a disposizione?
- **Verificabilità** Sarà possibile mostrare che il sistema soddisfa i requisiti?

Tecniche di convalida dei requisiti

- **Revisione** requisiti
  - Analisi sistematica dei requisiti per scoprire anomalie e omissioni

- **Prototipazione**
  - Uso di un modello (prototipo) del sistema per verificare i requisiti (isolando alcuni aspetti)
    - Può servire per dimostrazioni con utenti e clienti

- Generazione **test**
  - Sviluppare test dei requisiti per controllarli

- Analisi di **consistenza automatica**
  - Controllo della consistenza di una descrizione dei requisiti strutturata o formale

Classi di sistemi secondo i requisiti temporali

- **sequenziali**: senza vincoli di tempo;

- **concorrenti**: con sincronizzazione fra processi;
  - **centralizzati**: eseguiti in time-sharing su un unico agente di calcolo;
  - **distribuiti**: eseguiti in parallelo su piú agenti di calcolo;

- **in tempo reale**: con tempi di risposta prefissati. Possono essere sequenziali, piú spesso concorrenti.

Classi di sistemi secondo l tipo di elaborazione

- **orientati ai dati**: mantengono e rendono accessibili grandi quantità di informazioni (p.es., banche dati, applicazioni gestionali);

- **orientati alle funzioni**: trasformano informazioni mediante elaborazioni complesse (p.es., compilatori);

- **orientati al controllo**: interagiscono con l'ambiente, modificando il proprio stato in séguito agli stimoli esterni (p.es., sistemi operativi, controllo di processi).

Nello specificare un sistema è comunque necessario prendere in considerazione
tutti questi tre aspetti.

Linguaggi di specifica: grado di formalità

- **formali**: sintassi (testuale o grafica) e semantica definite rigorosamente;
  - non sono ambigui;
  - sono verificabili;
  - tradurre da linguaggio naturale a linguaggio formale chiarisce le idee (anche la programmazione è una traduzione).

- **semiformali**: sintassi (generalmente grafica) rigorosa, semantica approssimativa;
  - aiutano a schematizzare i concetti;
  - sono espressivi.

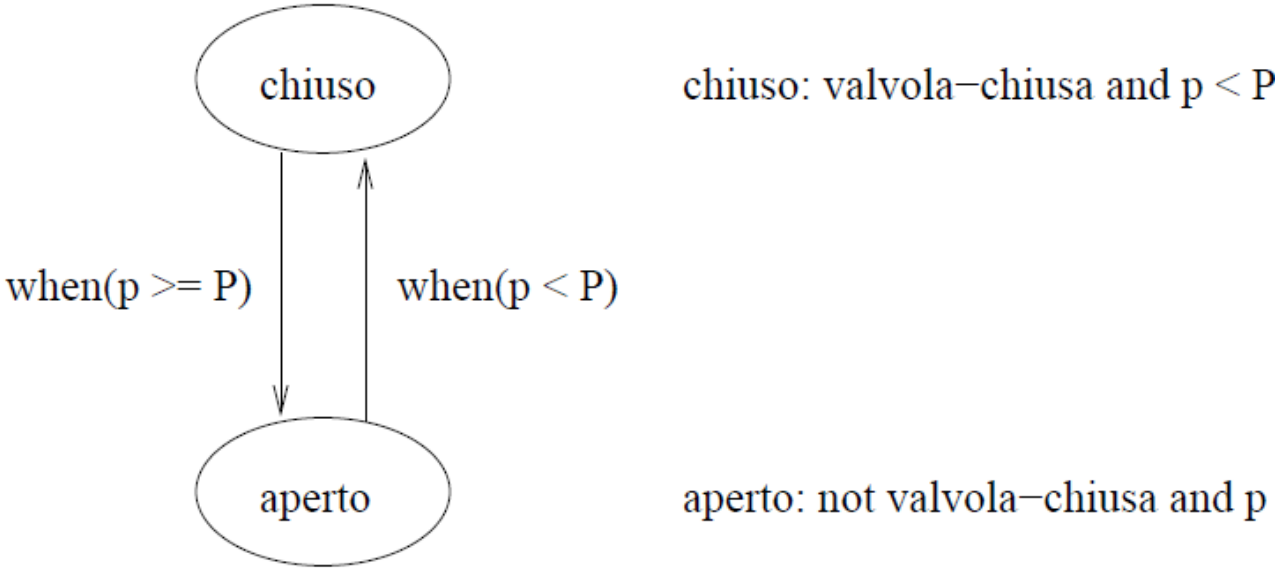
- **informali**: sintassi e semantica approssimative (linguaggio naturale).
  - sono ambigui,
    - **ambiguità del linguaggio naturale**
      - ogni uomo ama una donna: semantica
      - time flies like an arrow: semantica
      - ho visto Maria nel bosco con gli occhiali: sintattica

- **descrittivi**: entità, proprietà, relazioni, senza riferimento al tempo;
- **operazionale**: stati e transizioni, evoluzione del sistema.

- esempio: pentola a pressione
- La valvola deve essere aperta quando la pressione è maggiore di P. chiusa se minore.
  - stile descrittivo:

$$p < P \Leftrightarrow \textbf{valvola-chiusa}$$

  - stile operazionale:



Linguaggi di specifica: automi a stati finiti

- Gli **automi deterministici a stati finiti** sono la classe piú semplice delle macchine a stati.

- Due fra i campi di applicazione piú noti sono l'elettronica digitale e i compilatori.

- **stati**: condizioni in cui l'automa permane per un certo tempo;
- **ingressi**: stimoli prodotti dall'ambiente a istanti discreti;
- **transizioni**: passaggi da uno stato all'altro, in risposta a ingressi;
- **uscite**: azioni dell'automa
- **associate agli stati**: macchine di Moore;
- **associate alle transizioni**: macchine di Mealy.

- Oss.: nel campo dei compilatori, gli ingressi corrispondono alla lettura di simboli (gli ASF sono associati alle espressioni regolari e alle grammatiche non contestuali).

Linguaggi di specifica: automi a stati finiti

- Un ASF (di Mealy) è definito da una sestupla
  - < S, I,U, d, t , s0 >

- S insieme finito degli stati
- I insieme finito degli ingressi
- U insieme finito delle uscite
- d : S × I → S funzione di transizione
- t : S × I → U funzione di uscita
- s0 ∈ S stato iniziale

Esempio: centralino

- Un centralino (PABX) accetta chiamate interne con numeri di due cifre e chiamate esterne con numeri di tre cifre, precedute dallo zero.
- L'utente chiamante può sollevare il microtelefono, comporre il numero da chiamare e riattaccare.
- L'utente chiamato può sollevare il microtelefono e riattaccare.
- La rete esterna può comunicare al centralino se il numero richiesto è occupato o libero.
- Il centralino produce dei segnali udibili per comunicare all'utente se il telefono è collegato solo alla rete locale (segnale interno), oppure ha chiesto una linea esterna (segnale esterno), oppure il numero richiesto è libero, oppure occupato.

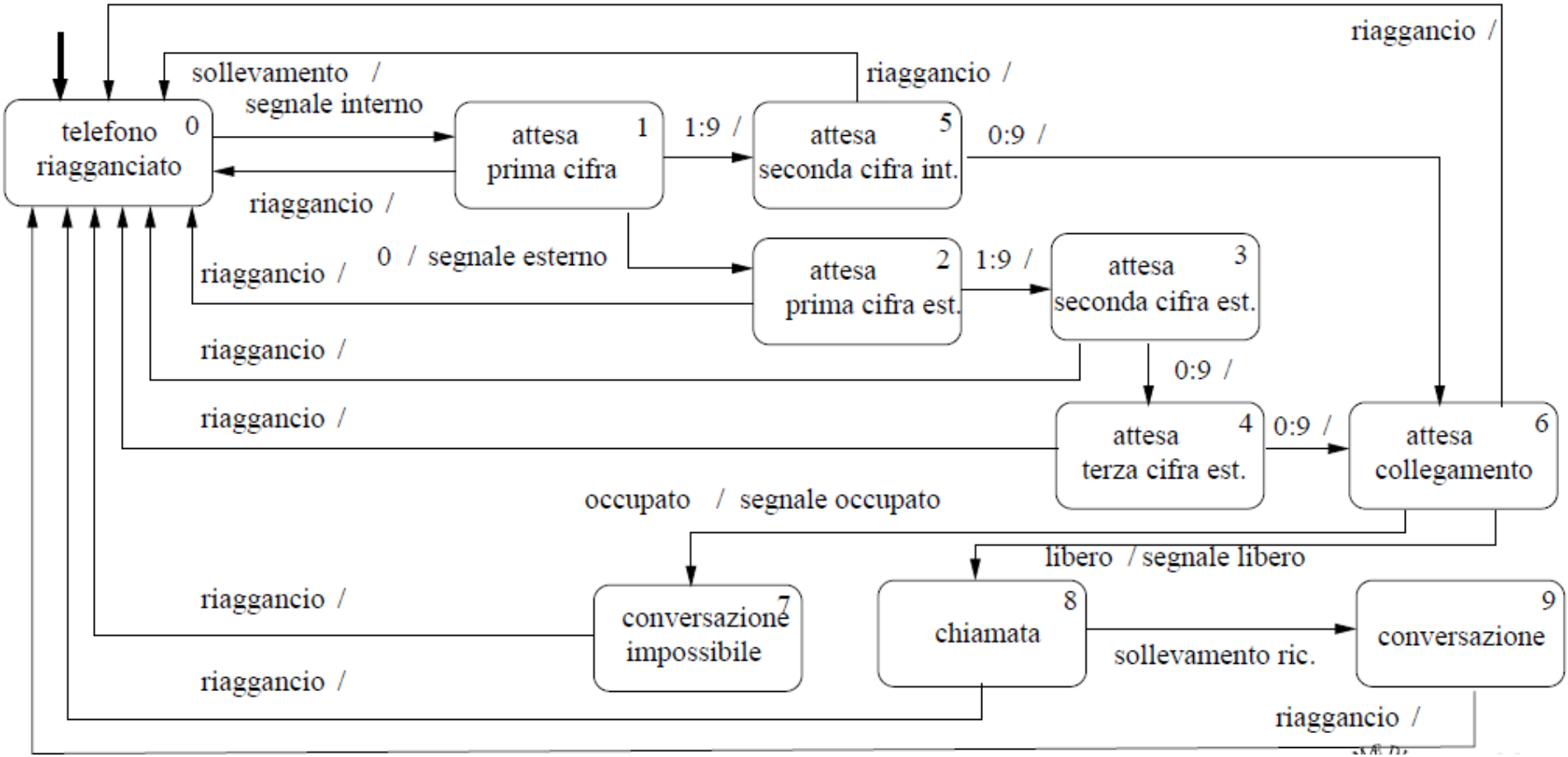
Linguaggi di specifica: automi a stati finiti

Esempio: centralino

- S = {telefono riagganciato, attesa prima cifra, attesa prima cifra esterna, attesa seconda cifra esterna, attesa terza cifra esterna, attesa seconda cifra interna, attesa collegamento, conversazione impossibile, chiamata, conversazione};
- I = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, sollevamento, sollevamento ricevente, riaggancio, occupato, libero};
- U = {segnale interno, segnale esterno, segnale occupato, segnale libero};
- d: vedi diagramma;
- t : vedi diagramma;
- s0 = riagganciato.

Non si distingue fra riaggancio del chiamante e del chiamato.

Esempio: centralino



Esempio: centralino

- Gli stati sono identificati da un nome ed un numero.
- Le transizioni hanno un'etichetta della forma ingresso / uscita.
- Un'espressione della forma m..n significa "qualsiasi cifra compresa fra m ed n incluse".
- Osservare che, per ogni stato, le transizioni uscenti sono mutuamente esclusive (determinismo).
- Osservare il numero di transizioni attivate dallo stesso ingresso (riaggancio) e incidenti sullo stesso stato di arrivo.

Esempio: centralino

- Rappresentazione tabulare

	0	1	2	3	4	5	6	7	8	9
0		S/int.								
1	R/		0/est.			1:9/				
2	R/			1:9/						
3	R/				0:9/					
4	R/						0:9/			
5	R/						0:9/			
6	R/							O/occ.	L/lib.	
7	R/									
8										Sr/
9	R/									

- Righe: stati di partenza; colonne stati di arrivo.
- R: riaggancio; S: sollevamento lato chiamante; O: occupato; L: libero; Sr: sollevamento lato chiamato; int: segnale interno; est: segnale esterno; occ: segnale occupato; lib: segnale libero

Dipendenze tra requisiti

Possono essere descritte tramite le tabelle di tracciabilità

- Delle funzionalità, dell'origine, dei sottosistemi, dell'interfaccia

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		U	R					
1.2			U			R		U
1.3	R			R				
2.1			R		U			U
2.2								U
2.3		R		U				
3.1								R
3.2							R	

U = usa, es. 1.1 usa 1.2

R = relazione

Linguaggi di specifica: automi a stati finiti

Esempio: centralino

Domande:

- cosa non dice questo modello?
- qual è il livello di astrazione di questo modello?

Linguaggi di specifica: UML

I metodi e i linguaggi O-O (ad esempio, UML) si basano sulla simulazione.

-
- Un **oggetto** rappresenta un'entità del mondo reale, concreta (ad esempio, un motore) o astratta (ad esempio, un sistema di equazioni).

- Un oggetto è definito dalla sua identità, dai valori degli **attributi** dell'entità (ad esempio, i dati di targa di un motore, la velocità e la coppia attuali. . .) e dalle **operazioni** l'entità può eseguire su richiesta di altre entità (ad esempio, modificando la velocità. . .).

- I **collegamenti** tra gli oggetti rappresentano relazioni logiche tra le entità corrispondenti. Ad esempio, un motore apre una valvola, un vettore soddisfa un insieme di equazioni.

- Una **classe** è una descrizione di modello per un insieme di oggetti con gli stessi attributi (possibilmente con valori diversi) e operazioni.

- Un'**associazione** tra due classi è un modello per i collegamenti tra i rispettivi oggetti.

Linguaggi di specifica: UML

Lo **Unified Modeling Language (UML)** è un formalismo di modellazione che comprende diversi linguaggi. È standardizzato dall'OMG (Object Management Group) (https://www.omg.org/spec/UML/2.5.1/). La versione attuale (2021) è la 2.5.1.

- Lo UML permette di costruire **modelli** di **analisi**, di **progetto** e di **implementazione**.

- Ciascun modello è formato da **sottomodelli (viste)** che descrivono diversi aspetti del sistema modellato, in particolare:
  - vista **funzionale**;
  - vista **strutturale**;
  - vista **dinamica**;
  - vista **fisica**,

- Per ciascuna vista si usano uno o piú **linguaggi grafici**.

- La vista **fondamentale** è quella **strutturale**, costruita con un linguaggio orientato agli oggetti.

Linguaggi di specifica: UML

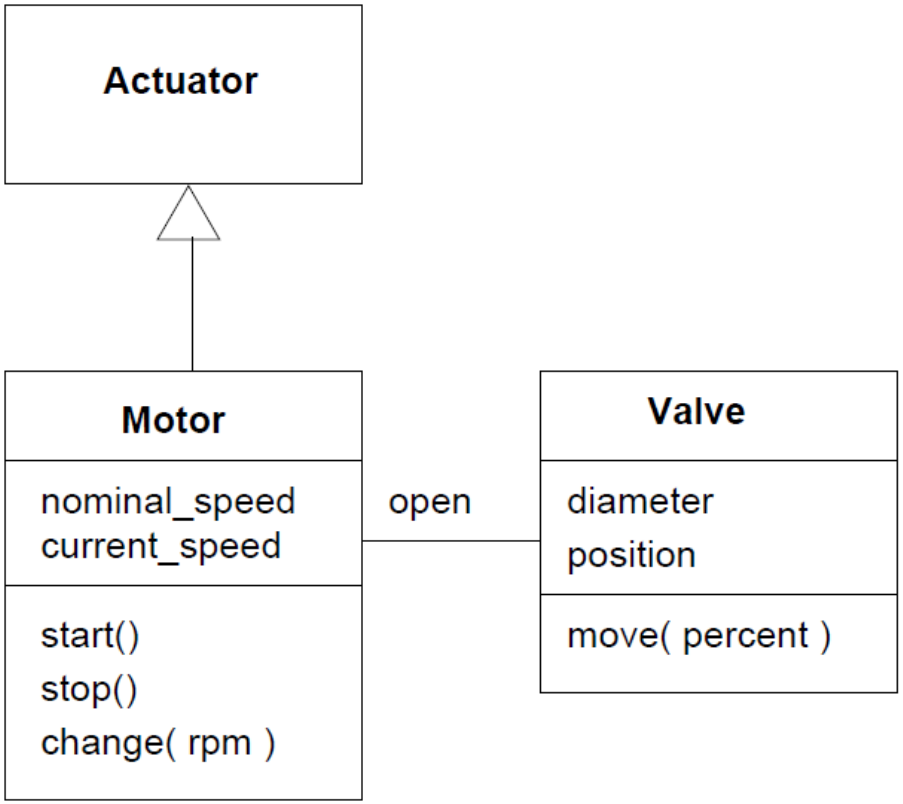
È un linguaggio **visuale**, standard, aperto ed estensibile di modellazione, cioè è un linguaggio che fornisce una sintassi per costruire modelli. Nato per modellare sistemi software object-oriented, grazie ai meccanismi di estensione messi a disposizione dallo stesso linguaggio, viene attualmente utilizzato in una varietà di domini applicativi.

UML non fornisce nessuna metodologia di modellazione: può essere usato con **qualsiasi metodologia** esistente.

Perché **unificato**?

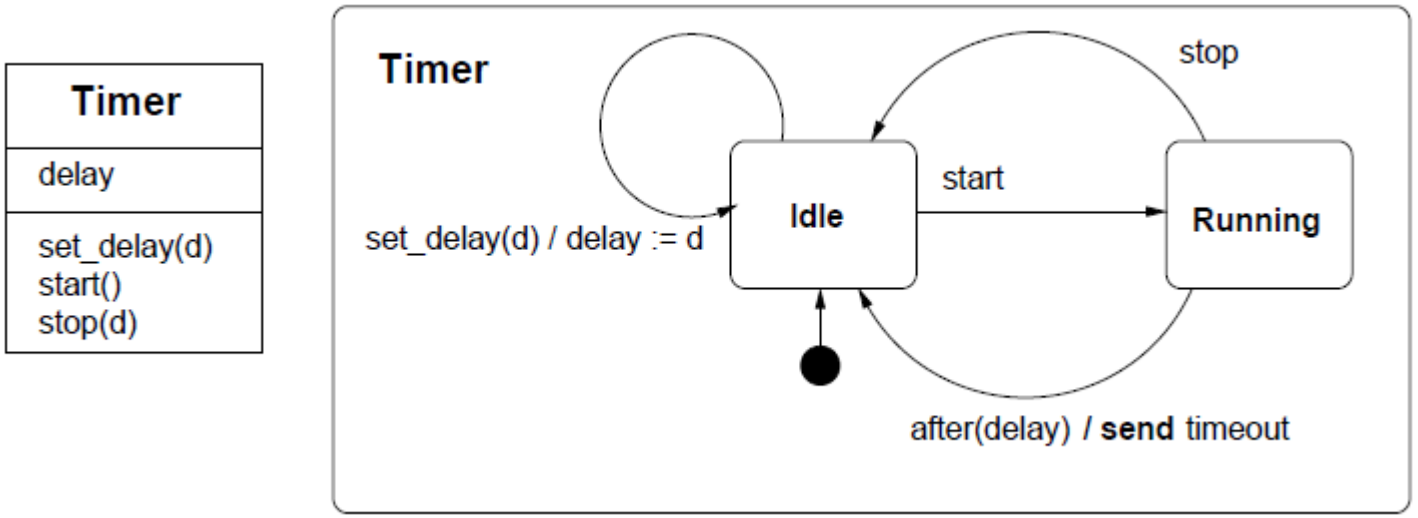
- UML fornisce una sintassi di modellazione visuale unica per l'intero ciclo di vita del software (dalle specifiche all'implementazione).
- UML è stato usato per modellare differenti domini applicativi (da sistemi embedded a sistemi di supporto alle decisioni).
- UML è indipendente dal linguaggio di programmazione e dalla piattaforma di sistema.
- UML può supportare differenti metodologie di sviluppo.
- UML tenta di essere consistente ed uniforme nell'applicazione di un piccolo insieme di concetti predefiniti.

Esempio classico di un diagramma delle classi (vista strutturale)

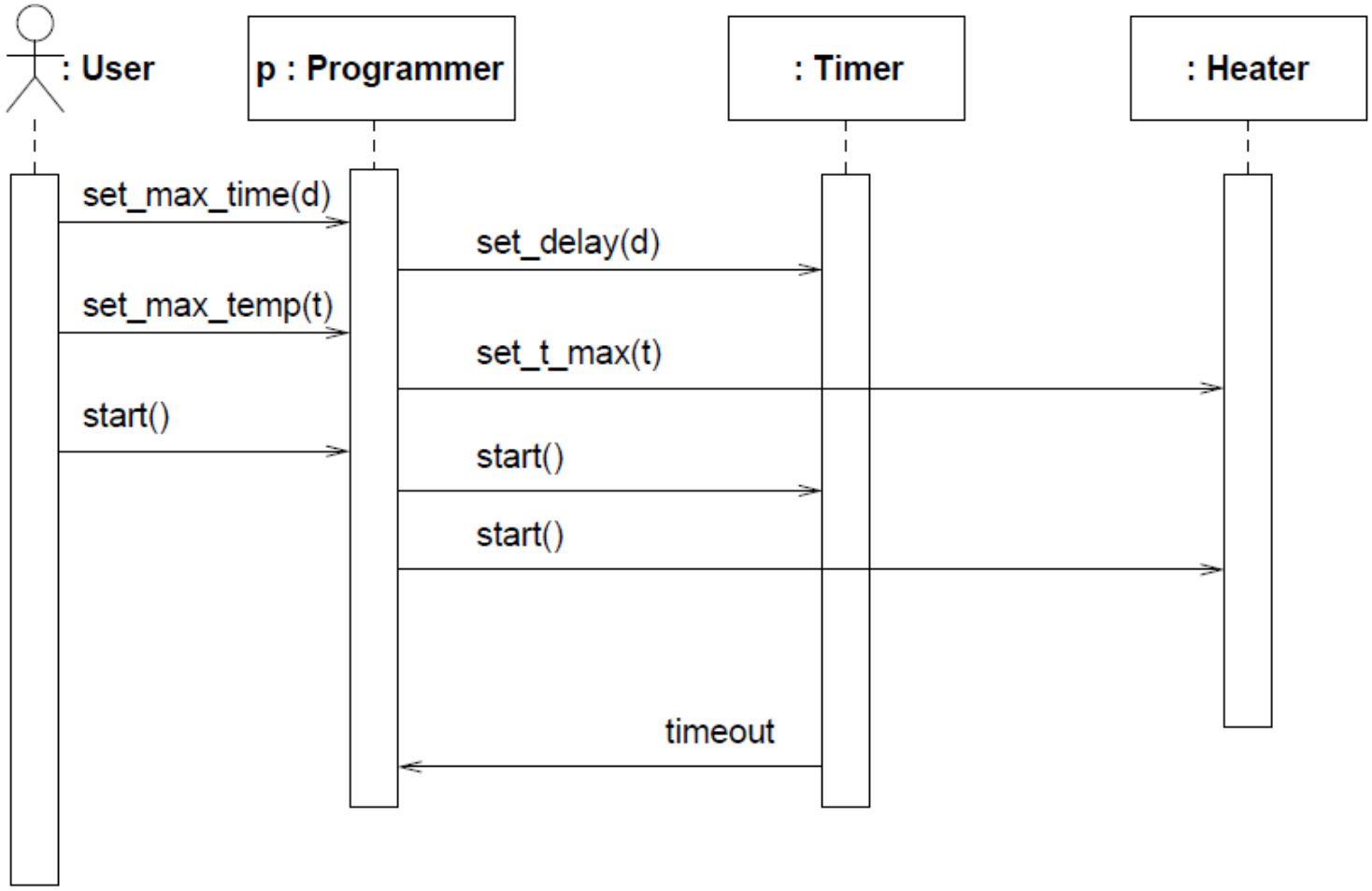


Modellazione comportamentale (vista dinamica)

- Le **macchine a stati** descrivono il modo in cui un oggetto o un (sotto)sistema risponde agli eventi.
- L'UML utilizza un complesso linguaggio di macchine a stati derivato dal formalismo di Statecharts.
- Le **interazioni** descrivono il modo in cui gli oggetti o i (sotto)sistemi interagiscono scambiandosi messaggi.
- Le **attività** descrivono il flusso di controllo e i dati coinvolti nell'esecuzione di un compito.



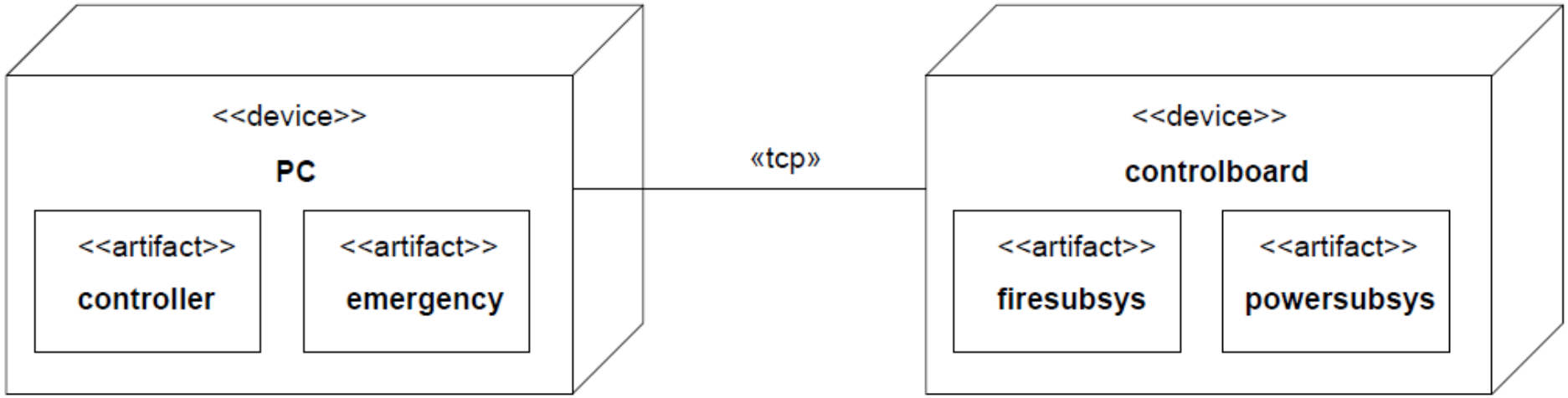
Modellazione comportamentale: diagramma di sequenza



Modellazione fisica (vista fisica): diagrammi di distribuzione

- I **diagrammi di distribuzione** descrivono la **struttura fisica** di un sistema:
  - I **nodi** rappresentano dispositivi hardware che eseguono calcoli (in genere, l'intero computer o parti di livello inferiore, se necessario).
  - Gli **artefatti** rappresentano file (contenenti programmi o dati).
  - Le **associazioni** rappresentano percorsi di comunicazione tra i nodi.

Modellazione fisica (vista fisica): diagrammi di distribuzione



Elementi, modelli e diagrammi

- Un **modello** è un insieme strutturato di **informazioni**;
- Le informazioni sono organizzate in **elementi di modello**;
- Ogni tipo di elementi di modello rappresenta un concetto del linguaggio, p.es. classe, associazione, stato . . .
- Gli elementi di modello in generale sono composti da altri elementi, p.es. l'elemento classe contiene elementi attributo ed elementi operazione;
- agli elementi di modello sono associati **elementi di presentazione** grafici e/o testuali;
- ogni elemento di modello può avere varie presentazioni, piú o meno dettagliate;
- un **diagramma** è un insieme di elementi di presentazione

Semiformalità

L'UML è un linguaggio **semiformale** in quanto permette di costruire modelli **incompleti**. Per esempio:

- La semantica delle operazioni può non essere definita;
- il tipo degli attributi può non essere definito;
- il tipo degli argomenti di un'operazione può non essere definito;
- una classe può essere definita senza specificarne attributi e operazioni.

Osserviamo che la **semantica** delle operazioni può essere definita in vari modi, mentre la loro **implementazione** non viene mai rappresentata esplicitamente in un modello UML.

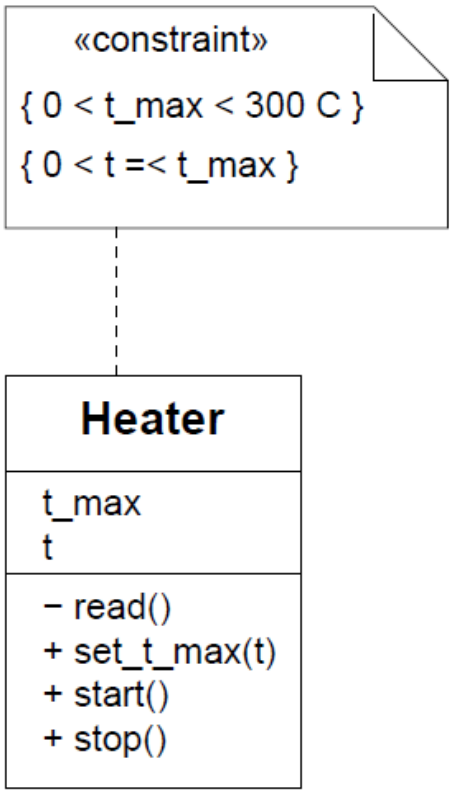
Inoltre, distinguiamo l'incompletezza semantica di un elemento di modello dall'**elisione** di elementi di rappresentazione. Dalla rappresentazione di qualsiasi elemento di modello si possono omettere, di volta in volta, i tratti non necessari in un dato contesto.

Meccanismi di estensione

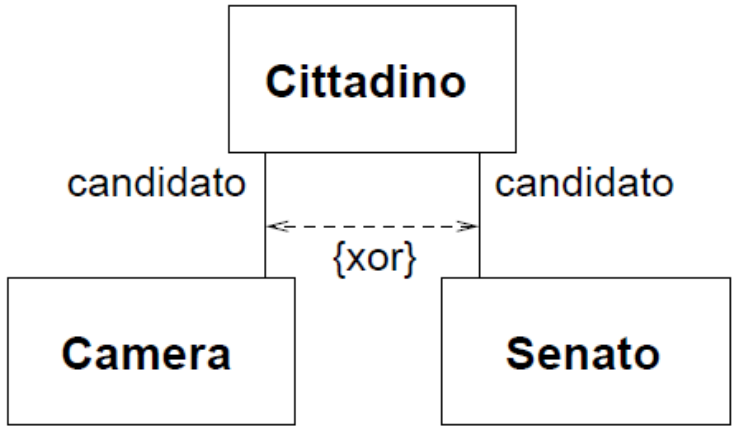
- I meccanismi di estensione permettono di adattare il linguaggio ad esigenze particolari aggiungendo informazioni a elementi di modello o introducendo nuovi elementi partendo da elementi preesistenti.
  - **vincoli**: applicabili a qualsiasi elemento, in linguaggio
    - naturale;
    - logico/matematico;
    - OCL (Object Constraint Language), FOL adattata a UML;
  - **stereotipi**: estensioni di elementi di modello;
  - **valori etichettatii**: proprietà di stereotipi, della forma {nome = valore}.

- Un **profilo** è un insieme organico di stereotipi

Meccanismi di estensione: vincoli



vincoli su attributi



vincoli su associazioni

Meccanismi di estensione: stereotipi e valori etichettati

- Il **metamodello** del linguaggio UML è il suo **vocabolario** e la sua **grammatica**.
  - Il metamodello è costituito da **metaclassi**, ognuna delle quali definisce un tipo di elementi di modello, cioè un concetto base del linguaggio. P.es., il concetto di classe è definito dalla metaclasse Class, che a sua volta dipende dalle metaclassi Attribute, Operation. ecc.

- Uno **stereotipo** è un nuovo tipo di elementi di modello, cioè una nuova metaclasse, ottenuta aggiungendo proprietà (metaattributi) e vincoli ad una metaclasse preesistente.

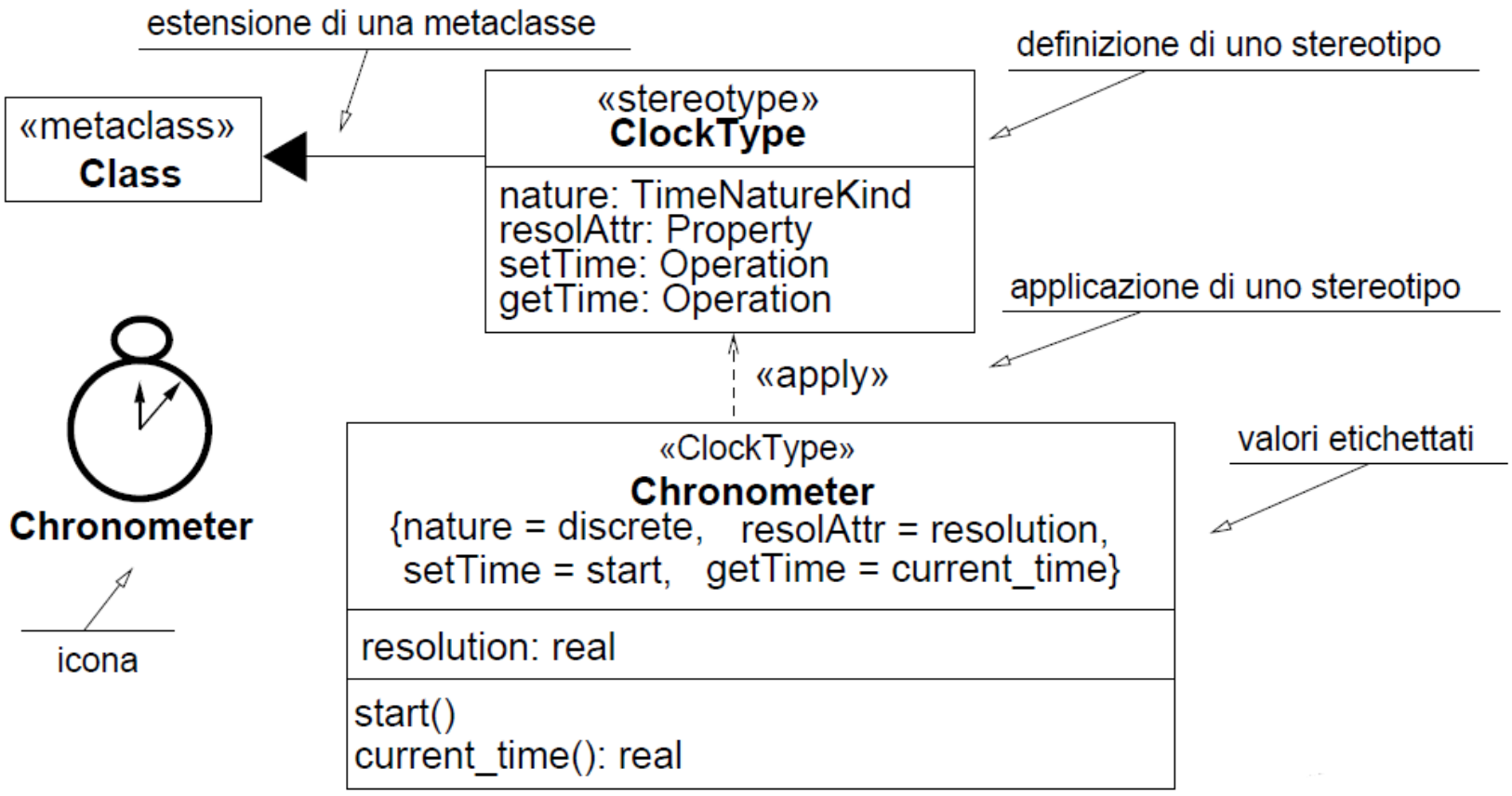
- Applicare uno stereotipo S ad un elemento E significa che E possiede le **proprietà** e soddisfa i vincoli di S. L'elemento E assegna valori specifici alle proprietà definite in S mediante valori etichettati (tagged values). Si possono applicare piú stereotipi ad uno stesso elemento.

Meccanismi di estensione: stereotipi e valori etichettati

- Nell'esempio seguente, lo **stereotipo** ClockType **estende** Class con queste proprietà:
  - (time) nature: discreto o denso;
  - resolAttr: risoluzione temporale;
  - setTime, getTime: operazioni per inizializzare e leggere la misura del tempo.

- Uno sviluppatore **applica** lo stereotipo creandone un'istanza (Chronometer) definita dai seguenti valori etichettati:
  - {nature = discrete};
  - {resolAttr = resolution};
  - {setTime = start};
  - {getTime = current_time};

Meccanismi di estensione: stereotipi e valori etichettati



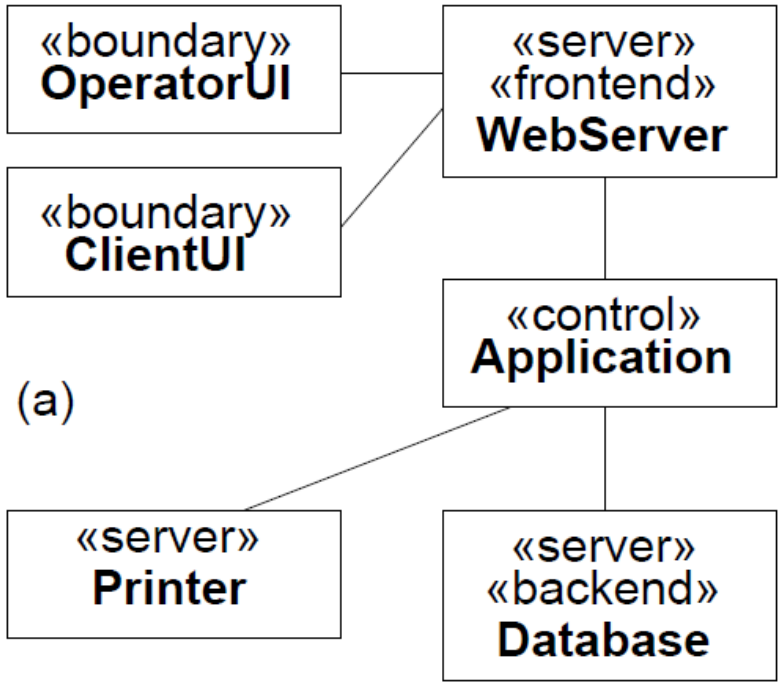
Meccanismi di estensione: stereotipi e valori etichettati

Uno stereotipo si può usare per varii scopi, fra cui:

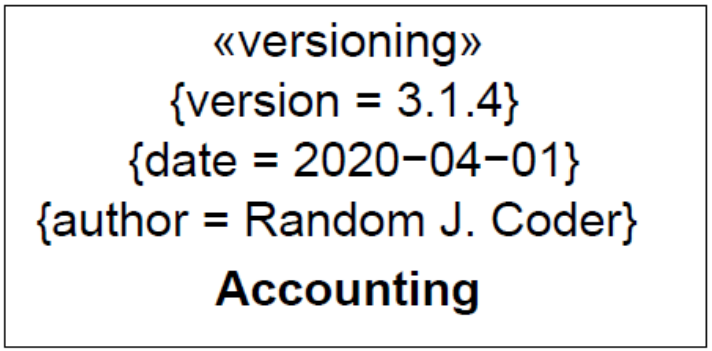
- introdurre elementi di modello specifici di un dominio di applicazione, come nell'esempio precedente;
- mettere in evidenza il ruolo nel sistema di certe istanze di elementi di modello, come nell'esempio (a) del lucido successivo;
- aggiungere ad istanze di elementi di modello informazioni non pertinenti al sistema modellato, ma utili per la gestione del progetto di sviluppo, come nell'esempio (b) successivo;
- aggiungere informazioni che possono essere interpretate da strumenti di sviluppo, p.es. per generare codice o documentazione, come nell'esempio (c) successivo.

Il nome di uno stereotipo può essere sostituito da un'icona, che può anche sostituire il normale elemento di presentazione dell'elemento di modello stereotipato.

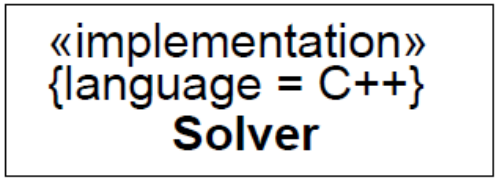
Meccanismi di estensione: stereotipi e valori etichettati



(a)



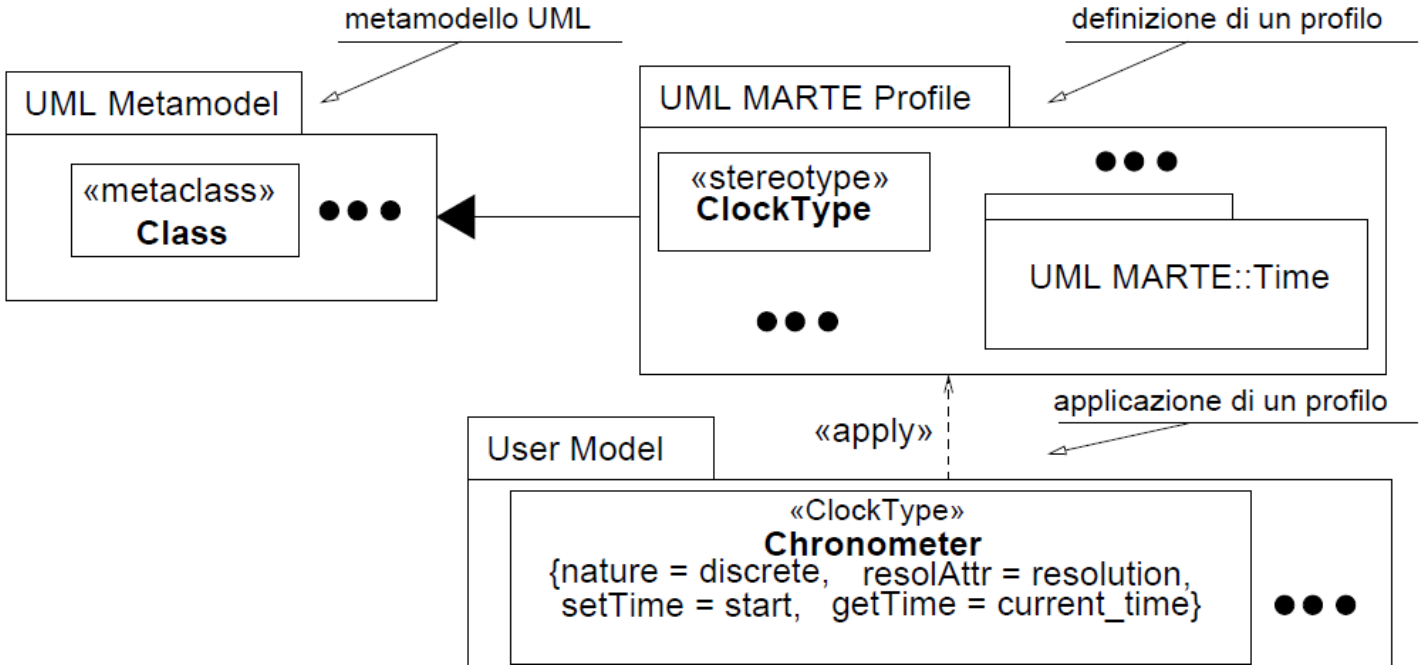
(b)



(c)

Meccanismi di estensione: profili

- Un profilo è un insieme organico di stereotipi, che definisce nuovi elementi di modello utili in particolari campi di applicazione.
- Lo OMG cura la standardizzazione di profili proposti da consorzi di industrie ed enti di ricerca.
- Per esempio, il profilo UML MARTE (Modeling and Analysis of Real-time and Embedded systems) è stato sviluppato e standardizzato per la modellazione di sistemi embedded e in tempo reale. http://www.omg.org/omgmarte/



Classi e oggetti

- Una **classe** rappresenta astrattamente un insieme di oggetti che hanno gli stessi attributi, operazioni, relazioni e comportamenti.
- In un **modello di analisi**, una classe rappresenta un concetto pertinente al sistema che viene specificato o al suo ambiente.
- In un **modello di progetto o di implementazione**, una classe rappresenta un'entità software, per esempio una classe del linguaggio C++.
- La **presentazione grafica** di una classe è un rettangolo contenente il nome della classe e, opzionalmente, l'elenco degli attributi e delle operazioni, ed eventuali altre informazioni.
- Se la classe ha uno o piú **stereotipi**, i loro nomi vengono scritti sopra al nome della classe, oppure si rappresentano con un'icona nell'angolo destro in alto.
- La **forma minima** è un rettangolo contenente solo il nome ed eventualmente lo stereotipo. Se la classe ha uno stereotipo rappresentabile da un'icona, la rappresentazione minima consiste nell'icona e nel nome.

Attributi

- **nome**: unico campo obbligatorio;
- **tipo**: un tipo predefinito dell'UML o di un linguaggio di programmazione, o una classe definita in UML dallo sviluppatore;
- **visibilità**: privata, protetta, pubblica, package; quest'ultimo livello di visibilità significa che l'attributo è visibile da tutte le classi appartenenti allo stesso package;
- **àmbito** (scope): istanza, se l'attributo appartiene al singolo oggetto, statico se appartiene alla classe, cioè è condiviso fra tutte le istanze della classe, analogamente ai campi statici del C++ o del Java;
- **molteplicità**: índica se l'attributo può essere replicato, cioè avere piú valori (si può usare per rappresentare un array);
- **valore iniziale**: valore assegnato all'attributo quando si istanzia un oggetto.

Attributi

<visibilita‘> <nome> <molteplicita‘> : <tipo> = <val-iniziale>

- Se un attributo ha **scope statico**, viene sottolineato.

- La **visibilità** si rappresenta con i seguenti simboli:
  - + pubblica
  - # protetta
  - ~ package
  - - privata

- La **molteplicità** si índica con un numero o un intervallo numerico fra parentesi quadre, come nei seguenti esempi:
  - [3] tre valori
  - [1..4] da uno a quattro valori
  - [1..*] uno o piú valori
  - [0..1] zero o un valore (attributo opzionale)

Operazioni

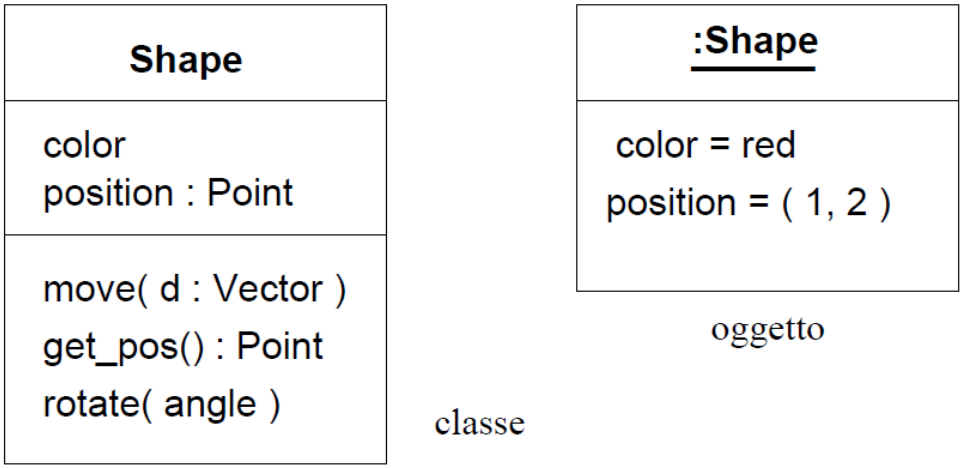
- **nome**: unico campo obbligatorio;
- **tipo** (restituito): come per gli attributi;
- **visibilità**: come per gli attributi;
- **àmbito** (scope): come per gli attributi;
- **lista dei parametri**: fra parentesi tonde, separati da virgole;

- Il nome e la lista dei parametri costituiscono la **segnatura/firma** dell'operazione.
  - <visibilita'> <nome> (<lista-parametri>) : <tipo>

- Per ciascun **parametro** si specificano
  - **nome**: unico campo obbligatorio;
  - **direzione**: ingresso (in), uscita (out), ingresso e uscita (inout);
  - **tipo**; come per gli attributi;
  - **valore default**: valore passato al metodo che implementa la funzione, se l'argomento corrispondente al parametro non viene specificato.
  - <direzione> <nome> : <tipo> = <val-default>

Oggetti

- Si scrivono i nomi dell'oggetto e della classe d'appartenenza, sottolineati e separati dal carattere ':', ed opzionalmente gli attributi con i rispettivi valori.
- Il nome dell'oggetto può mancare.
- I modelli UML sono basati principalmente sulle classi e le loro relazioni. Gli oggetti si usano per esemplificare delle situazioni particolari o tipiche.



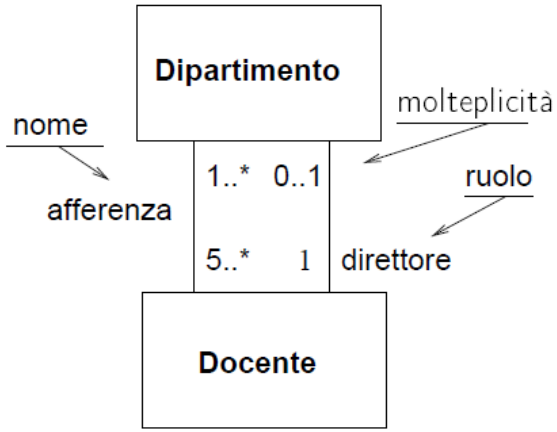
Associazioni

- In UML sono possibili associazioni di qualsiasi **arietà** (binarie, ternarie, . . .), ma tratteremo solo quelle binarie

- Un'associazione **binaria** è definita dai suoi **estremi** (end), ciascuno dei quali comprende
  - **nome** della classe identificata dall'estremo;
  - **ruolo** della classe nell'associazione;
  - **molteplicità**: numero di istanze della classe che possono partecipare all'associazione con una istanza dell'altra classe;
  - **altre proprietà**.

- Un'associazione può avere un nome.

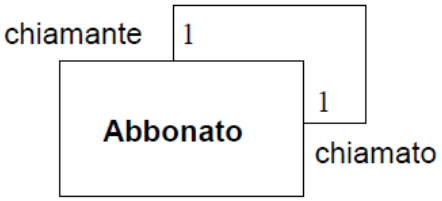
- L'associazione di nome "afferenza" dà queste informazioni:
  - un docente può afferire ad uno o piú dipartimenti (molteplicità 1 .. * all'estremo Dipartimento);
  - ad un dipartimento afferiscono almeno cinque docenti (molteplicità 5 .. * all'estremo Docente).
- L'associazione anonima fra Dipartimento e Docente dà queste informazioni:
  - un docente può partecipare all'associazione con uno o nessun dipartimento (molteplicità 0 .. 1 all'estremo Dipartimento);
  - ad un dipartimento è associato esattamente un docente col ruolo "direttore" (molteplicità 1 all'estremo Docente).
- N.B.: Il diagramma non specifica se il direttore deve afferire al dipartimento!



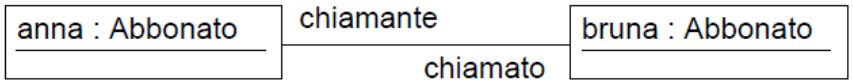
(a)

- La fig. (b) del lucido mostra un'associazione fra istanze di una stessa classe, in cui esattamente un abbonato ha il ruolo di chiamante ed esattamente un abbonato ha il ruolo di chiamato.

- La fig. (c) mostra un'istanza (link) dell'associazione



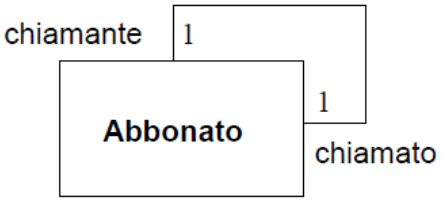
(b)



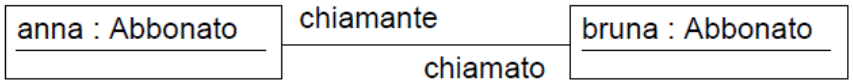
(c)

- La fig. (b) del lucido mostra un'associazione fra istanze di una stessa classe, in cui esattamente un abbonato ha il ruolo di chiamante ed esattamente un abbonato ha il ruolo di chiamato.

- La fig. (c) mostra un'istanza (link) dell'associazione



(b)



(c)

Associazioni e relazioni speciali

- **Aggregazione**: l'associazione (debole) di un'entità composta con i suoi componenti, quando i componenti possono esistere al di fuori dell'entità composta (ad esempio una squadra e i suoi giocatori, una biblioteca e i suoi libri).
- **Composizione**: L'associazione (forte) di un'entità composta con i suoi componenti, quando i componenti potrebbero non esistere al di fuori dell'entità composta (ad esempio un motore e le sue parti).
- **Generalizzazione**: una relazione (non un'associazione) che specifica che una classe è un sottoinsieme di un'altra (ad esempio, i motori sono una sottoclasse di attuatori).

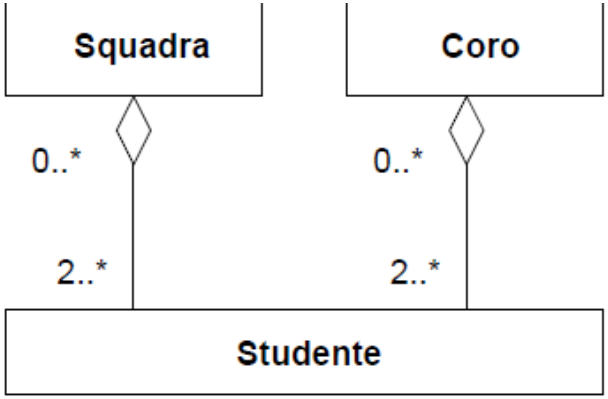
Aggregazione

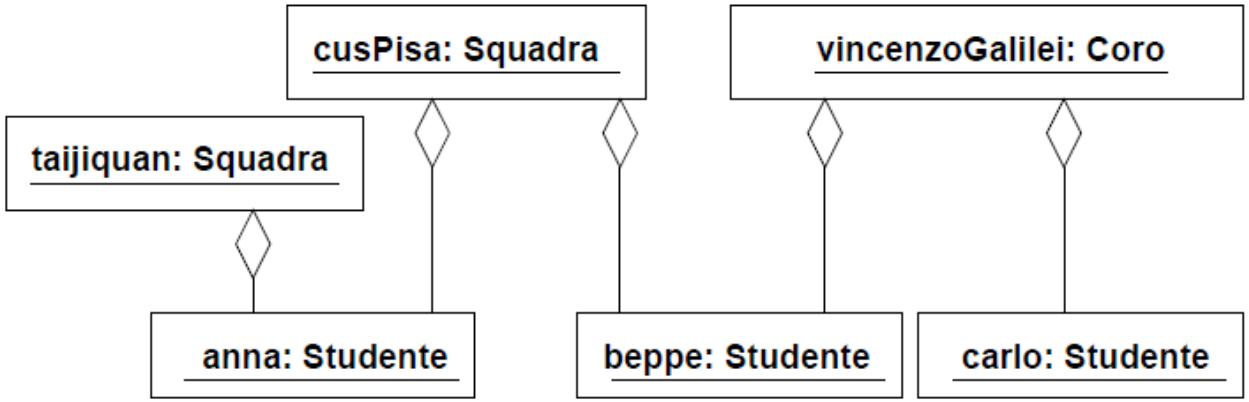
L'aggregazione è un'annotazione (corrispondente ad una proprietà) aggiunta ad un'associazione per esprimere il concetto di appartenenza ad un gruppo o struttura. Per esempio, una squadra è un'aggregazione di giocatori, un catalogo è un'aggregazione di titoli di libri.

Questo concetto non ha una definizione precisa, quindi l'uso di questa annotazione è facoltativo. La figura seguente può dare un'idea della differenza fra un'associazione annotata come aggregazione (con una losanga) ed una non annotata.

Aggregazione

Un’istanza di una classe può partecipare a piú di una aggregazione:

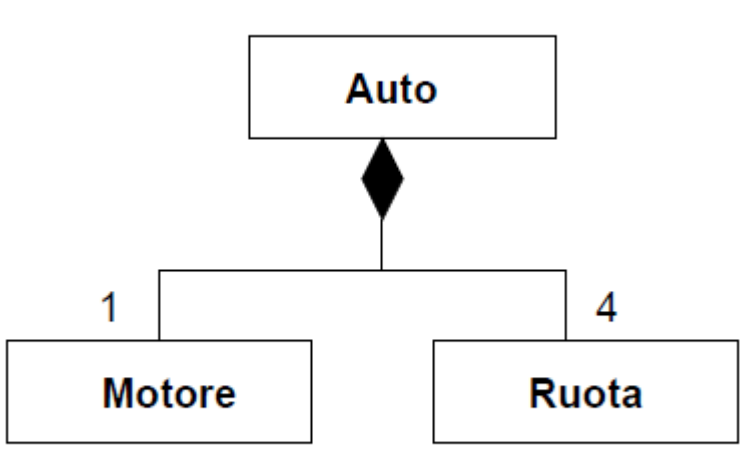


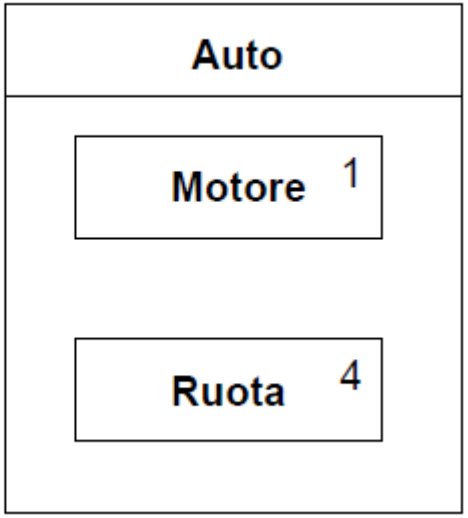


Composizione

La composizione modella una subordinazione strutturale rigida, con una dipendenza stretta fra composto e componenti.

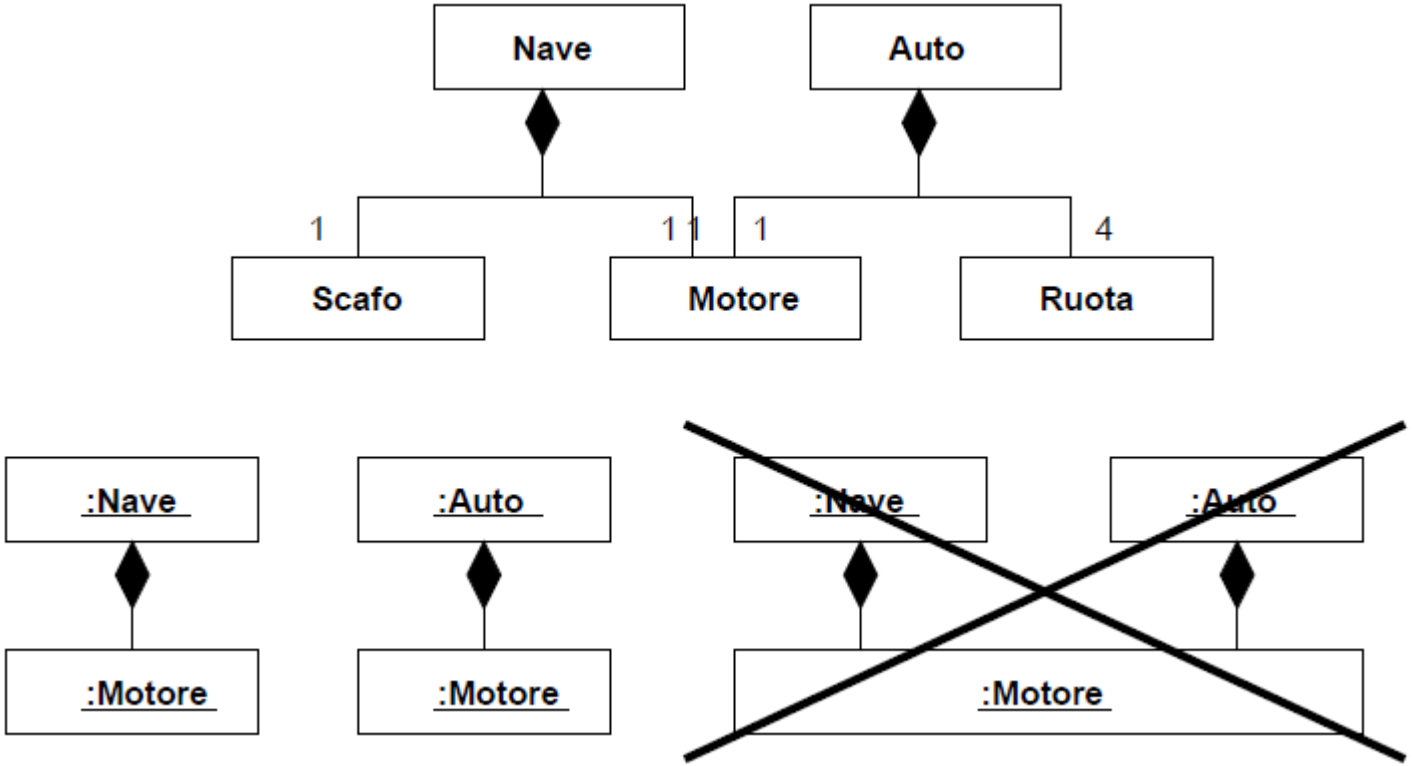
In certi casi, la relazione di composizione può implicare che i componenti non possano esistere al di fuori del composto.





Composizione

Un’istanza di una classe può partecipare ad una sola composizione



Generalizzazione

La generalizzazione è una relazione fra classi (non una associazione).

Se A generalizza B, ovvero B specializza A, si dice che A è base di B, ovvero che B deriva da A.

La generalizzazione fra classi corrisponde all'inclusione fra insiemi: la frase "la classe A generalizza la classe B" significa che ogni istanza di B è istanza di A.

Da questo deriva il principio di sostituzione (B. Liskov):
"un'istanza della classe derivata può sostituire un'istanza della classe base".

Generalizzazione

Una **classe derivata** può essere ulteriormente **specializzata** in una o piú classi, e una **classe base** può essere ulteriormente **generalizzata**.

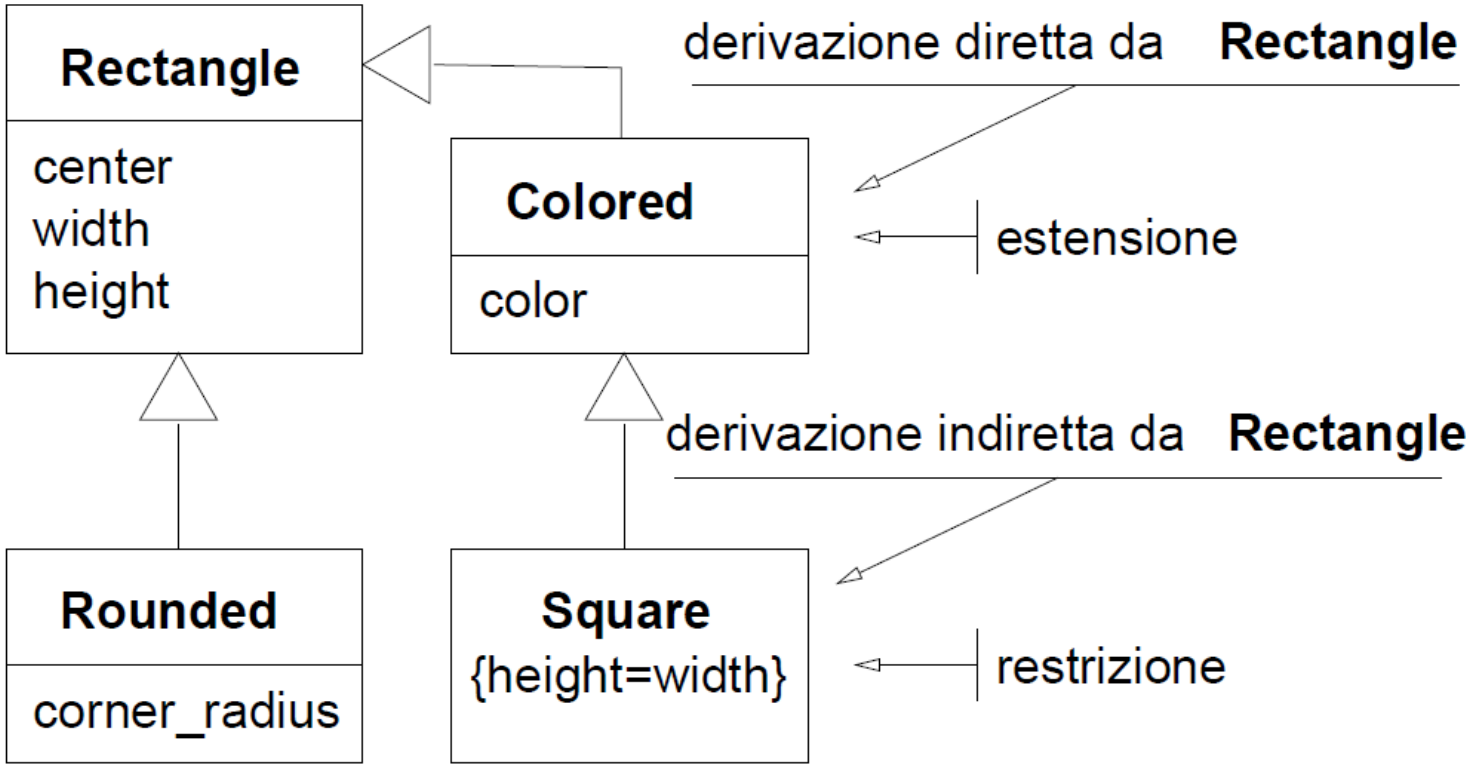
In questo caso si distinguono le classi basi o derivate **dirette** da quelle **indirette**.

Una classe derivata generalmente è un'estensione della classe base, poichè aggiunge proprietà alla stessa.

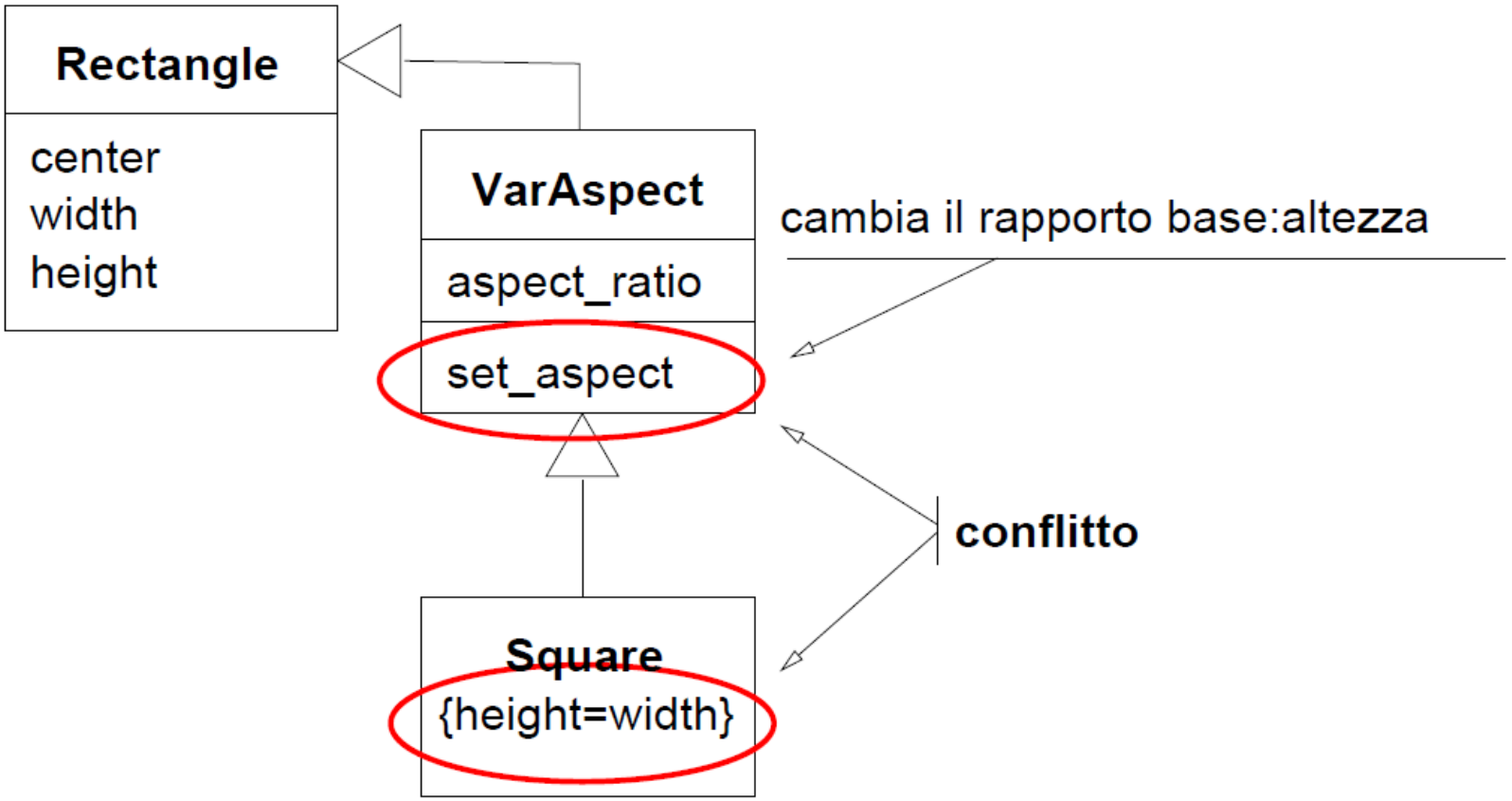
N.B.: l'insieme delle istanze di una classe derivata è sempre un sottoinsieme dell'insieme delle istanze della classe base.

Una classe derivata può anche essere una **restrizione** della classe base, quando aggiunge dei **vincoli**.
In questo caso, bisogna evitare che i vincoli víolino il principio di sostituzione.

Generalizzazione: derivazione

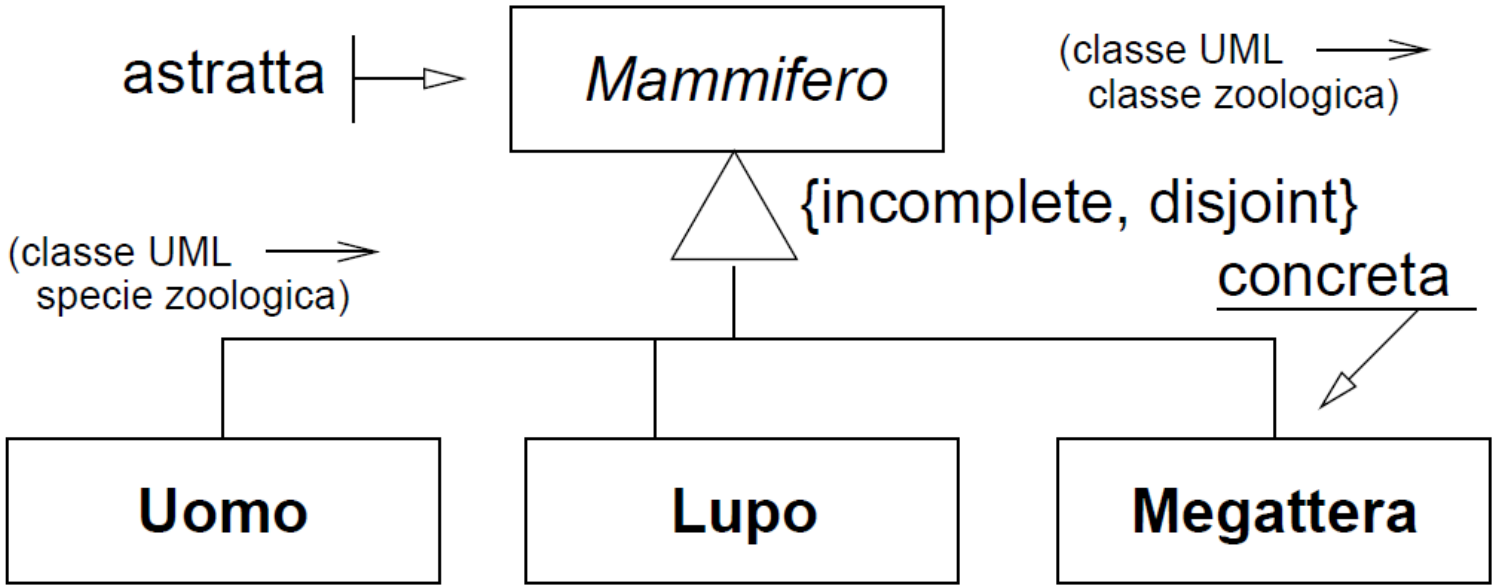


Generalizzazione: principio di sostituzione

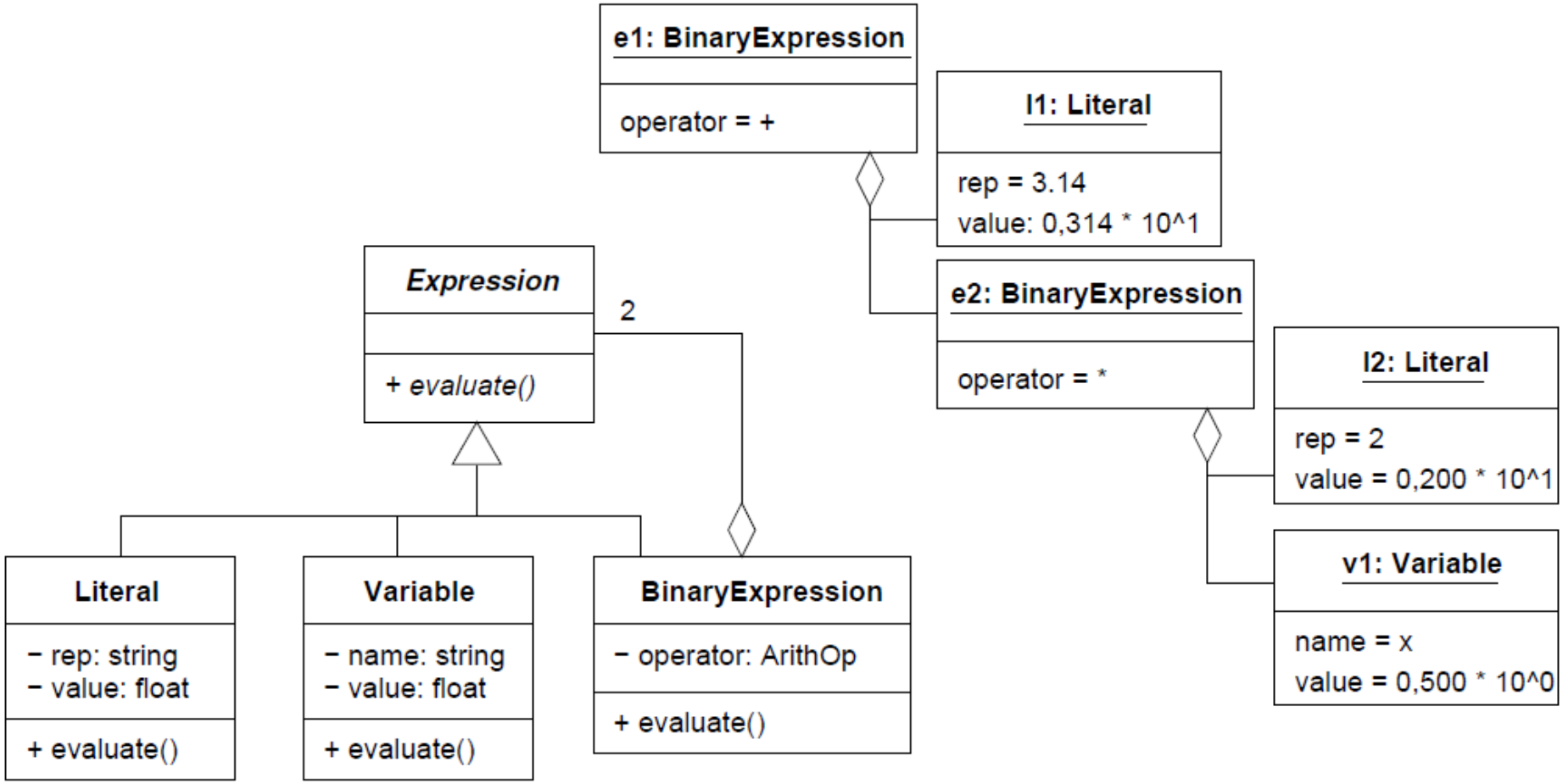


Generalizzazione: classi astratte e concrete

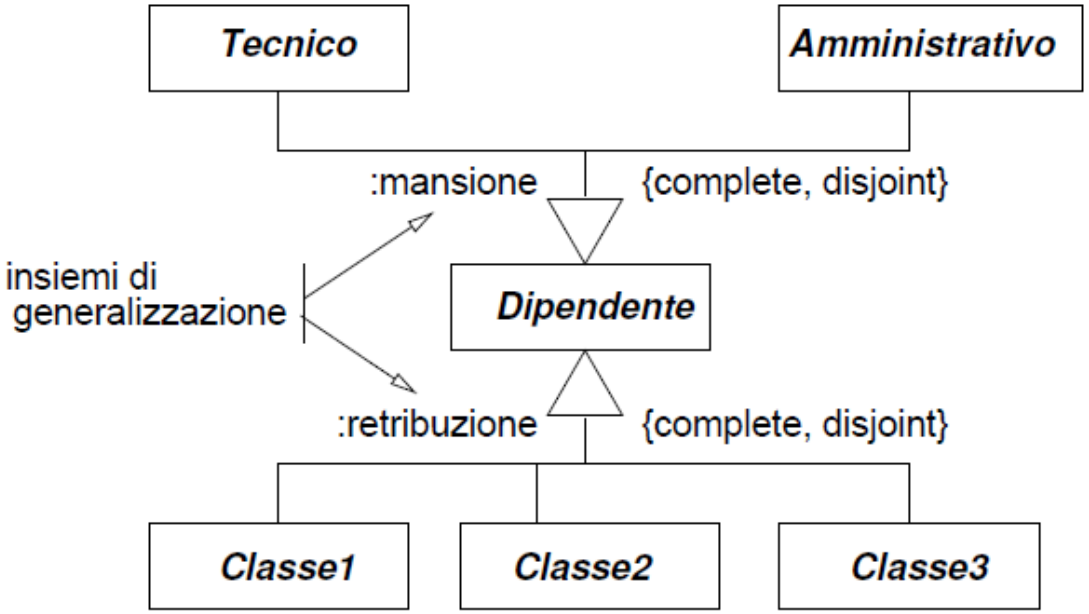
Una classe che ha istanze dirette si dice **concreta**.
Una classe che non ha istanze dirette si dice **astratta**.



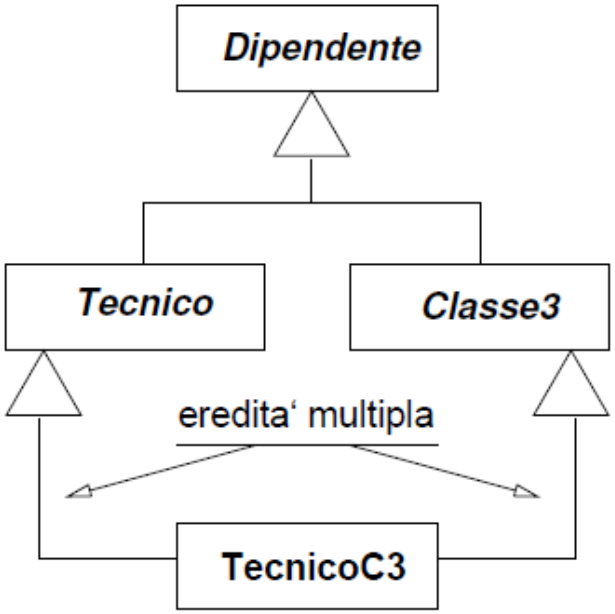
Generalizzazione: aggregazione ricorsiva



Generalizzazione: insiemi di generalizzazioni

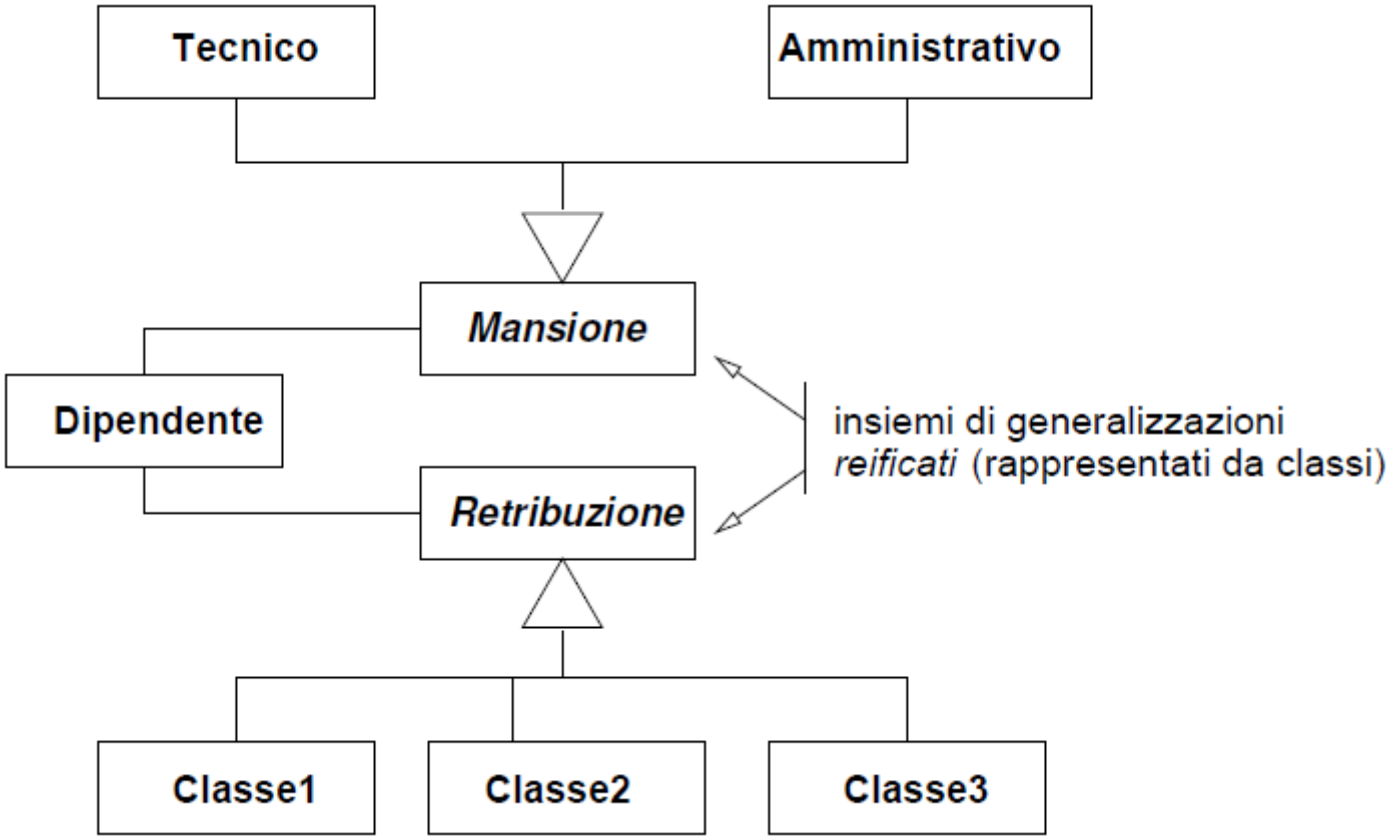


(a)

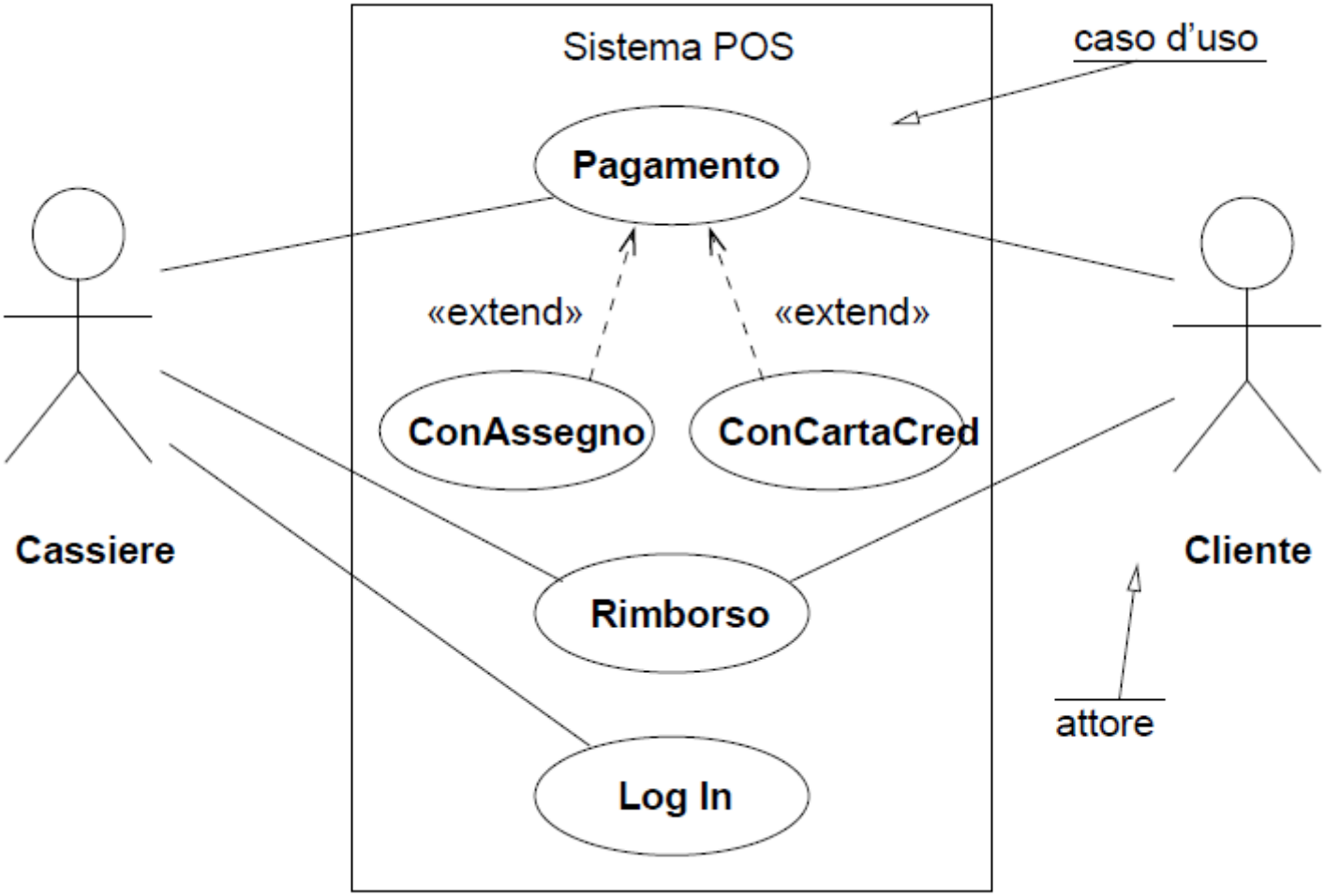


(b)

Generalizzazione: insiemi di generalizzazioni



UML – Casi d'uso



La modellazione dei casi d’uso è anch’essa una forma dell’ingegneria dei requisiti e procede tipicamente nel modo seguente:

- Identifica un confine candidato del sistema;
- Trova gli attori;
- Trova i casi d’uso;
- specifica i casi d’uso;
- identifica i flussi alternativi;
- itera i punti precedenti finché i casi d’uso, gli attori e i confini del sistema non sono stabili.

Il risultato di queste attività è il **modello dei casi d’uso**.

A partire dal modello del business o del dominio applicativo, da un modello dei requisiti e da una lista di caratteristiche, l’analista del sistema produce il modello dei casi d’uso ed un glossario di progetto. Esaminiamo i singoli passi in dettaglio.

Identifica il confine del sistema

Identifica che cosa è parte del sistema e che cosa non lo è. L’individuazione corretta dei confini del sistema consentono di determinare correttamente le specifiche funzionali. In UML 2 i confini del sistema sono chiamati soggetto. Il soggetto è definito da chi o che cosa usa il sistema (gli attori) e da quali benefici il sistema offre a questi attori (i casi d’uso).

Il soggetto è rappresentato da un rettangolo, con gli attori all’esterno e i casi d’uso all’interno. Inizialmente la modellazione dei casi d’uso verrà intrapresa avendo solo una vaga idea del soggetto: questa vaga idea diventerà sempre più chiara man mano che vengono definiti attori e casi d’uso.

Trova gli attori

L'attore è un ruolo che qualche entità esterna interpreta in qualche contesto quando interagisce direttamente con il sistema. Tipicamente è un utente, ma può essere un altro sistema, un pezzo di hardware, il tempo (es. salvataggi automatici) o qualsiasi cosa che venga a contatto con il sistema.

In UML 2, gli attori possono essere anche altri soggetti e questo permette di collegare tra loro modelli di casi d'uso.

Un **ruolo** è come un cappello che viene indossato in un particolare contesto. Un'entità può avere ruoli differenti e quindi essere attori differenti. Per esempio, un sistema potrebbe avere Customer e SystemAdministrator come attori. Jim potrebbe sia amministrare il sistema che usare il sistema. Quindi, Jim può interpretare sia il ruolo di Customer che di SystemAdministrator.

Gli **attori** sono sempre esterni al sistema. Il sistema comunque potrebbe avere una rappresentazione interna degli attori. L'attore Customer è esterno al sistema, ma il sistema potrebbe avere una classe CustomerDetails, che è una rappresentazione interna dell'attore.

Trova gli attori

Per identificare gli attori, devi considerare chi o che cosa usa il sistema, e quali ruoli interpretano nelle loro interazioni con il sistema. Le domande seguenti possono essere un valido aiuto:

- Chi o cosa usa il sistema?
- Quali ruoli interpretano nell'interazione con il sistema?
- Chi installa il sistema?
- Chi o cosa avvia e termina il sistema?
- Chi mantiene il sistema?
- Quali altri sistemi interagiscono con il sistema?
- Chi o cosa riceve/fornisce informazioni da/al sistema?
- Avviene qualcosa a tempi prefissati?

Trova gli attori

In termini di modellazione, si deve tener presente che:

- gli attori sono sempre esterni;
- gli attori interagiscono direttamente con il sistema;
- un attore non è una specifica persona o cosa, ma piuttosto un ruolo che questa persona o cosa interpreta in relazione al sistema;
- una persona o una cosa può interpretare molti ruoli;
- ogni attore deve essere individuato da un nome corto, significativo per il dominio applicativo;
- ad ogni attore deve essere associata una descrizione per illustrare cosa l'attore è nel dominio applicativo;
- il modello dell'attore può mostrare attributi ed eventi che l'attore può ricevere (quasi mai usati).

Trova i casi di uso

Un caso d'uso è "una specifica di sequenze di azioni, incluse sequenze alternative e di errore, che un sistema, sottosistema o classe possono eseguire interagendo con attori esterni".

Un caso d'uso è sempre attivato da un attore ed è sempre scritto dal punto di vista dell'attore. I casi d'uso sono rappresentati come ovali, con all'interno il nome del caso d'uso.

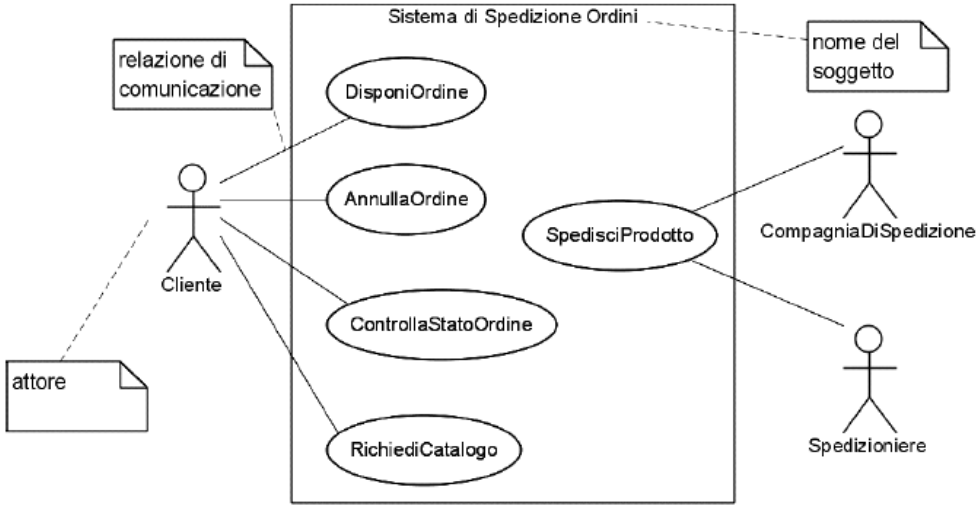
- Il miglior modo per identificare i casi d'uso è il seguente:
- esamina la lista degli attori;
- considera come ogni attore usa il sistema;
- determina una lista di casi d'uso candidati;
- individua ogni caso d'uso attraverso una frase con un verbo (DisponiOrdine, CancellaOrdine, etc.);
- Se necessario, introduci nuovi attori.

L'individuazione dei casi d'uso è un'attività iterativa e procede attraverso un raffinamento per passi successivi. Si inizia con identificare il caso d'uso con un nome, quindi con una breve descrizione ed infine con una specifica completa.

Trova i casi di uso

Di seguito, viene presentata una lista di domande che possono risultare utili nell'identificare i casi d'uso.

- Quali funzioni un attore desidera che il sistema fornisca?
- Se il sistema memorizza e recupera informazioni, chi avvia tale comportamento?
- Vi sono attori a cui viene notificato il cambio di stato del sistema (avviamento, terminazione)?
- Qualche evento esterno influenza il sistema? Cosa informa il sistema di tali eventi?
- Il sistema interagisce con qualche sistema esterno?
- Il sistema genera qualche rapporto?



UML - Il glossario di progetto

Il glossario di progetto è uno degli artefatti più importanti, in quanto fornisce un dizionario di termini-chiave e di definizioni usati nel dominio, comprensibili a chiunque sia coinvolto nel progetto. In pratica, il glossario cerca di catturare il linguaggio usato nello specifico domino.

Oltre a definire i termini principali usati nel dominio, il glossario deve anche risolvere il problema dei sinonimi e degli omonimi. I sinonimi sono parole diverse con un'unica semantica, es. alloggio ed abitazione. Nel modello, la semantica deve essere individuata da una sola parola. Gli omonimi sono parole che hanno significati diversi e possono creare problemi di comunicazione. Nel modello, si dovrebbe assegnare ad ogni nome un unico significato.

Nel glossario, dovrebbero essere memorizzati i termini preferiti e inserita una lista di sinonimi sotto la loro definizione.

I termini usati nel modello UML devono essere consistenti con i termini definiti nel glossario. Tipicamente, i glossari vengono scritti in forma testuale su un file. Per progetti complessi, possono essere organizzati in semplici basi di dati. Di seguito viene dato un esempio di glossario

Dettagliare i casi d'uso

Il nome (unico nel modello) dovrebbe essere un verbo o una frase con un verbo e deve dare una chiara idea della funzione del business o del processo. Tutte le parole dovrebbero avere la prima lettera maiuscola (UpperCamelCase).	**Use case**: *PagaTassaVendita*
Tipicamente un numero. In casi d'uso con flussi alternativi, è utile utilizzare un sistema di numerazione gerarchico: 1.1, 1.2,...	**ID**: 1
Paragrafo che cattura l'obiettivo del caso d'uso	**Brief description**: *Paga la tassa di vendita all'autorità fiscale alla fine del trimestre d'esercizio*
Attori che avviano il caso d'uso	**Primary actors**: *tempo*
Attori che interagiscono con il caso dopo l'avvio	**Secondary actors**: *autorità fiscale*
Vincoli sullo stato del sistema prima dell'avvio del caso d'uso (condizioni booleane)	**Preconditions**: *1. È la fine del trimestre d'esercizio*
Passi elencati come flusso di eventi	**Main flow**: *1. Il caso d'uso inizia quando è la fine del trimestre d'esercizio* *2. Il sistema determina l'ammontare della tassa di vendita dovuta all'autorità fiscale* *3. Il sistema esegue un pagamento elettronico all'autorità fiscale*
Vincoli sullo stato del sistema dopo l'esecuzione del caso d'uso (condizioni booleane)	**Postconditions**: *1. L'autorità fiscale riceve il corretto importo della tassa di vendita*
Eventuali alternative	**Alternative flows**:

Confrontare requisiti e casi d'uso

Una volta terminati il modello dei requisiti ed il modello dei casi d'uso, i due modelli vanno confrontati per capire se ogni requisito nel modello dei requisiti è coperto dai casi d'uso e viceversa. Eseguire questo confronto non è banale perché la relazione tra requisiti funzionali e casi d'uso è del tipo molti-a-molti. Il processo di confrontare i requisiti con i casi d'uso potrebbe essere fatto durante la generazione dei casi d'uso: usando i valori etichettati, è possibile associare una lista di requisiti, ognuno associato al proprio identificatore, ad ogni caso d'uso.

Un utile strumento per verificare la consistenza tra requisiti e casi d'uso è la **matrice di tracciabilità**, presentata di seguito.

Requirement \ Use case	UC_1	UC_2	UC_3	UC_4
R_1	X			
R_2		X	X	
R_3			X	
R_4				X
R_5	X			

La X indica che il requisito indicato nella riga ed il caso d'uso indicato nella colonna si riferiscono alla stessa specifica funzionale.
Se un requisito non corrisponde a nessun caso d'uso, allora un caso d'uso è mancante.
Viceversa, se nessun requisito corrisponde ad un caso d'uso, allora il modello dei requisiti è incompleto

Modellazione comportamentale

- Le **macchine a stati** descrivono il modo in cui un oggetto o un (sotto)sistema risponde agli eventi.
  L'UML utilizza un complesso linguaggio di macchine a stati derivato dal formalismo di **Statecharts**.

- Le **interazioni** descrivono il modo in cui gli oggetti o i (sotto)sistemi interagiscono scambiando messaggi

- Le **attività** descrivono il **flusso di controllo e di dati** coinvolti nello svolgimento di un **compito**.

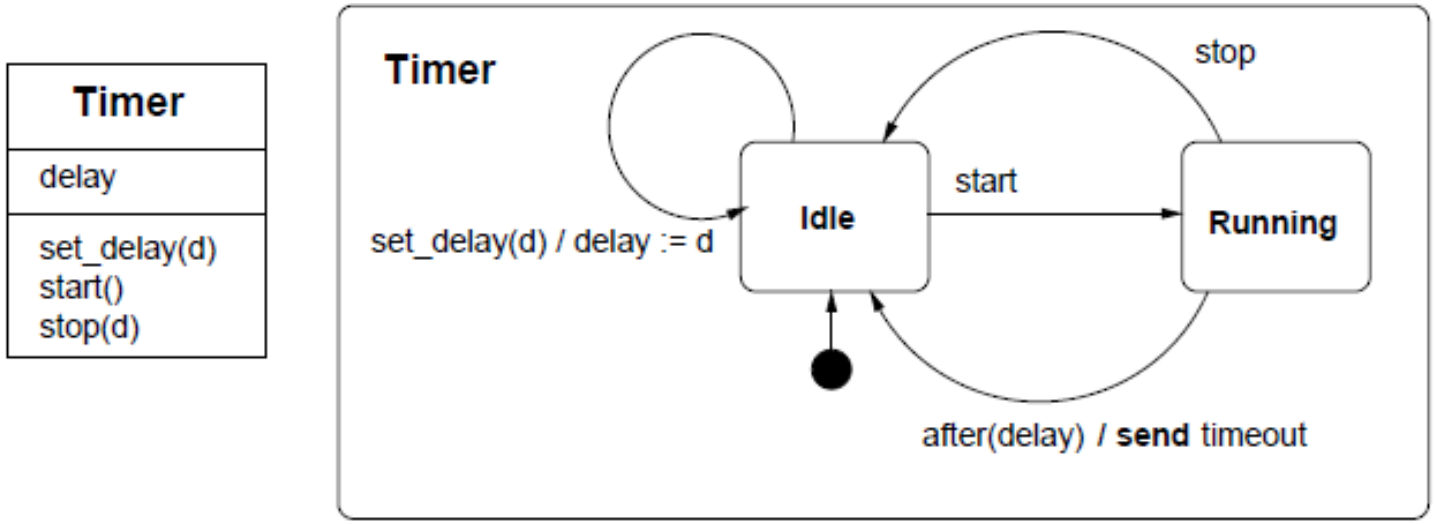
Modellazione comportamentale

- Estensione degli automi a stati finiti;
- basati sugli **Statecharts** (David Harel);
- associati a una **classe**, o ad una **collaborazione** (interazione di un insieme di oggetti), o a un **metodo**.

Le **transizioni** possono essere annotate con la seguente sintassi:

<evento> <condizione> / <azioni>

dove l'evento e le azioni (v. oltre) possono avere dei **parametri**, e la condizione è un' **espressione logica** fra parentesi quadre. Tutti questi elementi sono facoltativi.



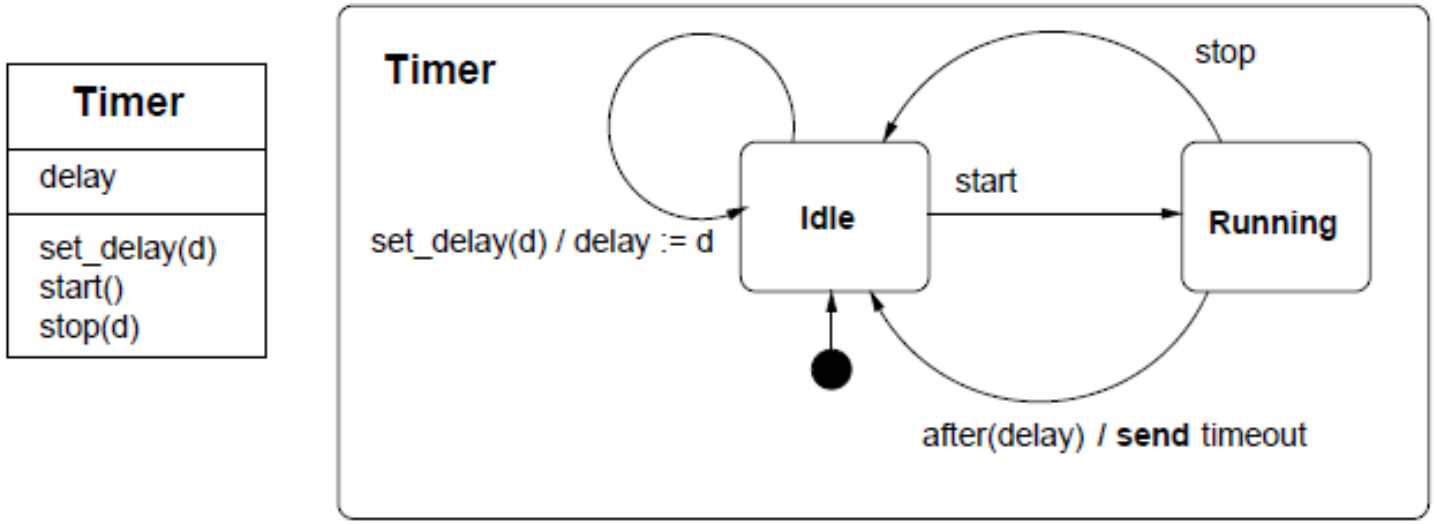
Modellazione comportamentale

- Estensione degli automi a stati finiti;
- basati sugli **Statecharts** (David Harel);
- associati a una **classe**, o ad una **collaborazione** (interazione di un insieme di oggetti), o a un **metodo**.

Le **transizioni** possono essere annotate con la seguente sintassi:

<evento> <condizione> / <azioni>

dove l'evento e le azioni (v. oltre) possono avere dei **parametri**, e la condizione è un' **espressione logica** fra parentesi quadre. Tutti questi elementi sono facoltativi.



UML – diagrammi di stato

- **occorrenze**: associate a istanti nel tempo e (spesso implicitamente) a punti nello spazio. P.es., la pressione di un tasto. Quando lo stesso tasto viene premuto di nuovo, è un'altra occorrenza.

- **eventi**: insiemi di occorrenze di uno stesso tipo. Per esempio, l'evento TastoPremuto è l'insieme di tutte le possibili occorrenze della pressione di un tasto. Spesso diremo "evento" invece di "occorrenza".

- **stati**: situazioni in cui un oggetto soddisfa certe condizioni, esegue delle attività, o semplicemente aspetta degli eventi.

- **transizioni**: passaggio da uno stato ad un altro, conseguente al verificarsi di un evento, che si può modellare come se fosse istantaneo.

- **azioni**: associate alle transizioni, non sono interrompibili, per cui si possono modellare come se fossero istantanee.

- **attività**: associate agli stati, possono essere interrotte dal verificarsi di eventi che causano l'uscita dallo stato, ed hanno una durata non nulla.

Eventi

- Ricezione di chiamate di operazione;
- ricezione di segnali;
- cambiamenti di condizioni logiche, p.es., when (p > P);
- eventi temporali
  - assoluti: at(time = 2020-12-25:00:00:00);
  - relativi all'istante di attivazione dello stato: after(20 s);
- eventi di completamento dell'attività associata allo stato.

Eventi

- Ricezione di chiamate di operazione;
- ricezione di segnali;
- cambiamenti di condizioni logiche, p.es., when (p > P);
- eventi temporali
  - assoluti: at(time = 2020-12-25:00:00:00);
  - relativi all'istante di attivazione dello stato: after(20 s);
- eventi di completamento dell'attività associata allo stato.

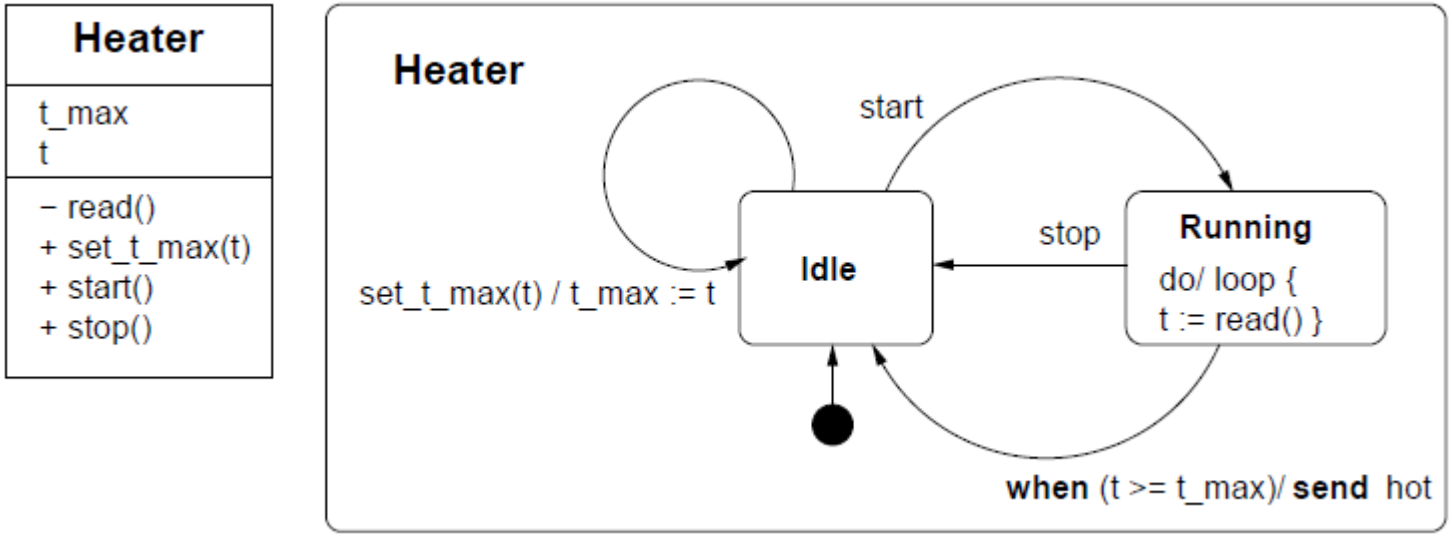
Segnali

I segnali sono delle entità che gli oggetti si possono scambiare per comunicare fra di loro, e possono essere strutturati in una gerarchia di generalizzazione:
per esempio, un ipotetico segnale Input può essere descritto come generalizzazione dei segnali Mouse e Keyboard, che a loro volta possono essere ulteriormente strutturati.

Una gerarchia di segnali si rappresenta graficamente in modo simile ad una gerarchia di classi.

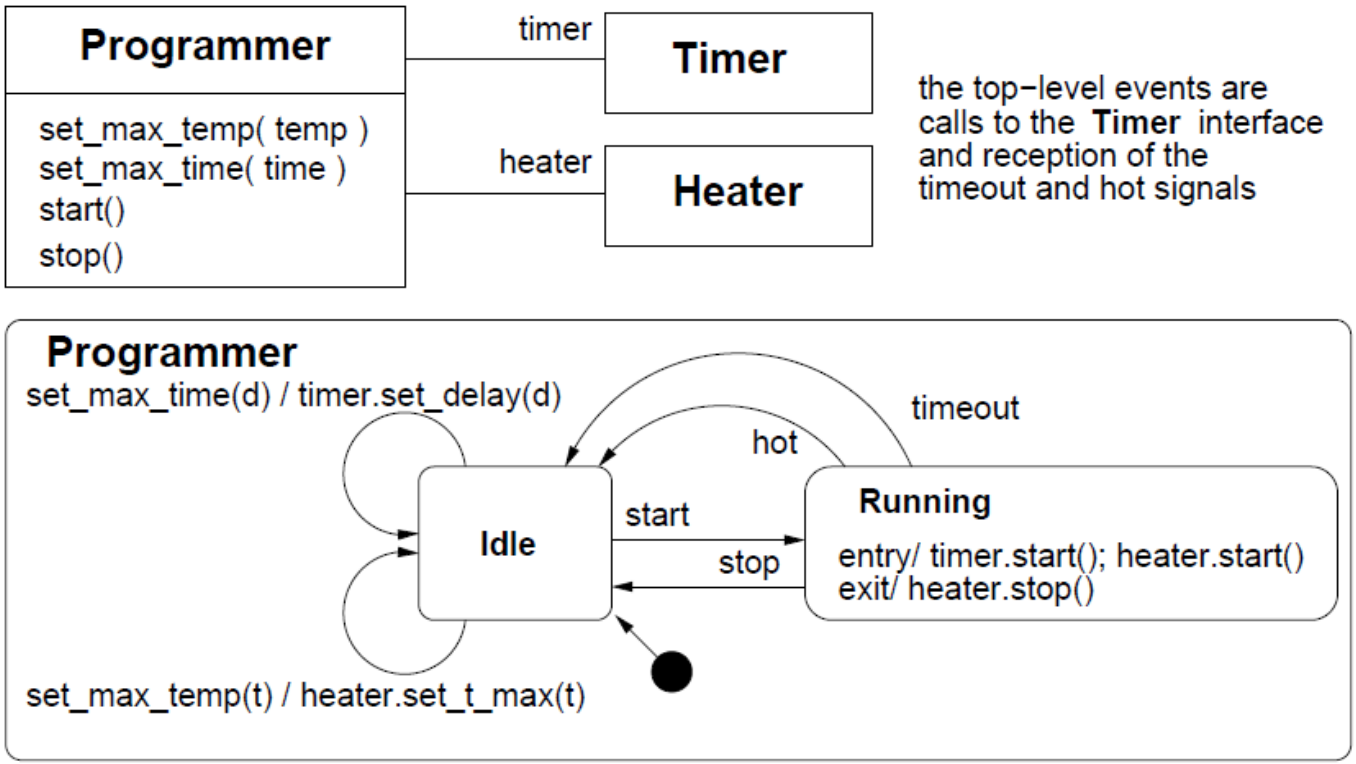
Un segnale si rappresenta come un rettangolo contenente lo stereotipo ≪signal≫, il nome del segnale ed eventuali attributi

Eventi di cambiamento



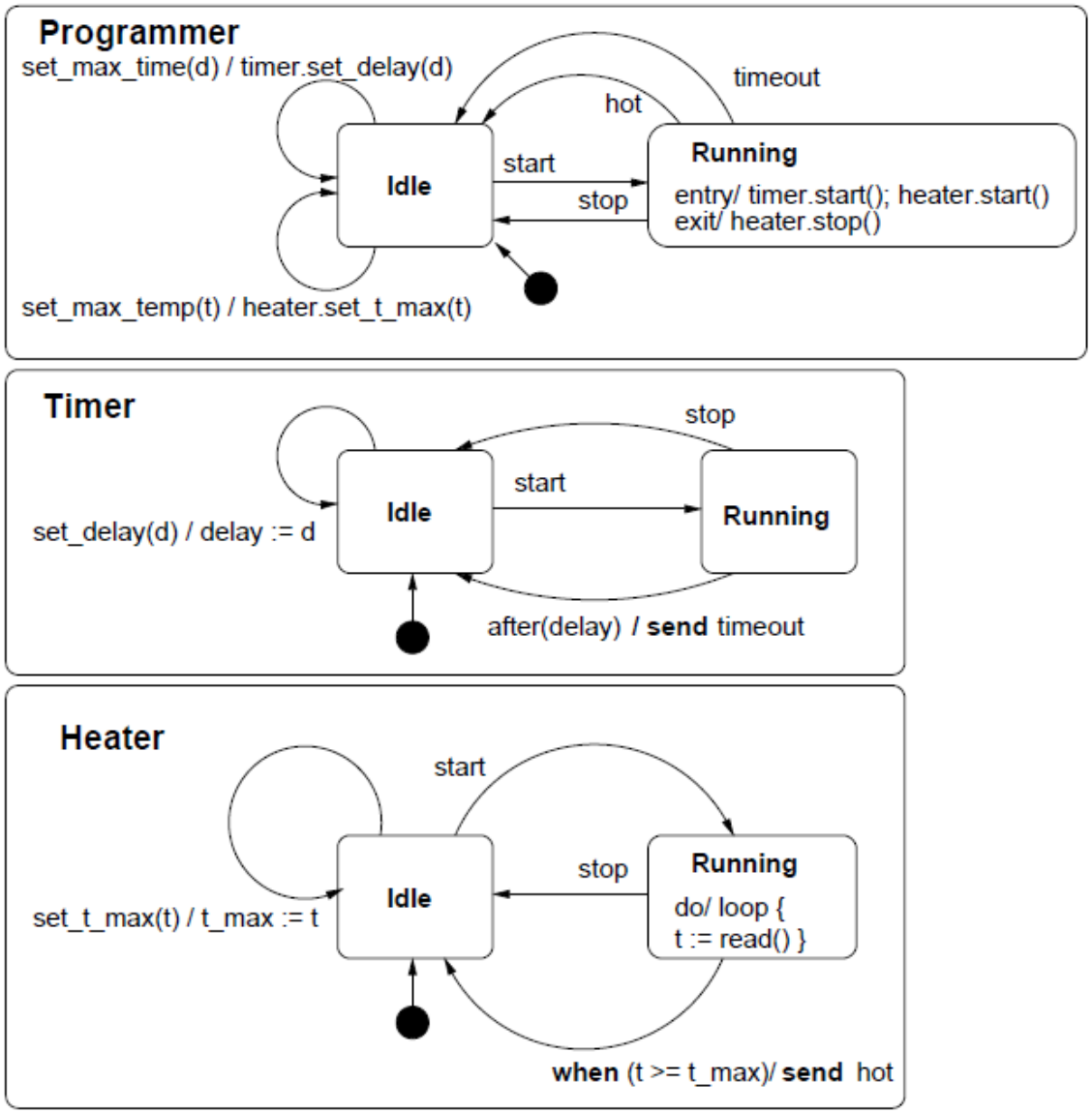
when(. . .): fire transition when a condition becomes true (change event)
do/: perform an activity while in a given state.

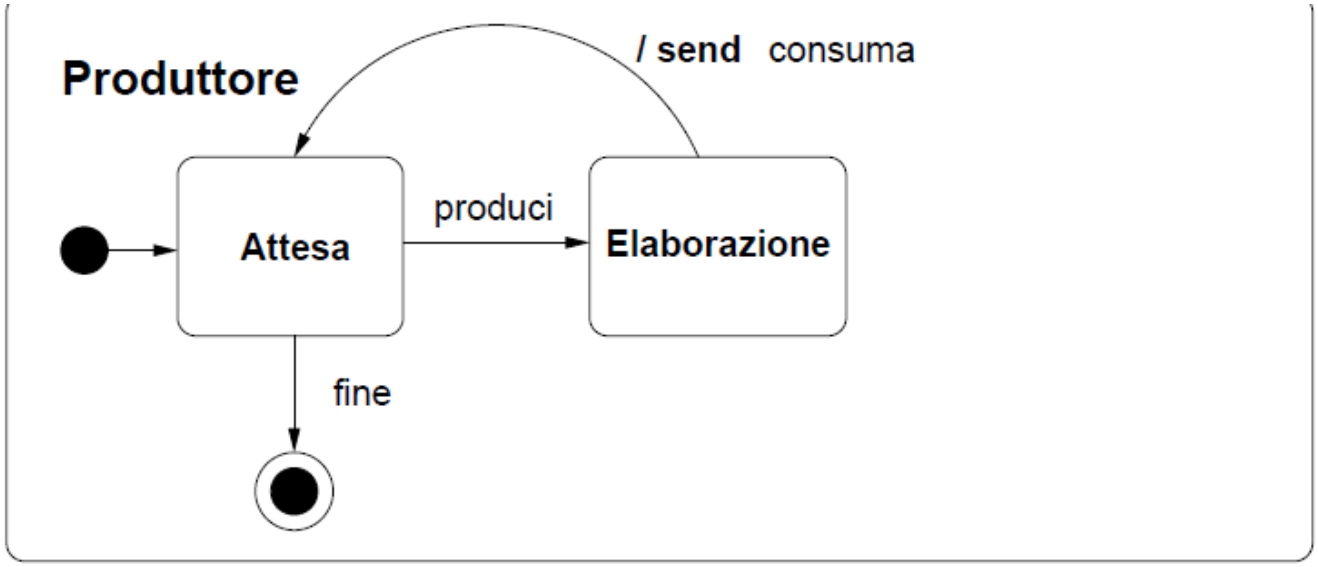
Azioni di entry ed exit

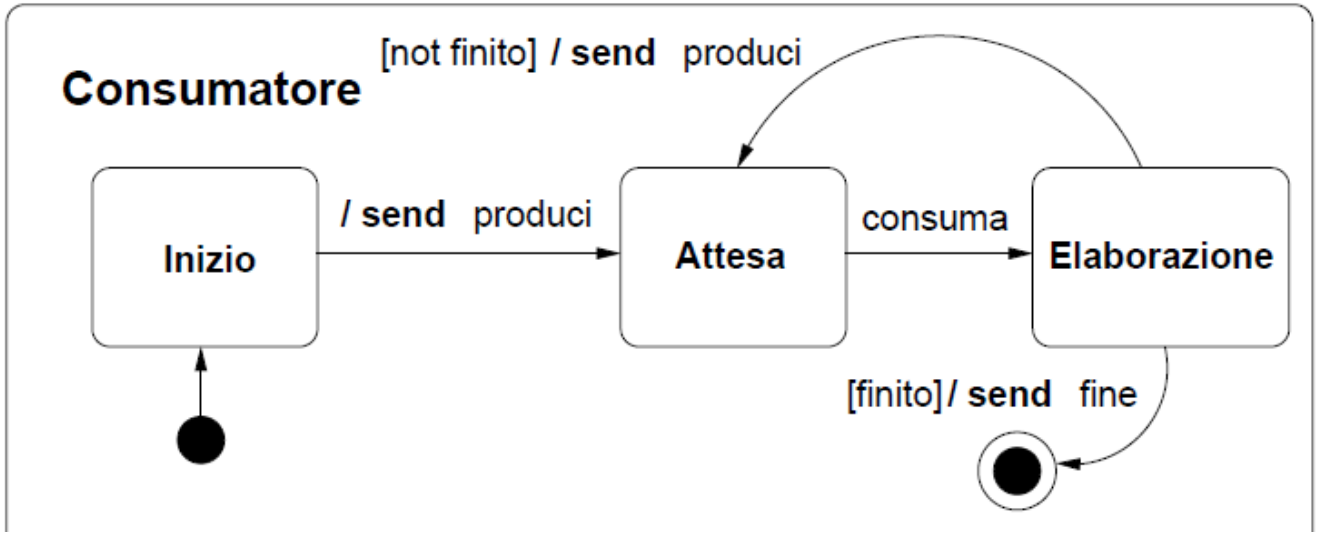


heater, timer: rolenames to identify participants in an association
entry/: action(s) performed when entering a state
exit/: action(s) performed when leaving a state

Azioni di entry ed exit




Produttore
/ send consuma
Attesa
produci
Elaborazione
fine


Consumatore
[not finito] / send produci
Inizio
/ send produci
Attesa
consuma
Elaborazione
[finito] / send fine

Gestione degli eventi

- Se un evento si verifica nel corso di una transizione, non ha influenza sull'eventuale azione associata alla transizione (ricordiamo che le azioni non sono interrompibili) e viene accantonato in una riserva di eventi (event pool) per essere considerato nello stato finale della transizione.

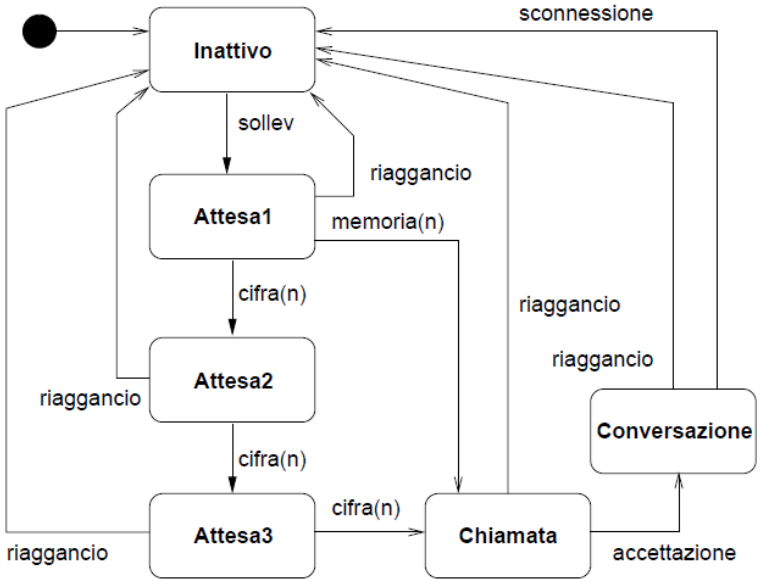
- Se nello stato finale l'evento non abilita alcuna transizione, viene cancellato.

- Se, mentre un oggetto si trova in un certo stato, si verificano degli eventi che non innescano transizioni uscenti da quello stato, l'oggetto si può comportare in due modi:
  - questi eventi vengono cancellati e quindi non potranno piú influenzare l'oggetto, oppure
  - questi eventi vengono marcati come differiti e memorizzati finché l'oggetto non entra in uno stato in cui tali eventi non sono piú marcati come differiti. In questo nuovo stato, gli eventi cosí memorizzati o innescano una transizione, o vengono perduti definitivamente.

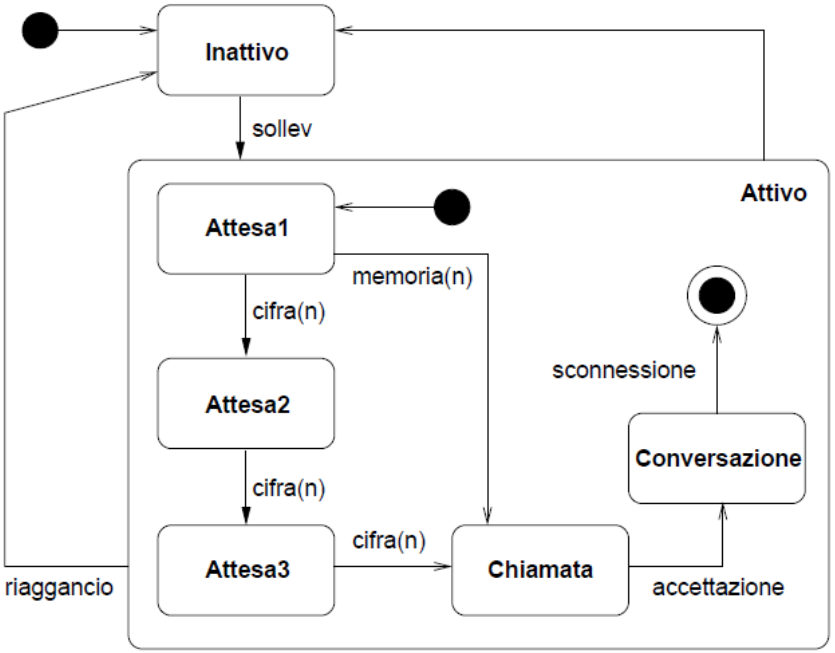
- Gli eventi differiti vengono dichiarati con la parola defer.

Macchine a stati gerarchiche

- In una macchina a stati gerarchica il comportamento del sistema in un dato stato (superstato) può essere specificato da un insieme di sottostati.
- I sottostati possono essere sequenziali (un solo stato alla volta è attivo) o concorrenti (piú stati sono attivi contemporaneamente).
- I sottostati ereditano le transizioni che coinvolgono il superstato



macchina a stati “piatta” (non gerarchica)



Macchina a stati gerarchica. I sottostati di Attivo ereditano le transizioni da Attivo a Inattivo.

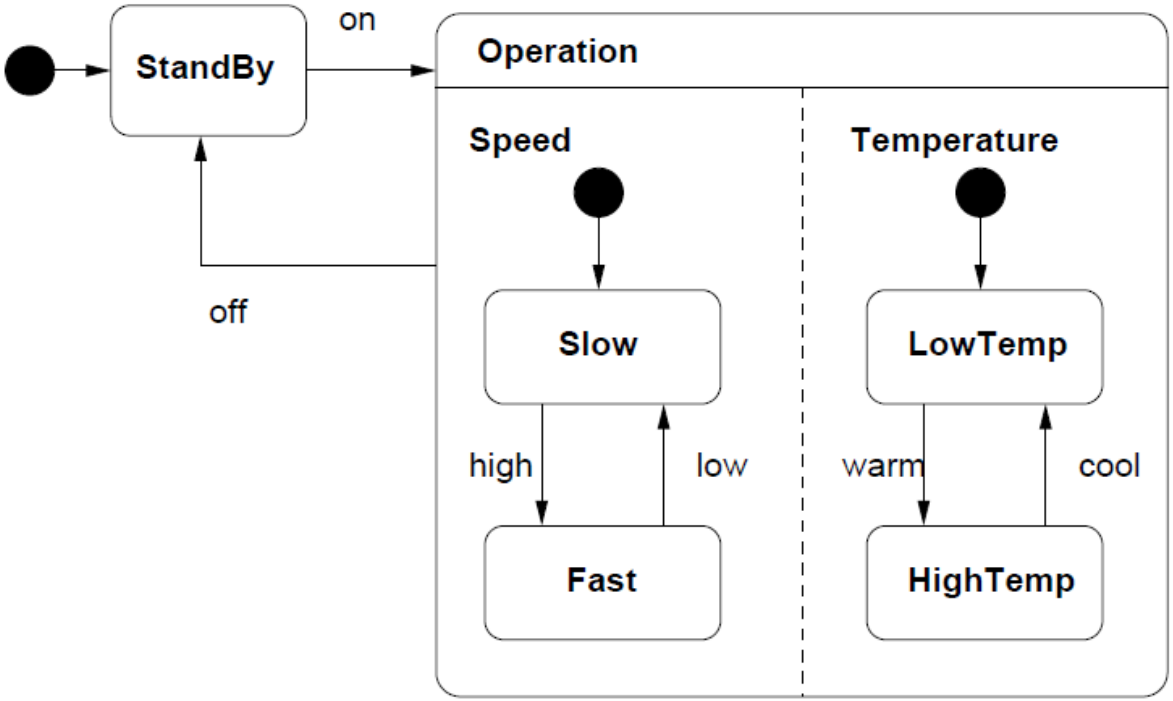
Macchine a stati gerarchiche

- La transizione in ingresso al superstato Attivo porta la macchina nel sottostato iniziale (Attesa1) di quest'ultimo.

- la transizione di completamento fra i due stati ad alto livello avviene quando la sottomacchina dello stato Attivo termina il proprio funzionamento.

- La transizione attivata dagli eventi riaggancio viene ereditata dai sottostati: questo significa che, in qualsiasi sottostato di Attivo, il riaggancio riporta la macchina nello stato Inattivo.

Regioni concorrenti



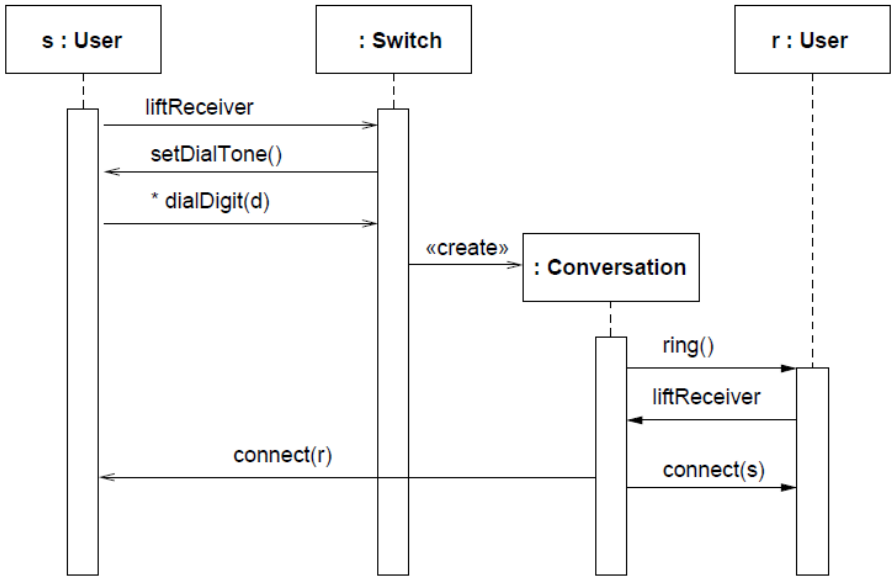
i sottosistemi per il controllo della velocità e della temperatura si evolvono in modo concorrente

UML – diagramma di sequenza

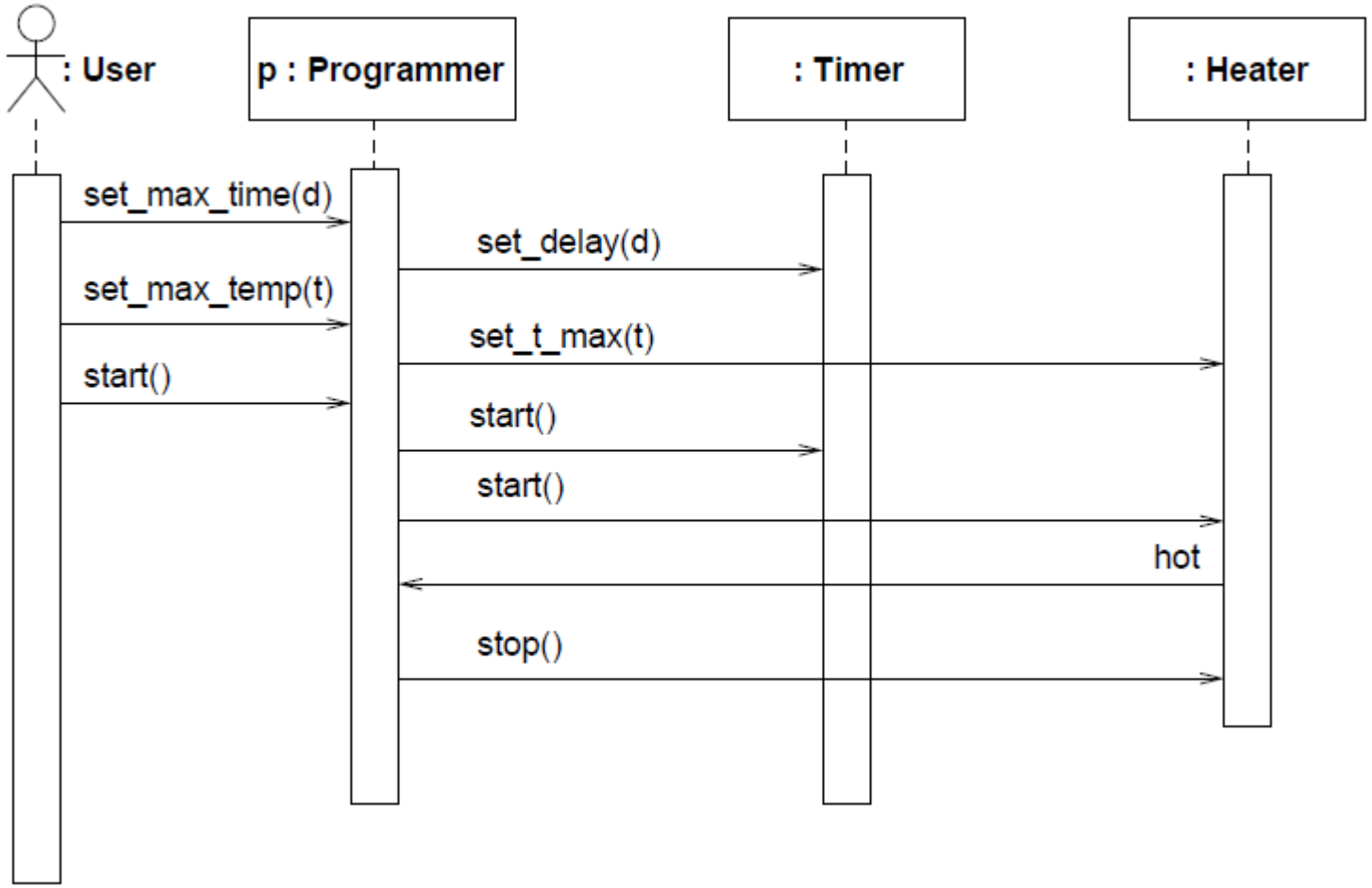
- Un diagramma di sequenza descrive l'interazione fra piú oggetti mettendo in evidenza il **flusso di messaggi** scambiati e la loro **successione temporale**.

- I diagrammi di sequenza sono quindi adatti a rappresentare degli scenari possibili nell' **evoluzione di un insieme di oggetti.**

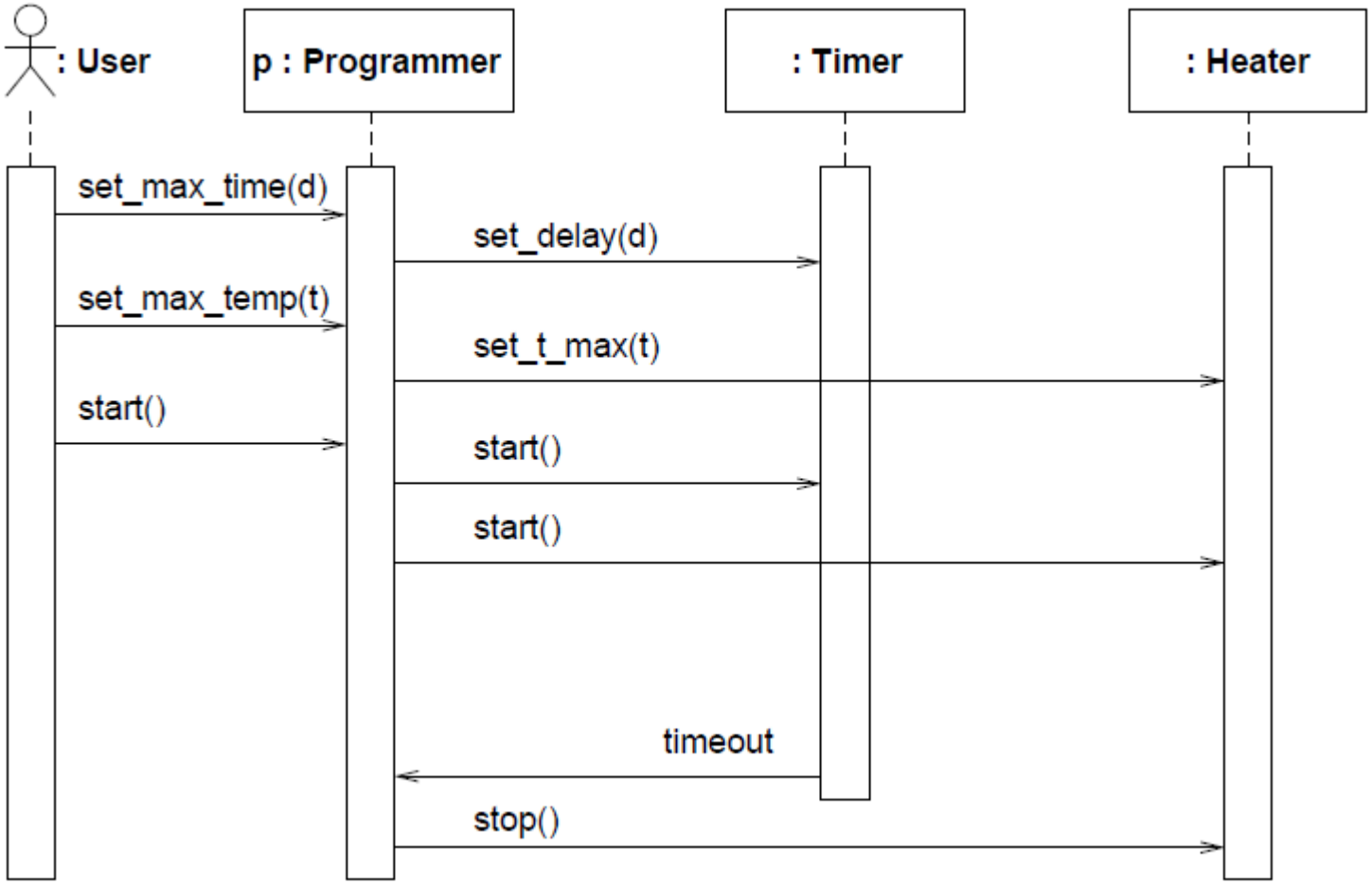
- È bene osservare che ciascun diagramma di sequenza rappresenta esplicitamente una o piú istanze delle possibili sequenze di messaggi, mentre un diagramma di stato definisce implicitamente tutte le possibili sequenze di messaggi ricevuti (eventi) o inviati (azioni send) da un oggetto interagente con altri.



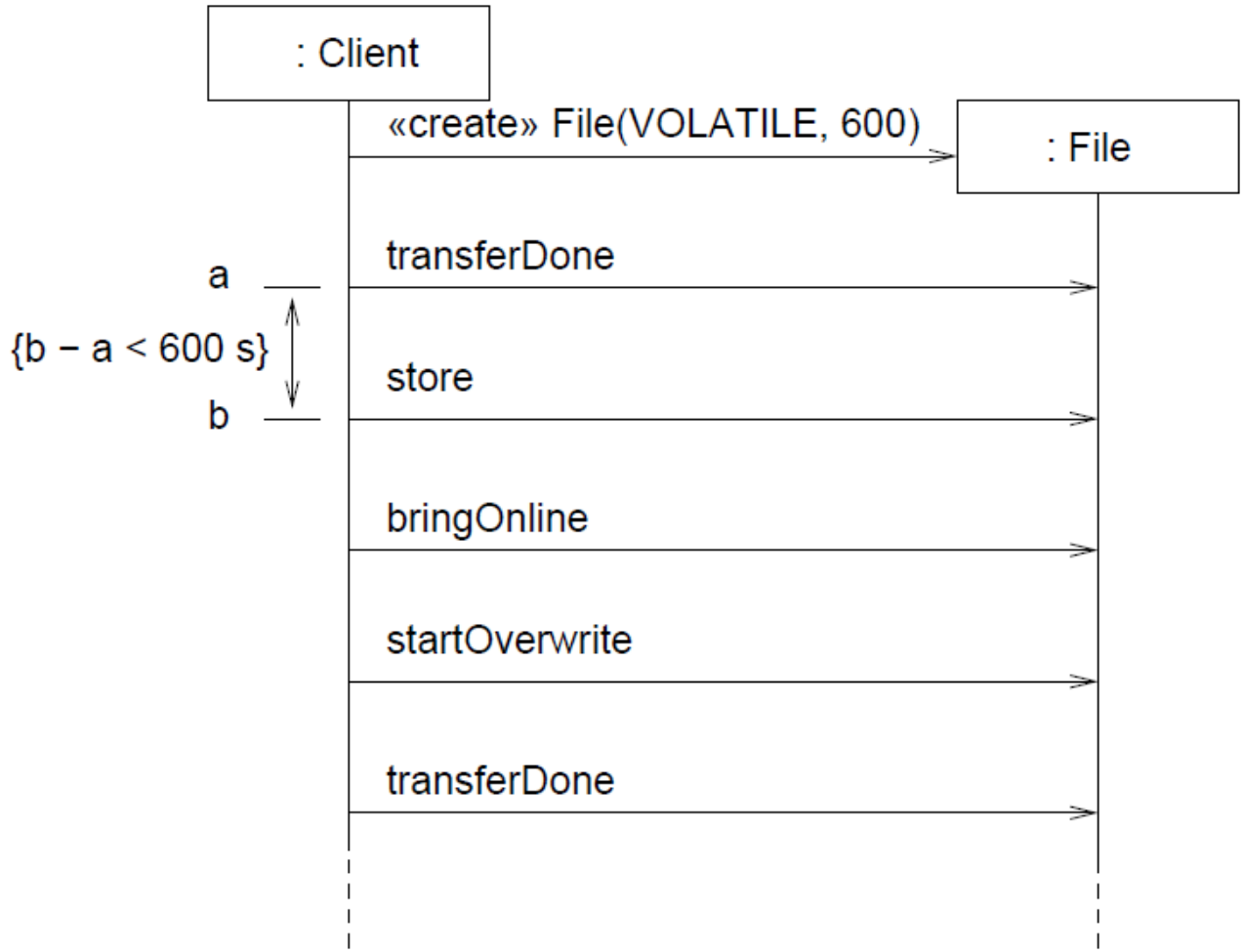
- Scenario 1: the heater reaches the limit temperature before timeout or manual stop



- Scenario 2: timeout occurs before the heater reaches the limit temperature.

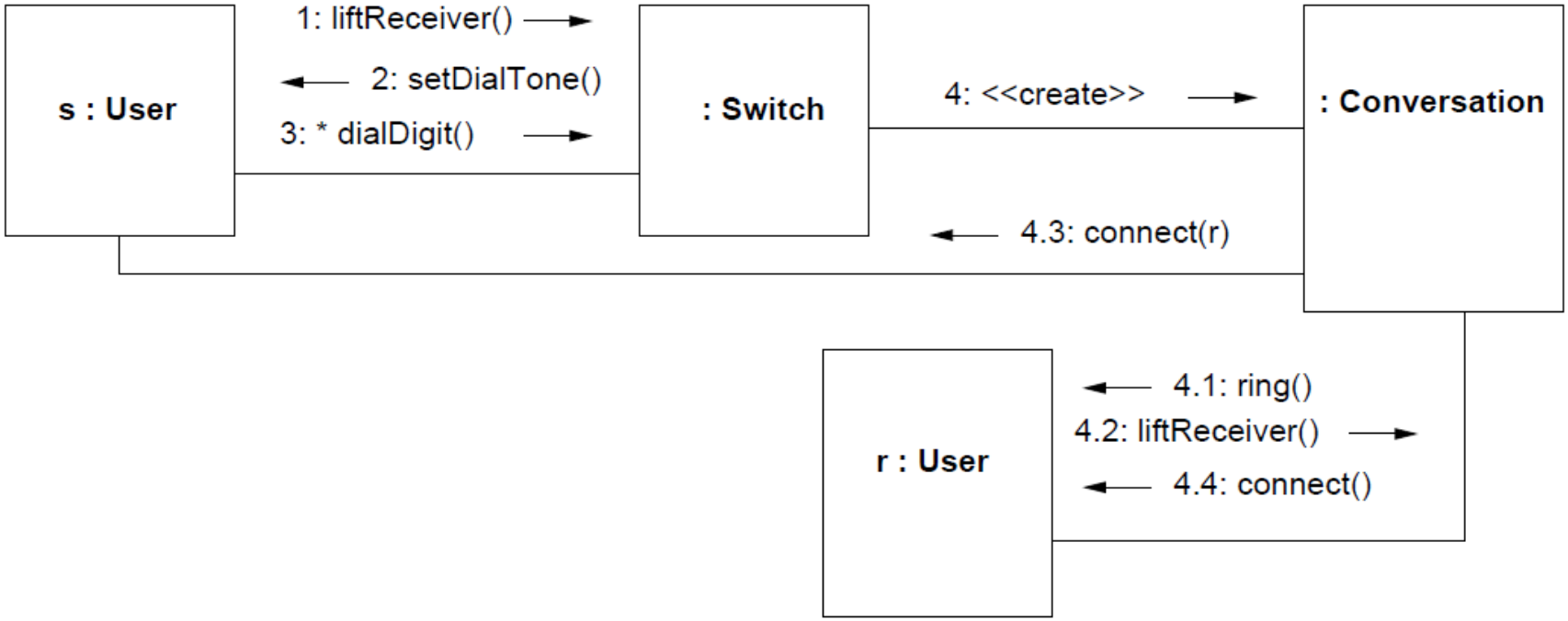


- Vincoli temporali



UML – diagramma di comunicazione

- Un diagramma di comunicazione mette in evidenza l'aspetto strutturale di un'interazione, mostrando esplicitamente i legami (istanze di associazioni) fra gli oggetti, e ricorrendo a un sistema di numerazione strutturato per indicare l'ordinamento temporale dei messaggi.



UML – diagramma di attività

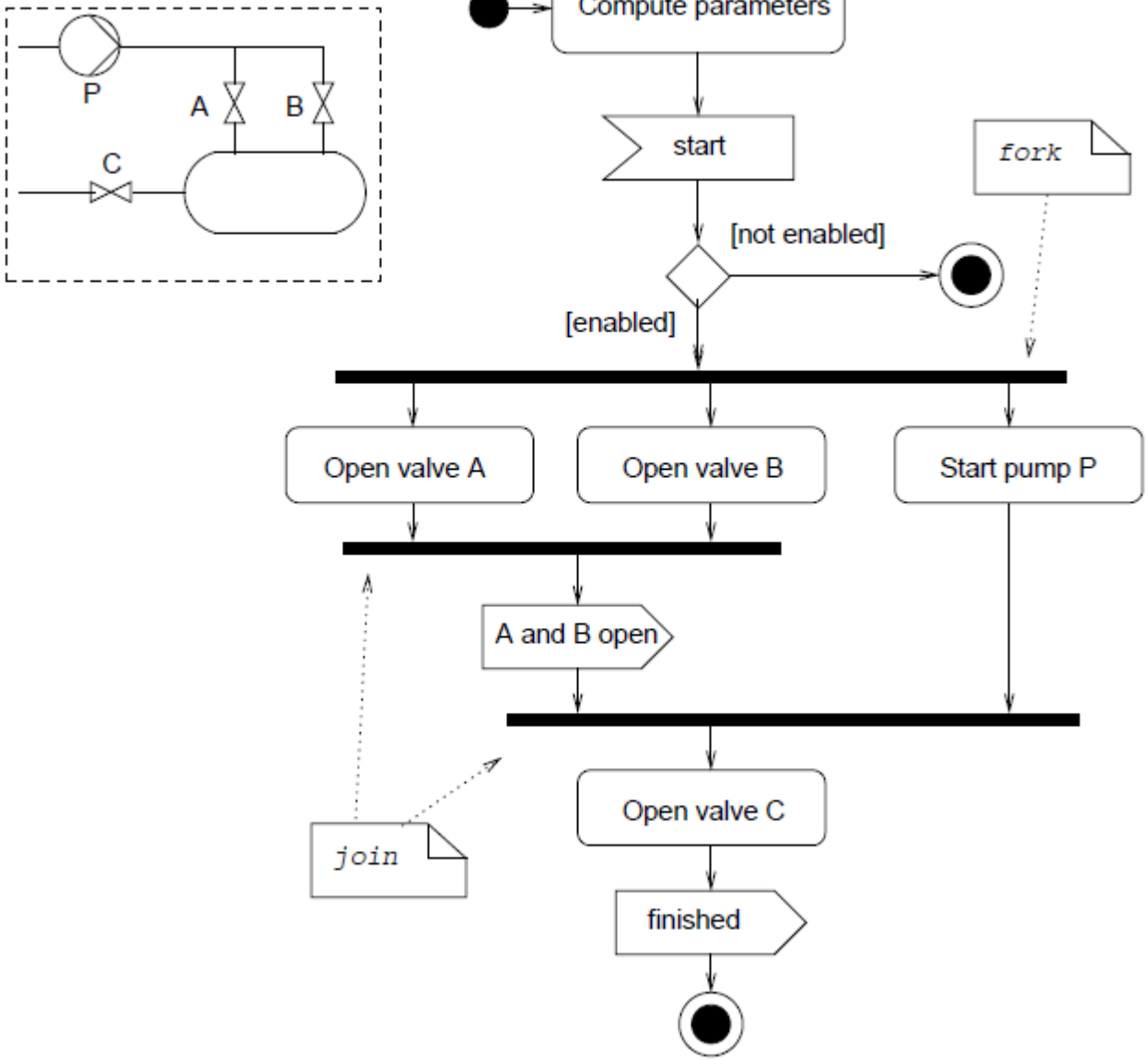
- I diagrammi di attività servono a descrivere il **flusso di controllo e di informazioni dei processi**. In un modello di analisi si usano spesso per descrivere i processi del dominio di applicazione, come, per esempio, le procedure richieste nella gestione di un'azienda, nello sviluppo di un prodotto, o nelle transazioni economiche.
- In un modello di progetto possono essere usati per **descrivere algoritmi o implementazioni di operazioni**.
- Si può osservare che, nella loro forma piú semplice, i diagrammi di attività sono molto simili ai tradizionali diagrammi di flusso (flowchart).

- Un diagramma di attività è formato da **nodi e archi**. I nodi rappresentano attività svolte in un processo, punti di controllo del flusso di esecuzione, o oggetti elaborati nel corso del processo. Gli archi collegano i nodi per rappresentare i flussi di controllo e di informazioni.

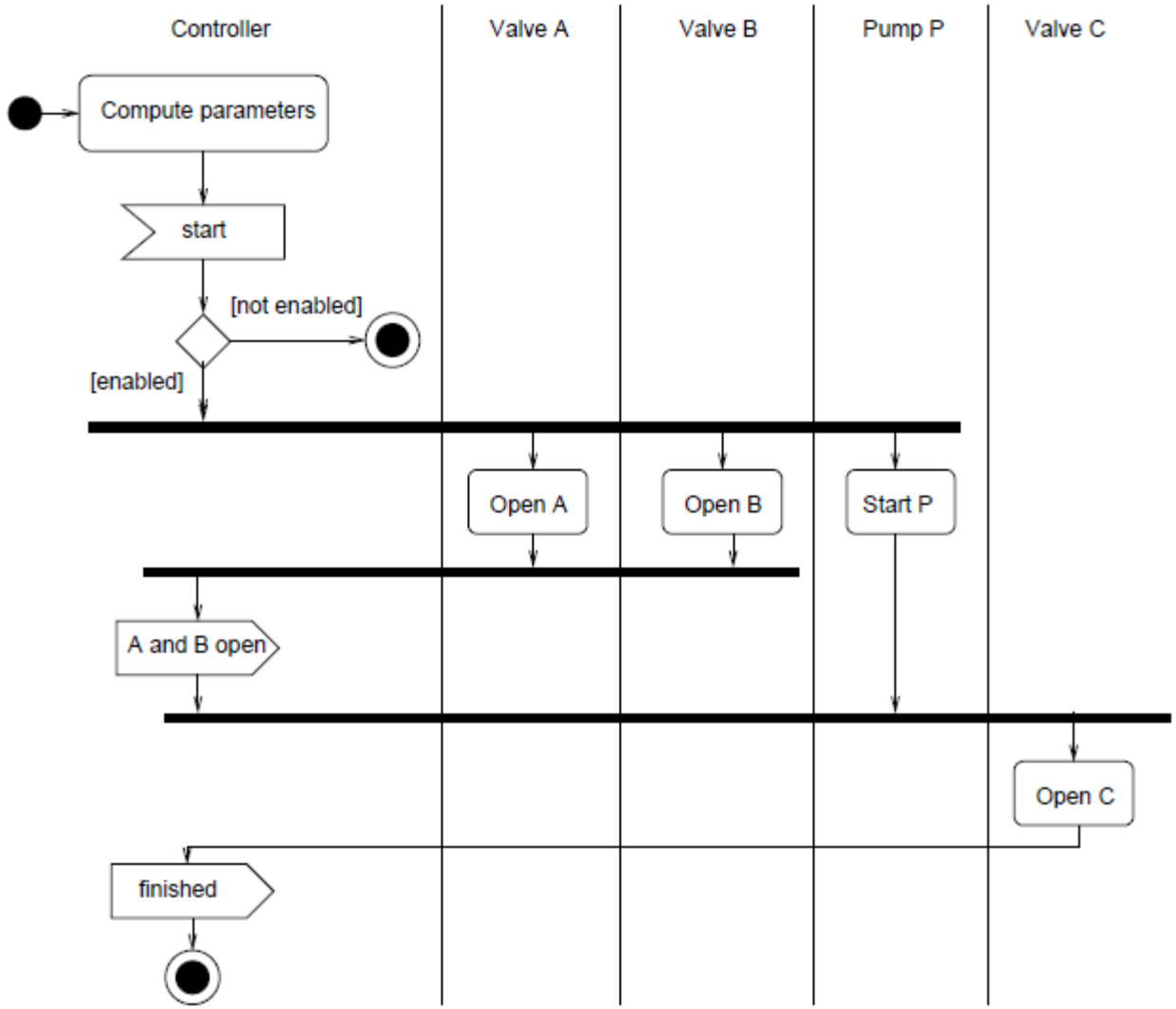
- I diagrammi di attività possono descrivere attività svolte da entità differenti, raggruppandole graficamente. Ciascuno dei gruppi cosí ottenuti è una **partizione**, detta anche corsia (swimlane).

- I lucidi successivi mostrano un processo di controllo, lo stesso processo usando le partizioni, e (in modo molto semplificato) il processo di sviluppo di un prodotto, mostrando quali reparti di un'azienda sono responsabili per le varie attività.

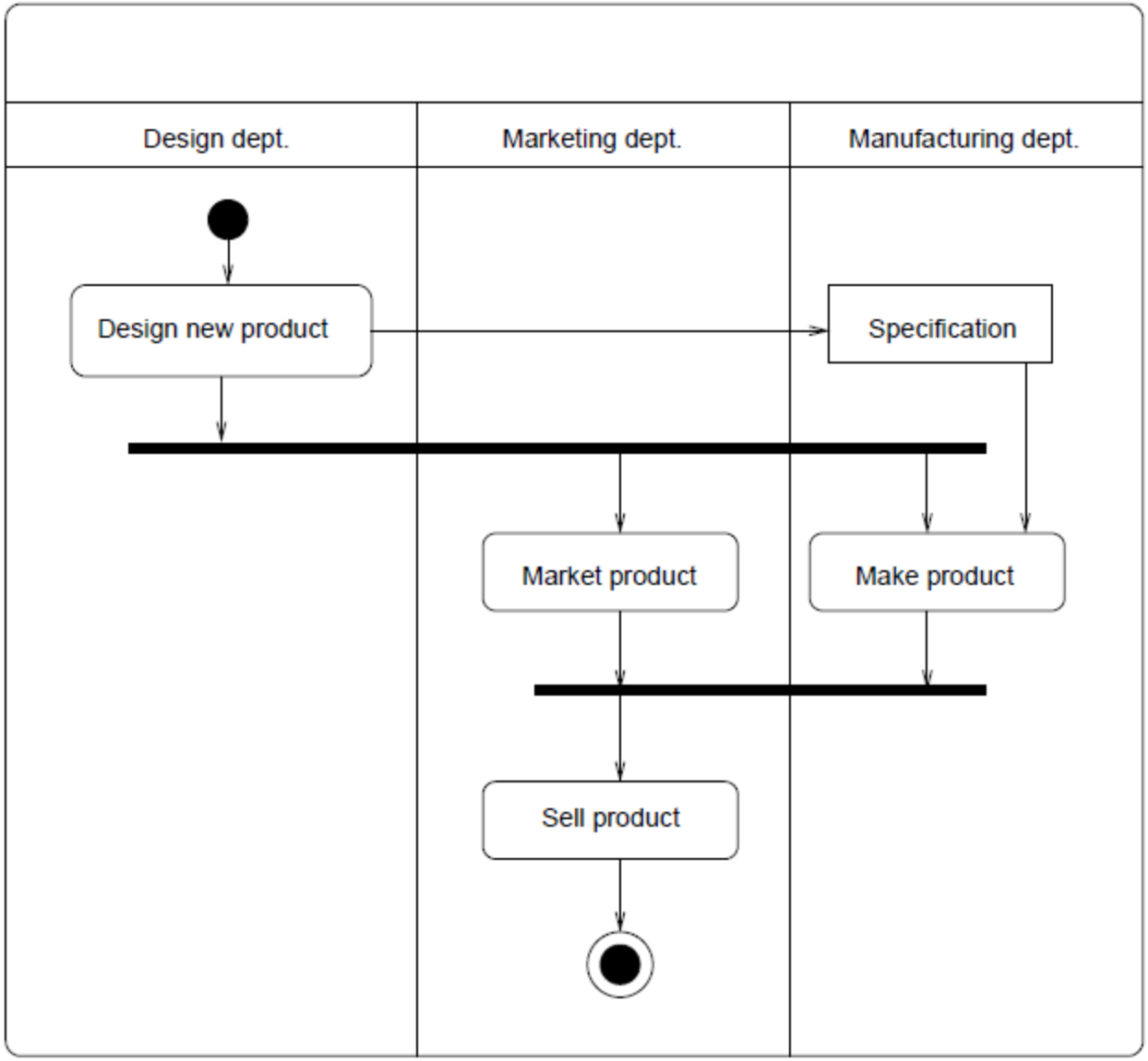
UML – diagramma di attività



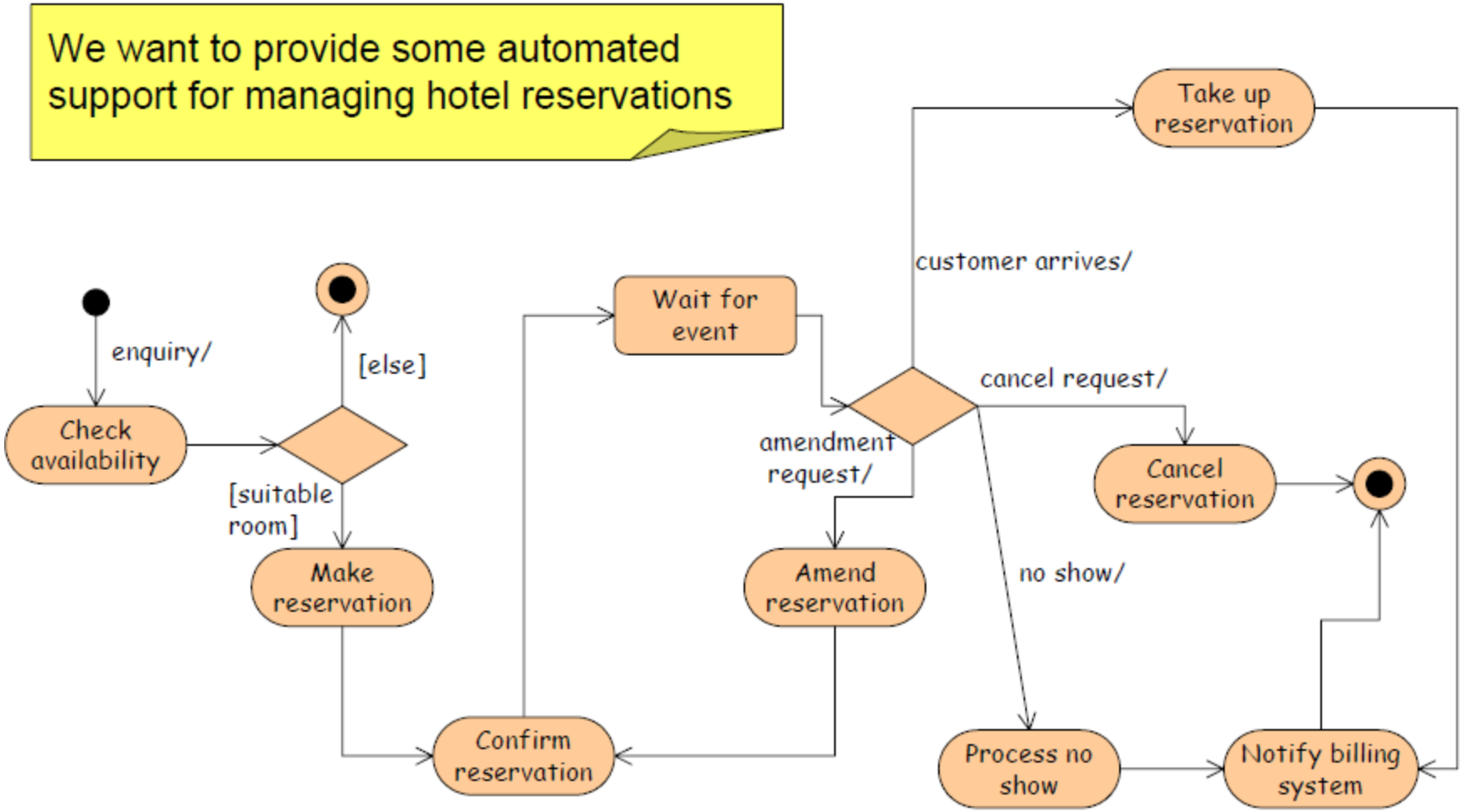
UML – diagramma di attività



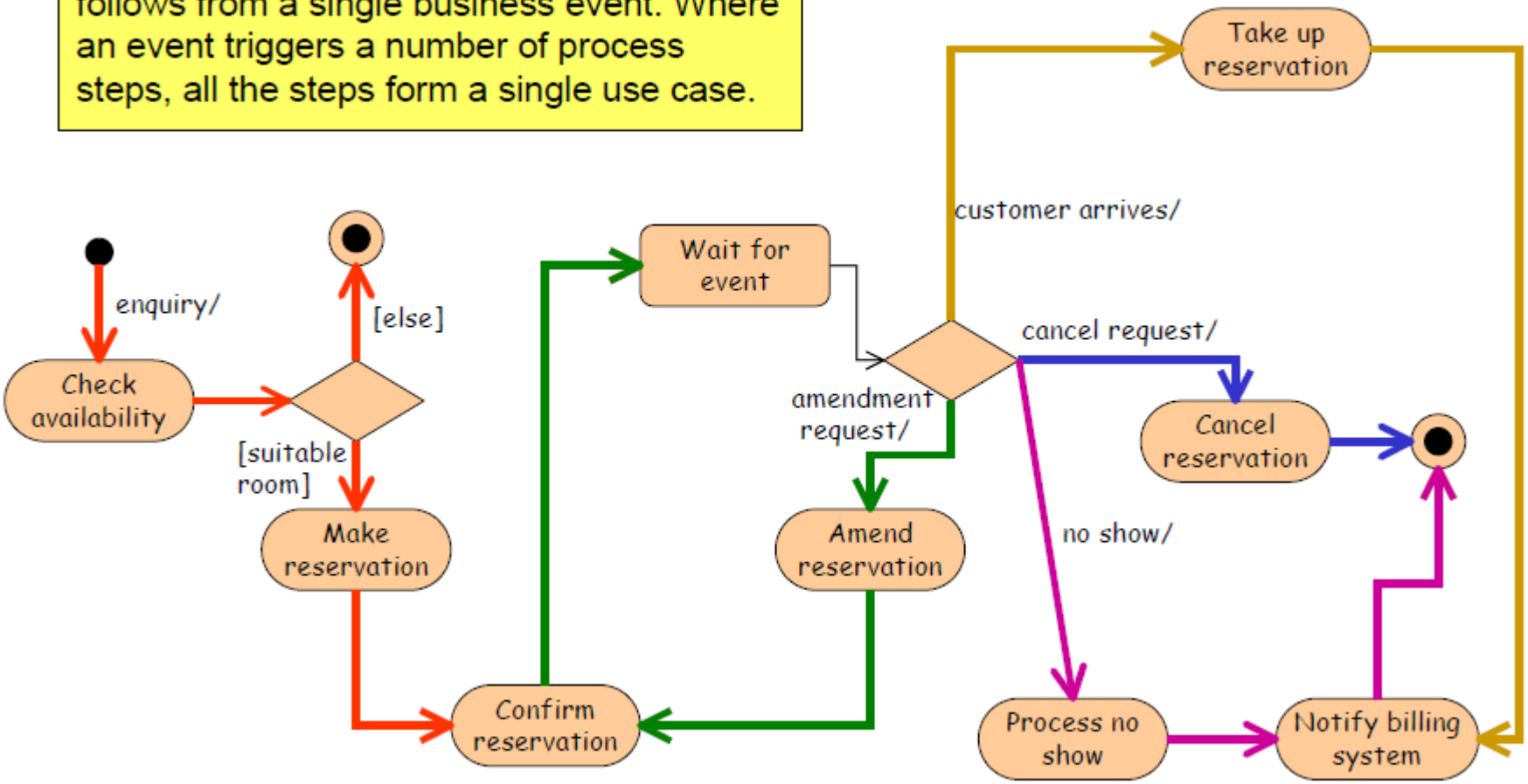
UML – diagramma di attività



UML – diagramma di attività

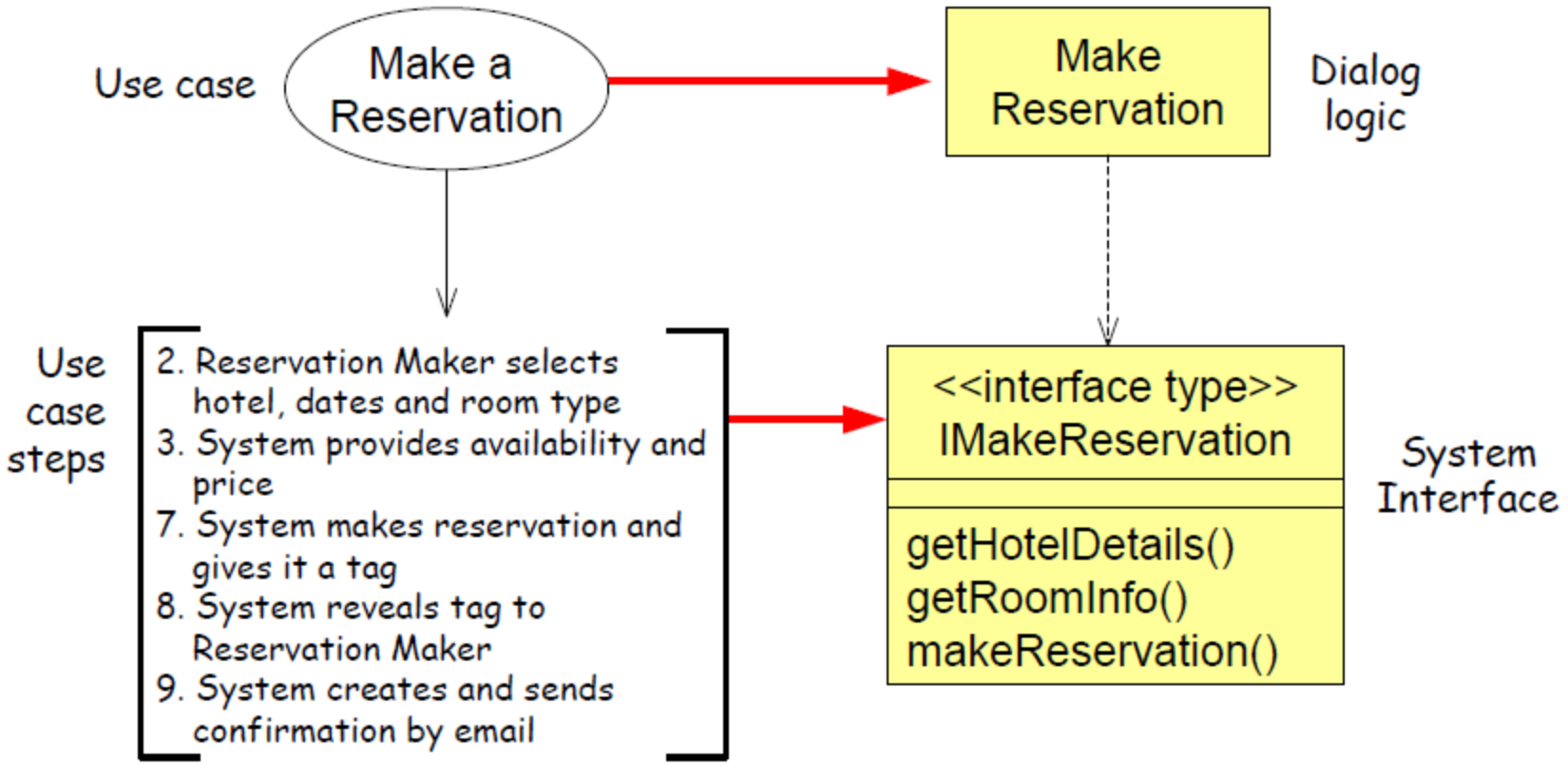


A use case describes the interaction that follows from a single business event. Where an event triggers a number of process steps, all the steps form a single use case.



Un caso d'uso dovrebbe riguardare sempre e solo una determinata attività.

UML – diagramma delle cllasi e casi d'uso



Cose da includere nella descrizione di un progetto sw

- architettura logica del sw
  - moduli:
    - sottosistemi;
    - componenti;
    - moduli unitari.
  - relazioni:
    - associazioni;
    - generalizzazione;
    - dipendenze;
    - collaborazioni.

- architettura fisica
  - del sw:
    - file eseguibili;
    - librerie;
    - dati;
    - . . . .
  - dello hw:
    - nodi computazionali;
    - ambienti di esecuzione.

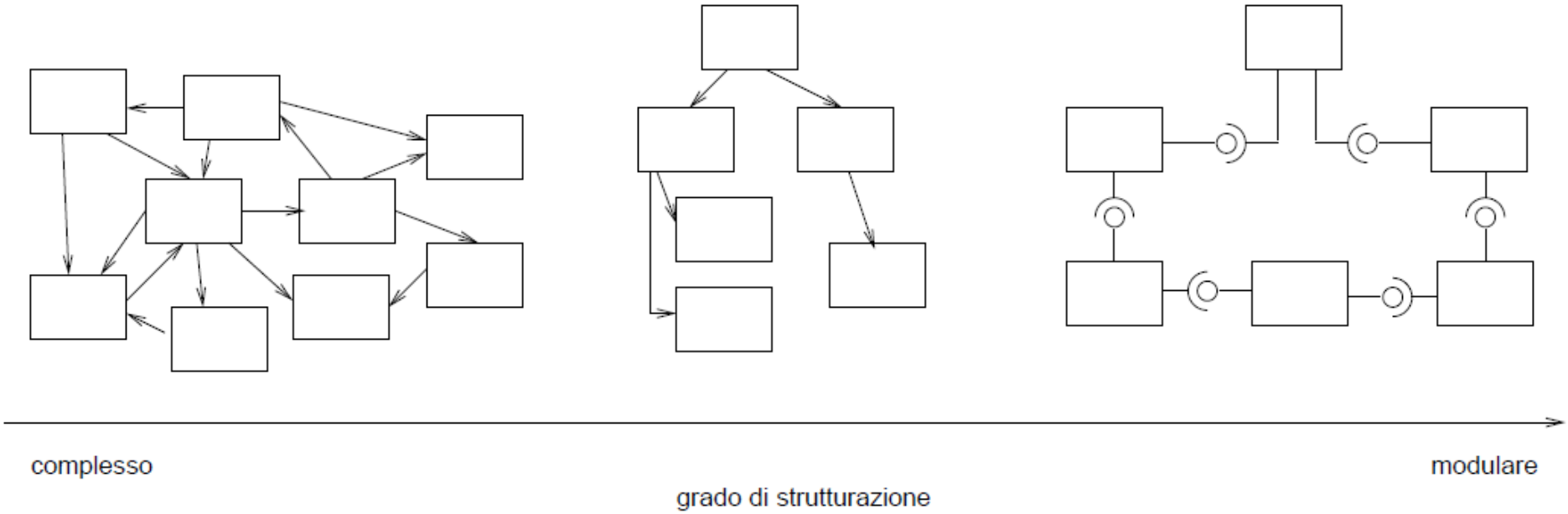
Proprietà

- **correttezza funzionale** del prodotto;
  - **rispetto dei requisiti**, non funzionali del prodotto (tradotti, se possibile, nei requisiti, funzionali);
  - **affidabilità** del prodotto;
  - **modificabilità** del prodotto e del progetto (“design for change”);
  - **riusabilità** del prodotto e del progetto;
  - **verificabilità** del progetto;
  - **comprensibilità** del progetto;
  - . . .

- cioè **modularità** del progetto.

UML - Modularità

- Un sistema è modulare se è composto da un certo numero di sottosistemi, ciascuno dei quali svolge un compito ben definito e dipende dagli altri in modo semplice.



UML - Modularità

- Un **modulo** è una porzione di software che contiene e fornisce risorse o servizi e a sua volta può usare risorse o servizi offerti da altri moduli.

- Un modulo viene quindi caratterizzato dalla sua **interfaccia**, cioè dall'elenco dei **servizi offerti** (o esportati) e **richiesti** (o importati).

- Le **risorse** offerte da un modulo possono essere operazioni, strutture dati, e definizioni.

- **L'interfaccia** di un modulo è una specifica, che viene realizzata dall'implementazione del modulo.
  - offerta (provided) o richiesta (required)
  - comprende
    - lista di operazioni non private (pubbliche, protette, o "package")
    - pre- e postcondizioni
    - eccezioni
    - protocollo (definito p.es. da una protocol machine)
  - implementazione
    - semplice (modulo unitario)
    - composta (modulo formato da sottomoduli)
- N.B.: spesso per "interfaccia" s'intende solo una lista di operazioni, e in particolare le operazioni offerte.

Ripartizione delle responsabilità

- Bisogna suddividere il lavoro svolto dal sistema (e ricorsivamente da ciascun sottosistema) fra i vari moduli, in modo che a ciascuno di essi venga affidato un **compito ben definito e limitato** ("do one thing well").

- I moduli progettati secondo questo criterio hanno la proprietà della coesione, cioè di offrire un insieme omogeneo di servizi.
  - migliore **comprensibilità** dei singoli moduli e dell'architettura;
  - migliore **modificabilità** dei singoli moduli e dell'architettura;
  - migliore **riusabilità** dei singoli moduli.

- L'interfaccia può essere considerata un **contratto** fra cliente e fornitore (design by contract).
  - Il contratto viene espresso in termini di precondizioni e postcondizioni.
    - le **precondizioni** sono responsabilità del cliente
      - il cliente garantisce che le precondizioni valgano prima di chiedere un servizio;
    - le **postcondizioni** sono responsabilità del fornitore
      - il fornitore garantisce, fornendo il servizio, che le postcondizioni valgano dopo la fornitura del servizio

Ripartizione delle responsabilità

- Se si prevede che nello svolgimento di un servizio si possano verificare delle **situazioni anomale**, bisogna decidere se tali situazioni possono essere gestite nel modulo fornitore, nascondendone gli effetti ai moduli clienti, oppure se il modulo fornitore debba limitarsi a segnalare il problema, delegandone la gestione ai moduli clienti.

- A questo scopo è stato sviluppato il meccanismo delle **eccezioni** nei linguaggi di programmazione.

Information hiding

- Bisogna rendere inaccessibile dall'esterno tutto ciò che non è strettamente necessario all'interazione con gli altri moduli, in modo che vengano ridotte al minimo le dipendenze e quindi sia possibile progettare, implementare e collaudare ciascun modulo indipendentemente dagli altri.

- I servizi offerti dal modulo richiedono varie strutture dati ed operazioni (o altre risorse, magari piú astratte, come tipi di dati o politiche di gestione di certe risorse). Il progettista deve decidere quali di queste entità devono far parte dell'interfaccia, ed essere cioè accessibili dall'esterno.

- Le entità che non sono parte dell'interfaccia servono unicamente a implementare le altre, e non devono essere accessibili.
  - ciò che non è accessibile non crea dipendenze;
  - ciò che non è accessibile può essere modificato isolatamente dagli altri moduli..

Incapsulamento

- Incapsulare significa, approssimativamente, raggruppare alcuni elementi isolandoli dal resto del sistema e definire l'interfaccia di tale gruppo, ascondendo le parti che non vi appartengono.

- L'incapsulamento è quindi l'applicazione del principio di information hiding.

- Nei prossimi lucidi vedremo che in UML gli elementi di modello per rappresentare l'incapsulamento sono le classi, gli elementi ≪interface≫, i package e i componenti.

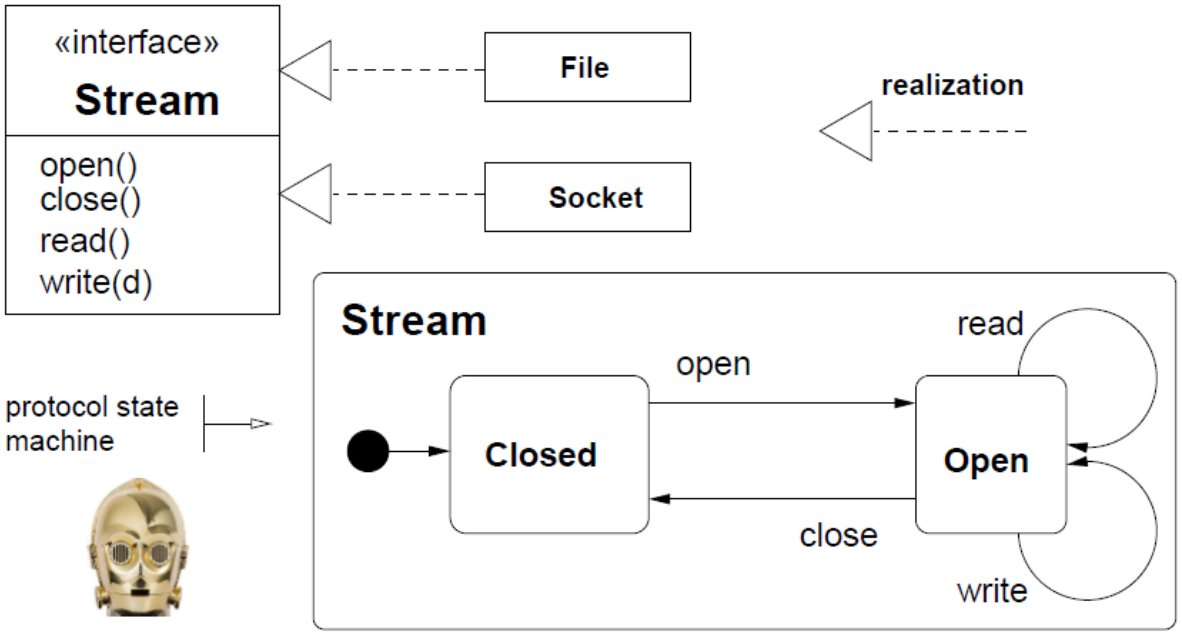
- **classi**
  - In un modello di progetto si devono specificare dettagliatamente le operazioni visibili delle classi, indicando tipi, argomenti, valori restituiti e visibilità.
  - Le operazioni e gli attributi con visibilità privata generalmente vengono aggiunte nella fase di codifica o di progetto dettagliato.

Incapsulamento

- **Classi astratte**
  - Salvo indicazioni contrarie, si presuppone che ogni operazione debba essere implementata da un metodo, sia cioè concreta.
  - Spesso è utile dichiarare in una classe delle operazioni che non devono essere implementate nella classe stessa, ma in classi derivate.
  - Tali operazioni si dicono astratte.
  - Una classe che contenga almeno una operazione astratta si dice astratta.
  - In un modello di progetto, “astratta” significa “non implementabile direttamente”.
  - Le operazioni e le classi astratte hanno la proprietà {abstract}. Graficamente, questa proprietà viene espressa scrivendo in caratteri obliqui il nome dell’elemento interessato.

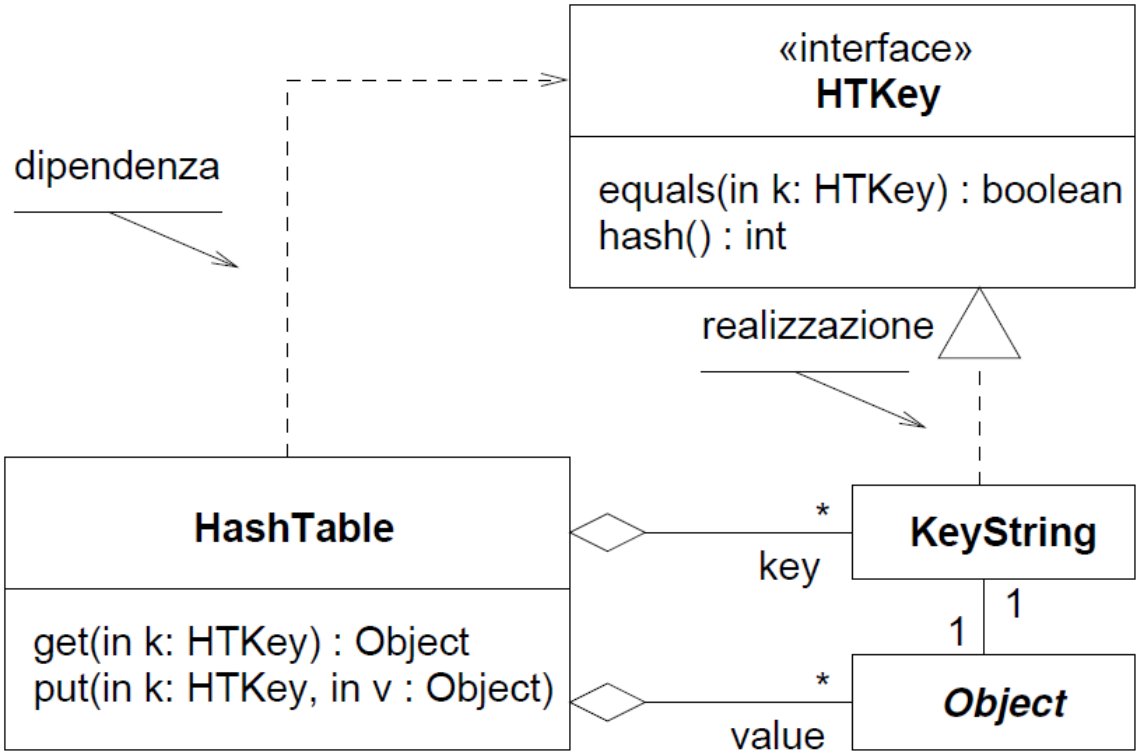
Incapsulamento

- **elementi ≪interface≫**
  - Un'interfaccia si può rappresentare separatamente dal modulo che la implementa, usando l'elemento ≪interface≫.
  - Un'interfaccia può avere piú implementazioni alternative.
  - L'elemento ≪interface≫ è un elenco di operazioni con visibilità pubblica, a cui si possono aggiungere dei vincoli e un protocollo.



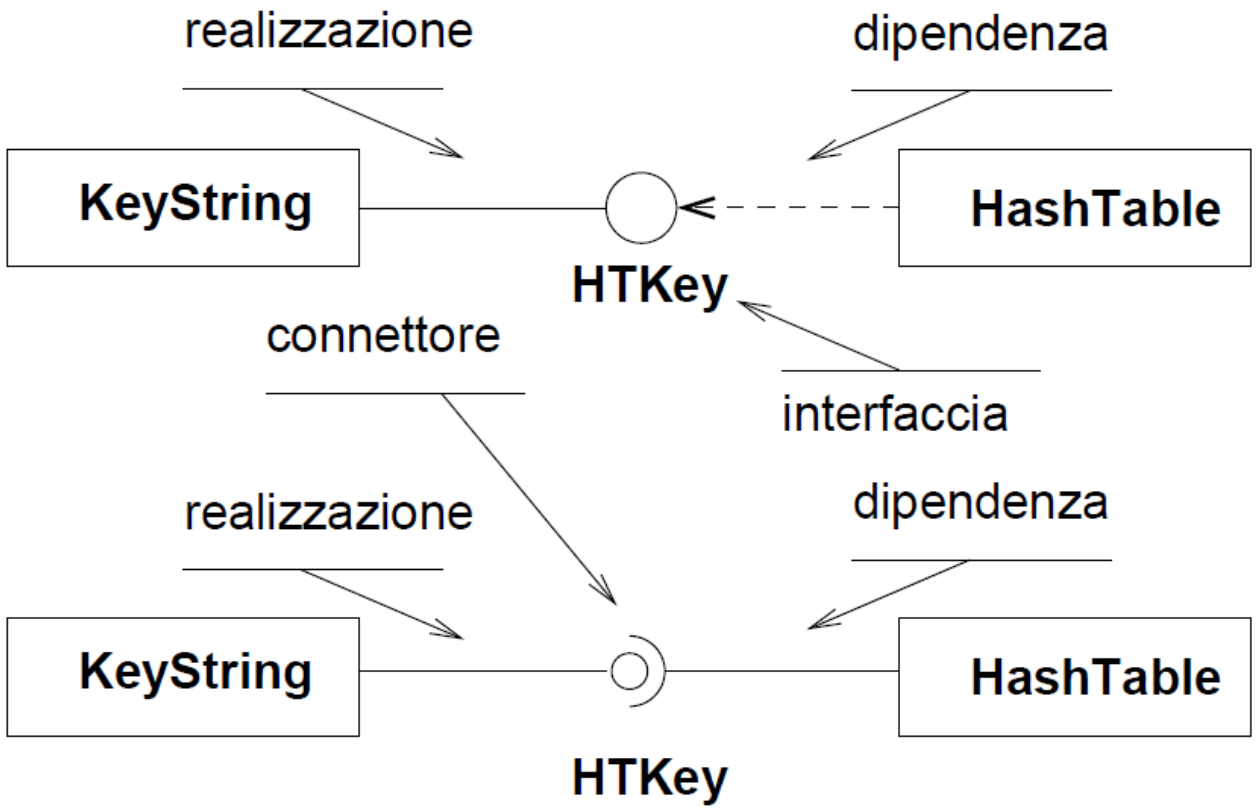
Incapsulamento

- **elementi ≪interface≫**



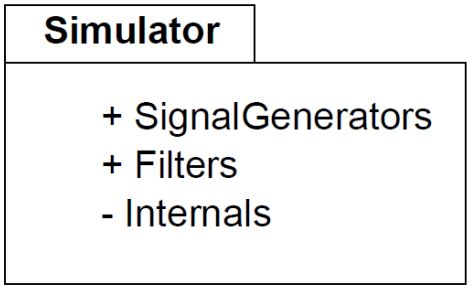
Incapsulamento

- **elementi ≪interface≫**



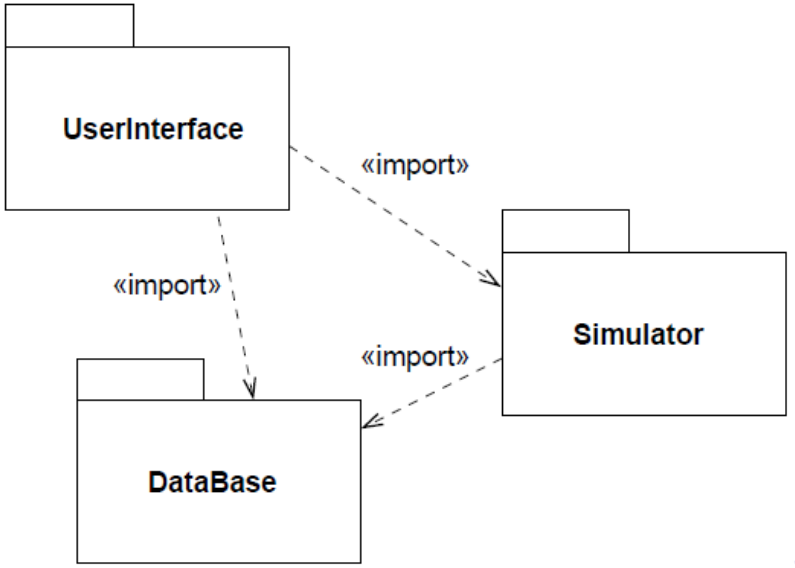
Incapsulamento

- **package**
  - Un package serve a organizzare elementi di modello;
  - può non avere una corrispondenza diretta con la struttura dell'architettura software;
  - quando viene usato per modellare un (sotto)sistema software, si chiama "interfaccia" l'insieme dei suoi elementi visibili:



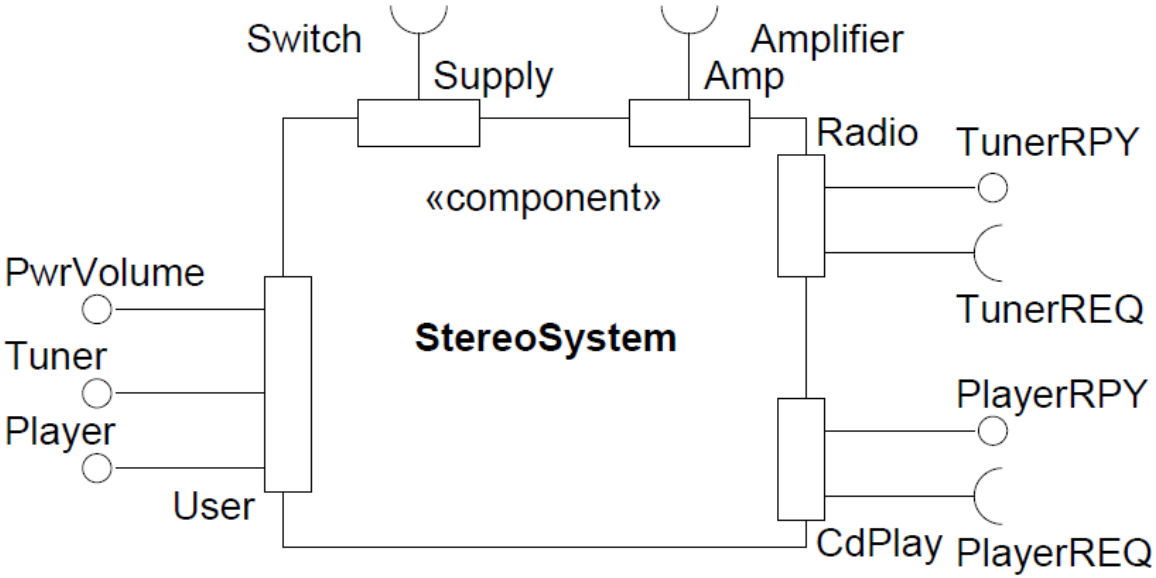
Incapsulamento

- **package**
  - l'interfaccia di un package è l'unione delle interfacce dei suoi elementi visibili;
  - la dipendenza di uso di un package A da un package B significa che almeno un elemento posseduto da A dipende da almeno un elemento di B.
  - un package è uno spazio di nomi;
  - la dipendenza di importazione di un packacge B da un package A significa che lo spazio di nomi di B viene incluso nello spazio di A (come using in C++).



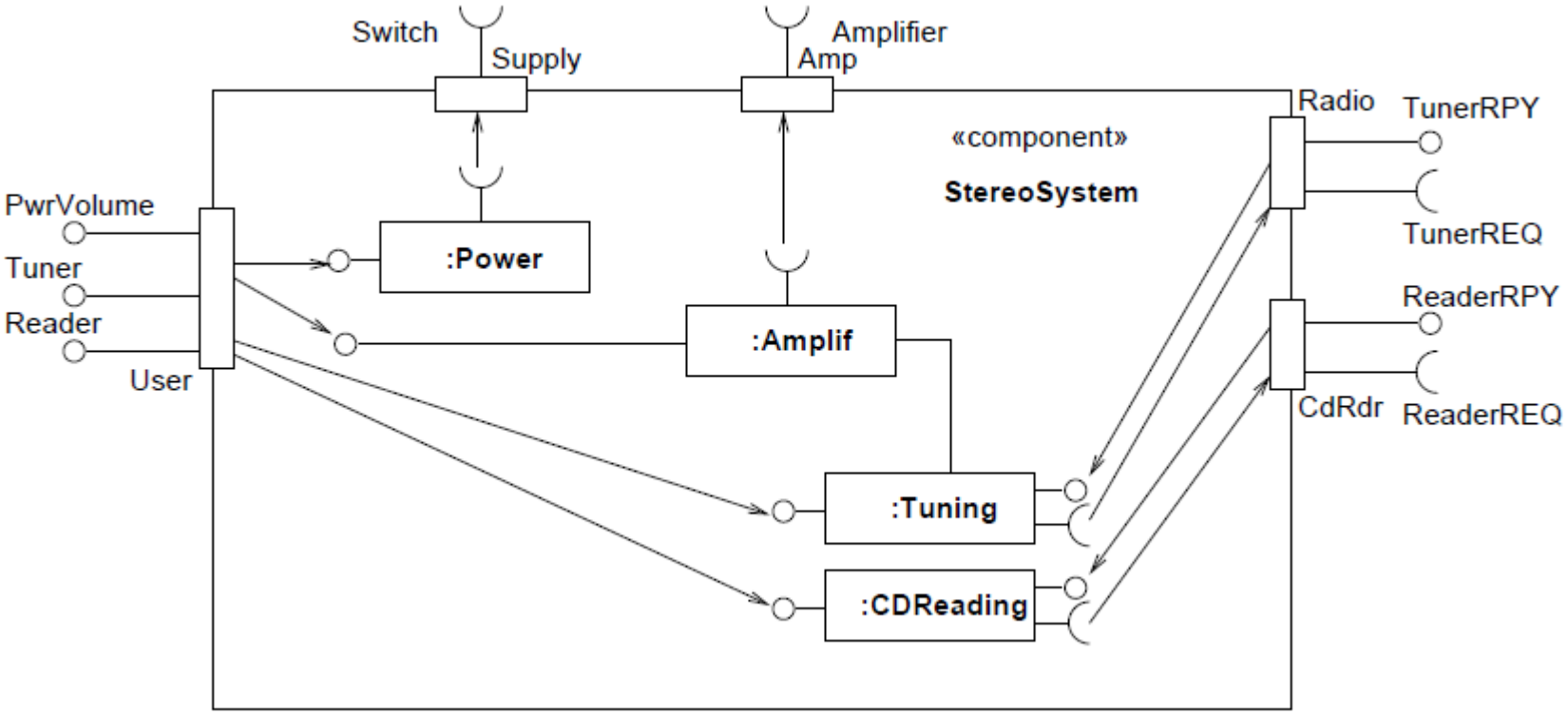
Incapsulamento

- **Componenti**
  - Un componente è un modulo logico definito da una o piú interfacce offerte e da una o piú interfacce richieste, sostituibile e riusabile.
  - Un port è un gruppo di interfacce usate per interagire con un altro specifico componente (p.es., i port User, Supply, Amp, Radio e CdPlay).



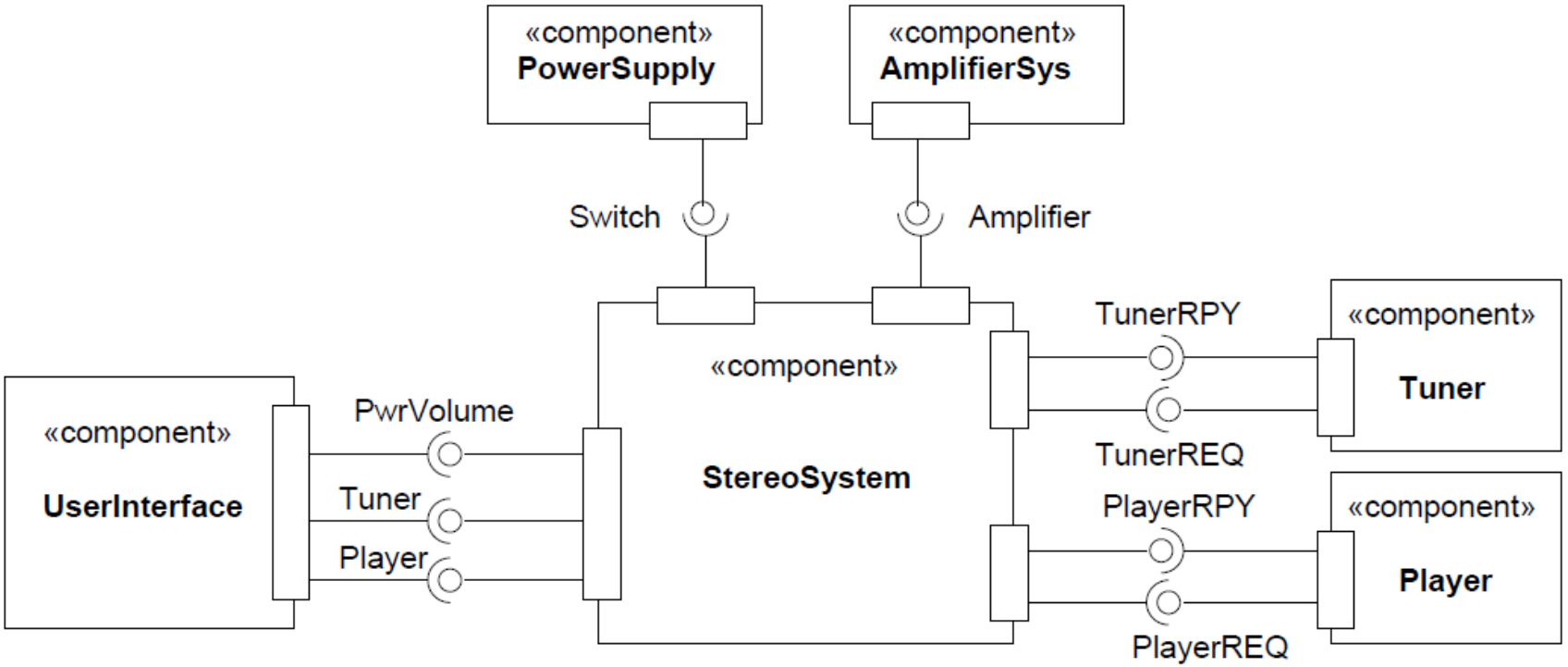
Incapsulamento

- **Componenti**
  - Un componente può essere realizzato direttamente da un'istanza di una classe, o indirettamente da istanze di piú classi o componenti.
  - La relazione di delega mostra la provenienza o la destinazione delle comunicazioni (chiamate di operazioni e trasmissioni di eventi) passanti attraverso i port.



Incapsulamento

- **Componenti**

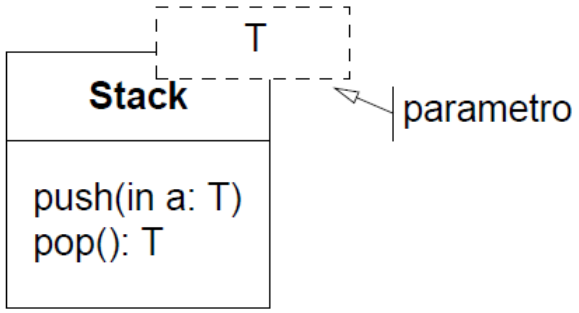


Moduli generici

- I moduli generici (**template** in UML) o parametrici permettono di "mettere a fattor comune" la struttura di algoritmi e tipi di dato, mettendo in evidenza la **parte variabile** che viene rappresentata **dai parametri del modulo**.

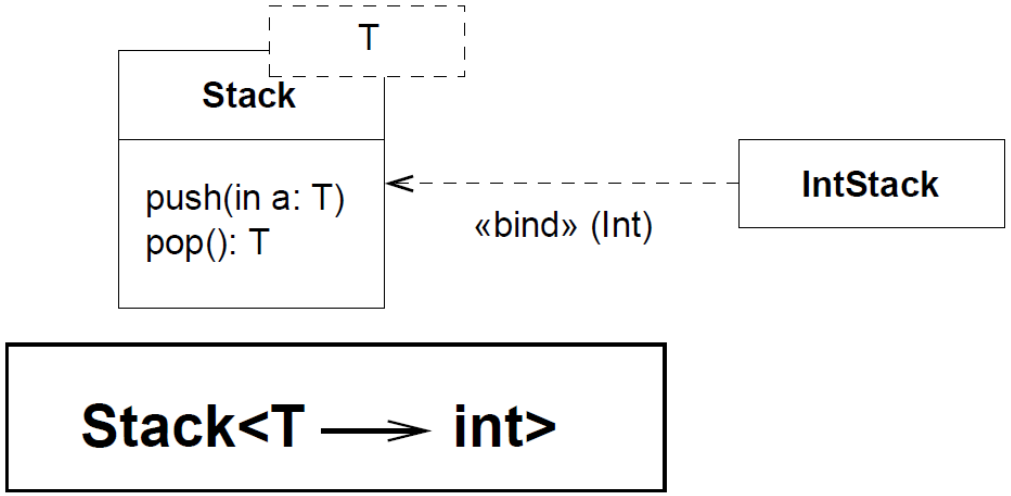
- I parametri possono essere tipi, valori numerici (oppure stringhe etc.), o operazioni.

- Gli elementi di modello che possono essere parametrizzati sono classi, package, componenti, operazioni, collaborazioni (gruppi di istanze interagenti) etc. (Quindi il concetto di genericità non si applica soltanto ai moduli).



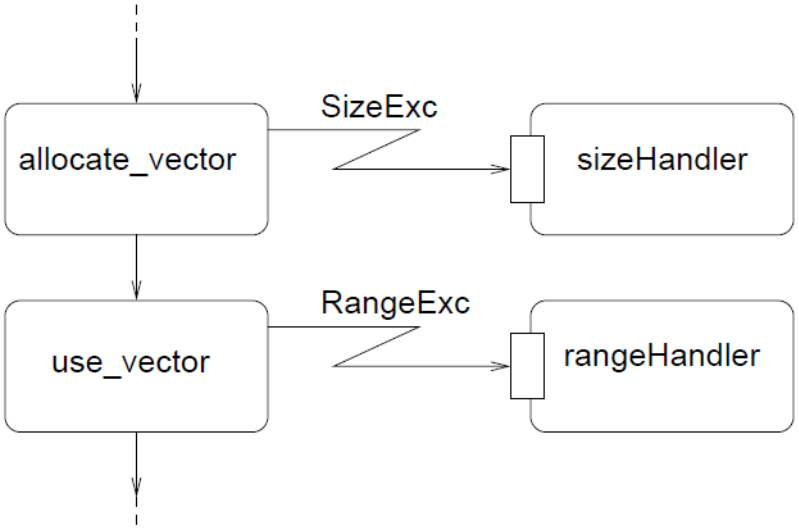
Moduli generici

- Da un modulo generico si ottiene un modulo specializzato assegnando valori ai parametri (binding). Il binding si può rappresentare i due modi:



Eccezioni

- In UML2 le eccezioni si modellano come oggetti che vengono creati quando l'esecuzione di un'azione incontra una situazione anomala e vengono passati come parametri d'ingresso ad un'altra azione, il gestore di eccezioni. Questo meccanismo viene modellato nel diagramma di attività.



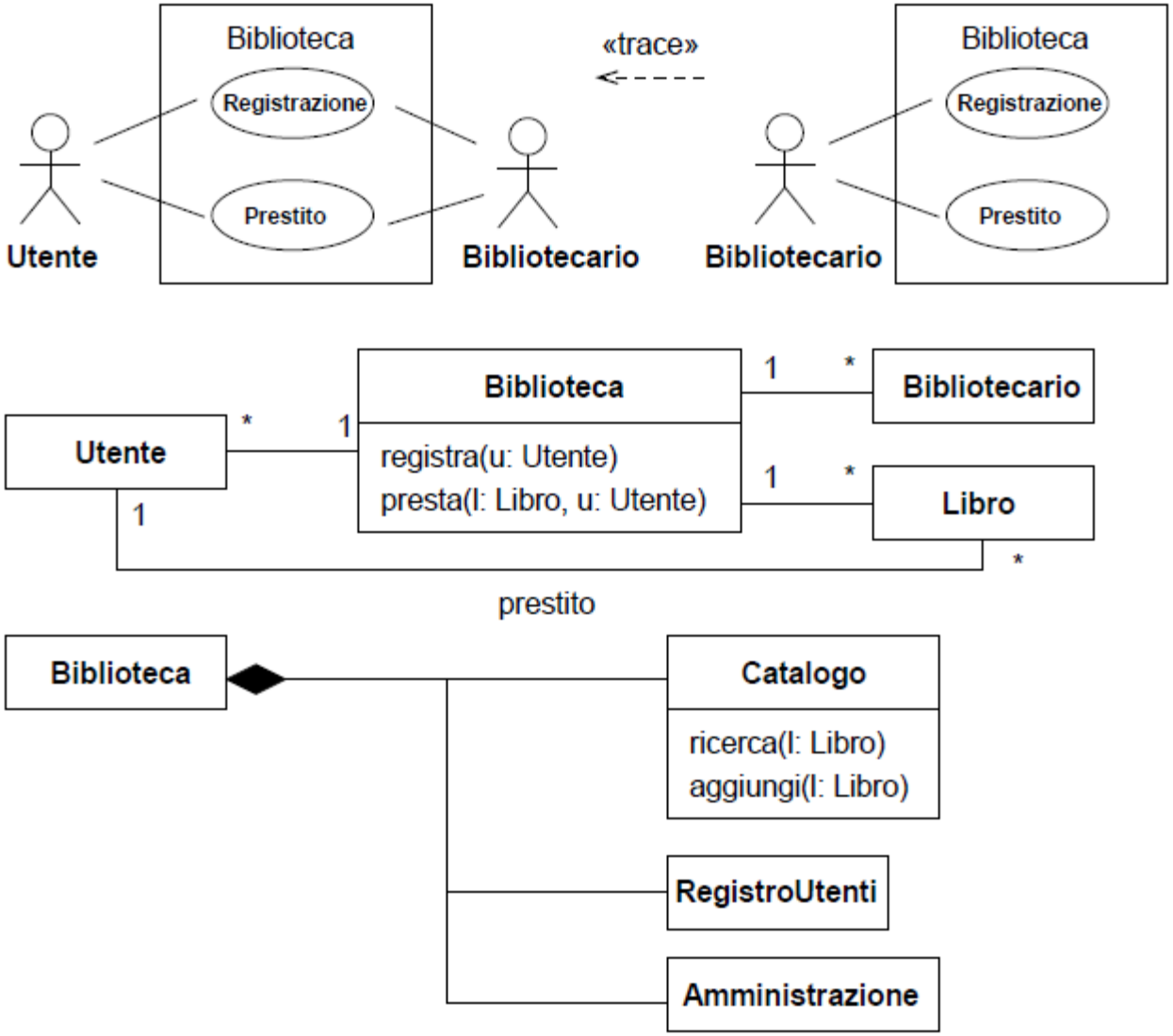
Modello di analisi

- Il modello di analisi è il punto di partenza per definire l'architettura software, la cui struttura, negli stadi iniziali della progettazione, ricalca quella del modello di analisi.

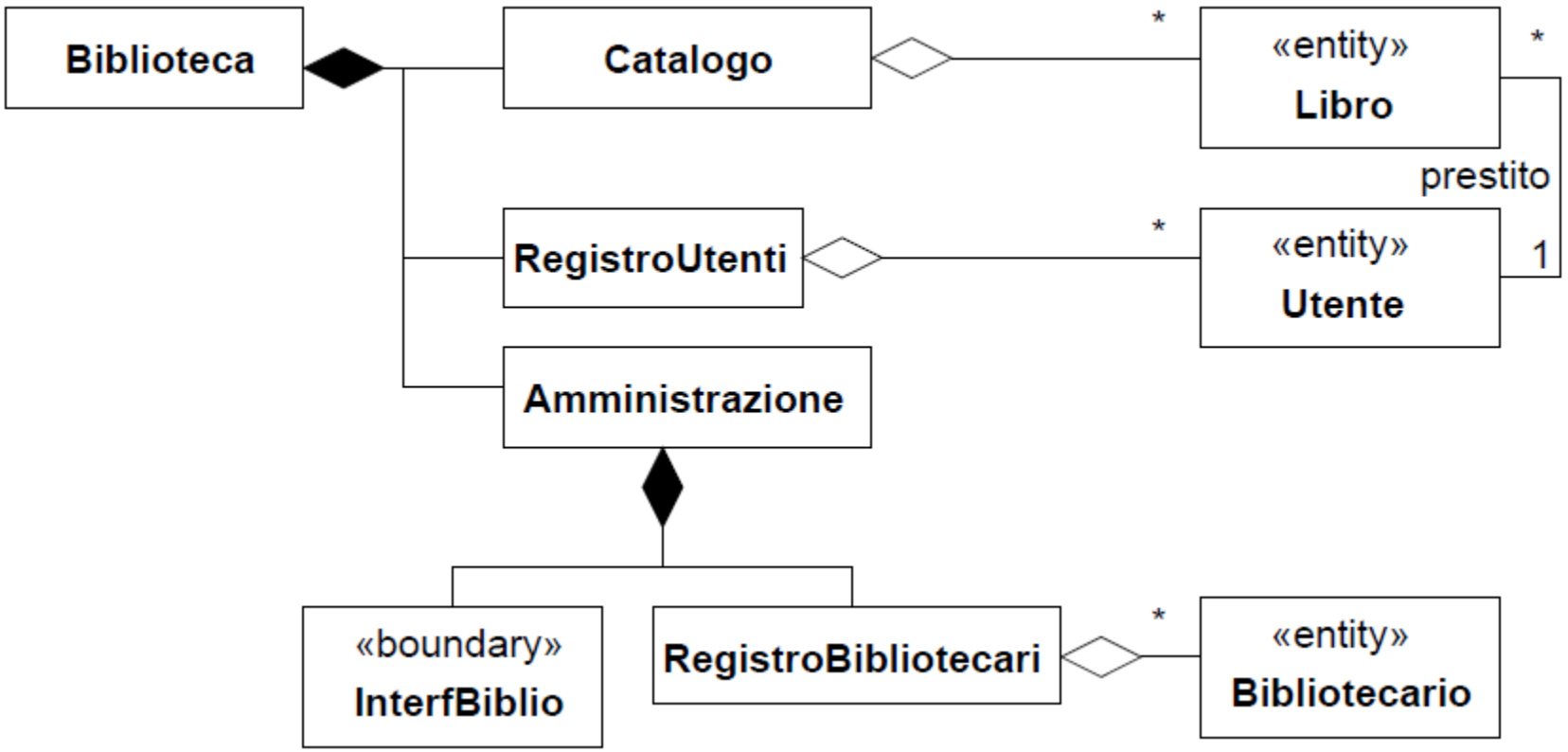
- La fase di progetto parte quindi dalle classi e relazioni definite in fase di analisi, a cui si aggiungono classi definite in fase di progetto di sistema e relative al dominio dell'implementazione.

- Nelle fasi successive del progetto questa struttura di base viene rielaborata, riorganizzando le classi e le associazioni, espandendo le classi in moduli complessi, introducendo nuove classi, e definendone le interfacce e gli aspetti piú importanti delle implementazioni.

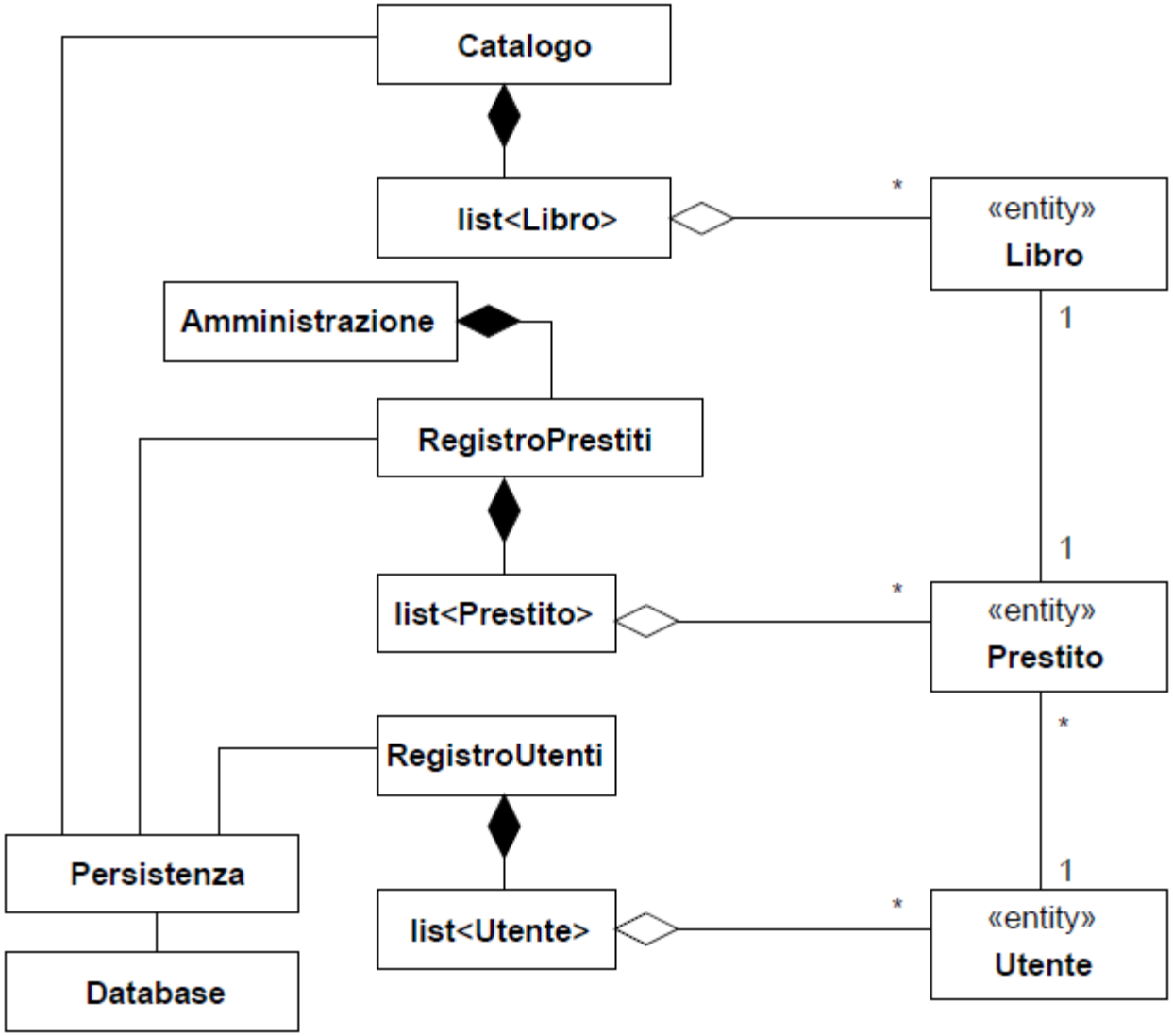
UML - Modello di analisi



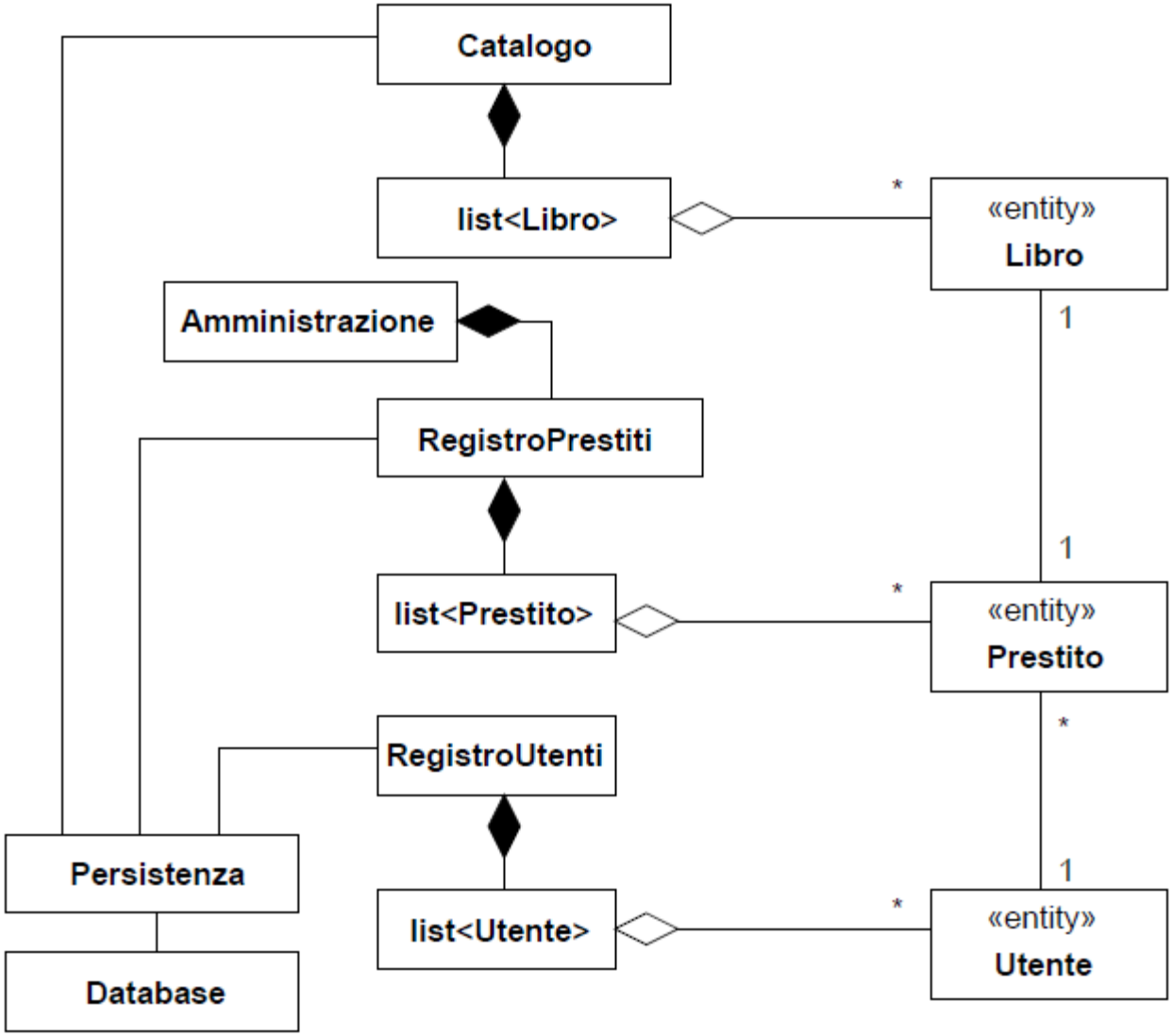
UML - Modello parziale di progetto



UML - Modello parziale di progetto



UML - Modello parziale di progetto



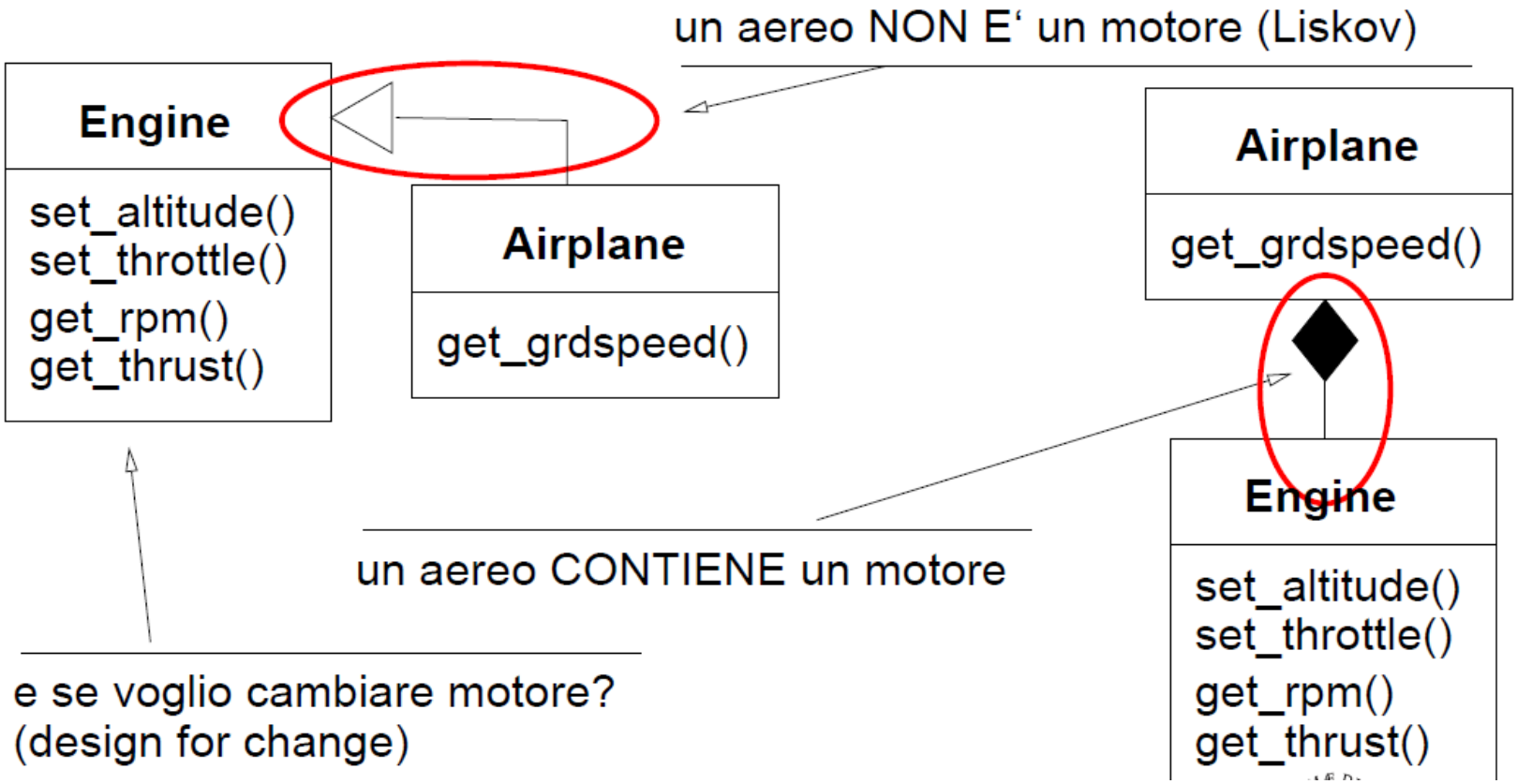
UML - Eredità e polimorfismo

Nei linguaggi di programmazione, l'eredità è il meccanismo che inserisce gli attributi e operazioni di una classe base nelle classi derivate. Questo meccanismo si può sfruttare per tre scopi:

- riusare moduli preesistenti (usare con cautela, v. lucido successivo);
- riprodurre nell'architettura software le relazioni di generalizzazione presenti nel dominio dell'applicazione (p.es., classi Person e Student);
- implementare le relazioni di realizzazione.

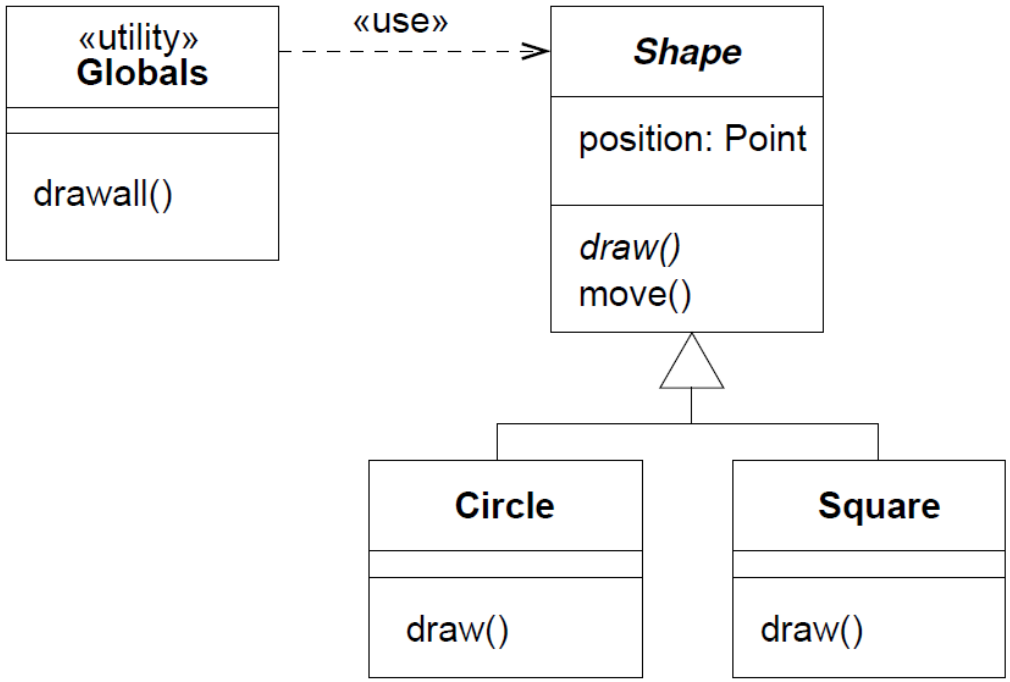
```
class Person {
    char* name;
    char* birthdate;
public:
    char* getName();
    char* getBirthdate();
};
```

```
class Student : public Person {
    char* student_number;
public:
    char* getStudentNumber();
};
```


un aereo NON E' un motore (Liskov)
Engine
set_altitude()
set_throttle()
get_rpm()
get_thrust()
Airplane
get_grdspeed()
Airplane
get_grdspeed()
Engine
set_altitude()
set_throttle()
get_rpm()
get_thrust()
un aereo CONTIENE un motore
e se voglio cambiare motore?
(design for change)

UML - Eredità e polimorfismo

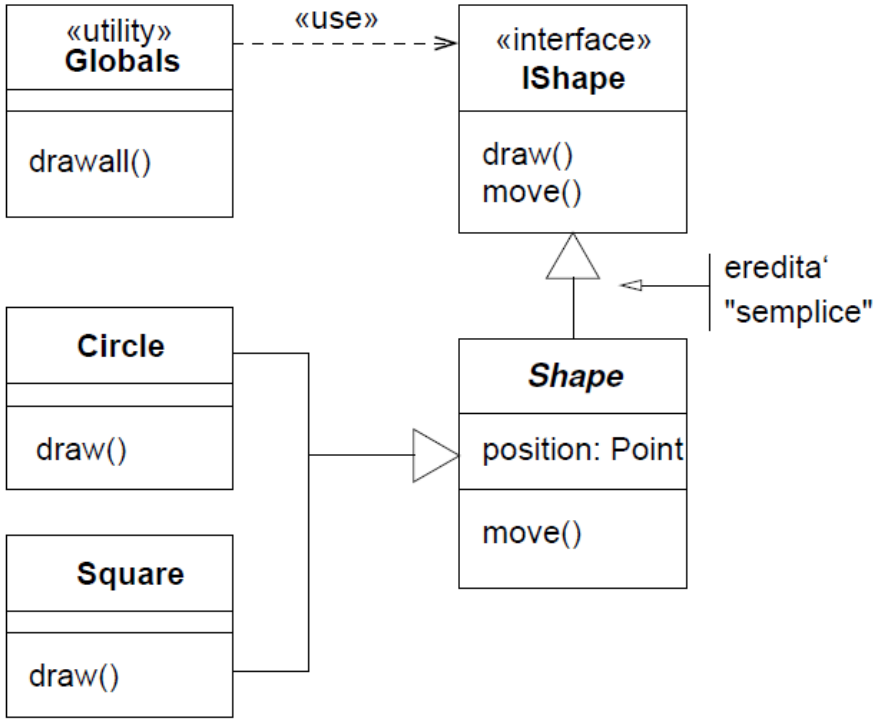
- Il polimorfismo è la possibilità che un riferimento (per esempio un identificatore o un puntatore) denoti oggetti o funzioni di tipo diverso.
- Nei linguaggi OO, il polimorfismo si basa sui meccanismi dell'eredità e del binding dinamico (operazioni virtuali in C++).
- Il polimorfismo è una tecnica fondamentale nel progetto OO.



UML - Eredità e polimorfismo

```
class Shape {
    Point position;
public:
  virtual void draw() = 0;      // operazione astratta
  virtual void move(Point p)    // operazione concreta
      { position = p; };
  //...
};

class Square : public Shape {
  //...
public:
  void draw()
    { /* impl. */ };
};
```

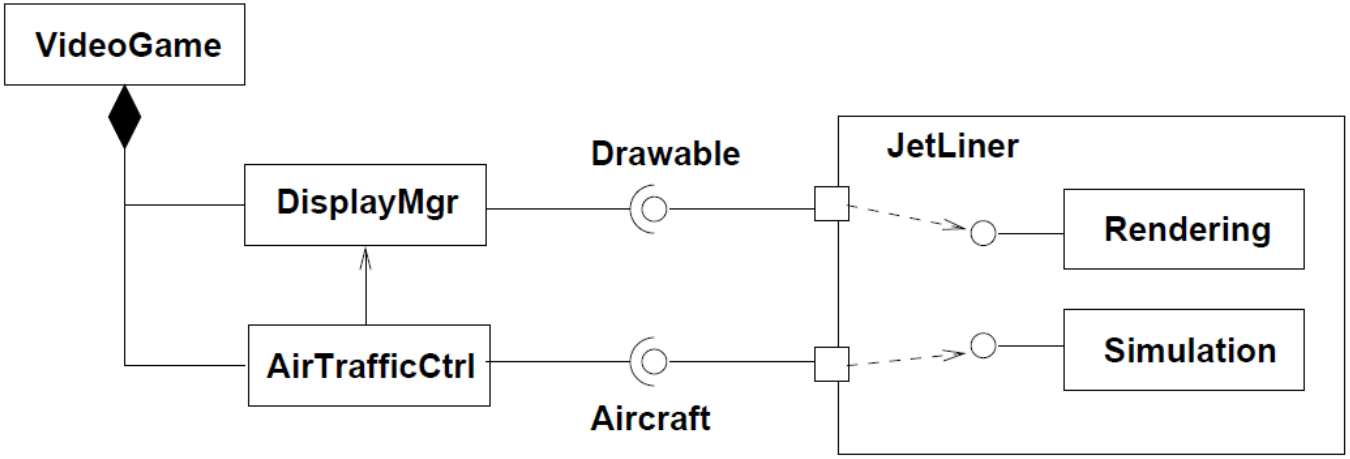
```
class Circle : public Shape {
  //...
public:
  void draw()
    { /* impl. */ };
};
```




```
/* disegna tutte le figure contenute nell'array shps */
void drawall(Shape** shps)   // puntatori alla classe base
{
    for (int i = 0; i < 2; i++)
        shps[i]->draw();     // operazione POLIMORFICA
}

main()
{
    Shape* shapes[2];
    shapes[0] = new Circle;  // puntatore a classe derivata
    shapes[1] = new Square;  // ALTRA classe derivata
    drawall(shapes);
}
```

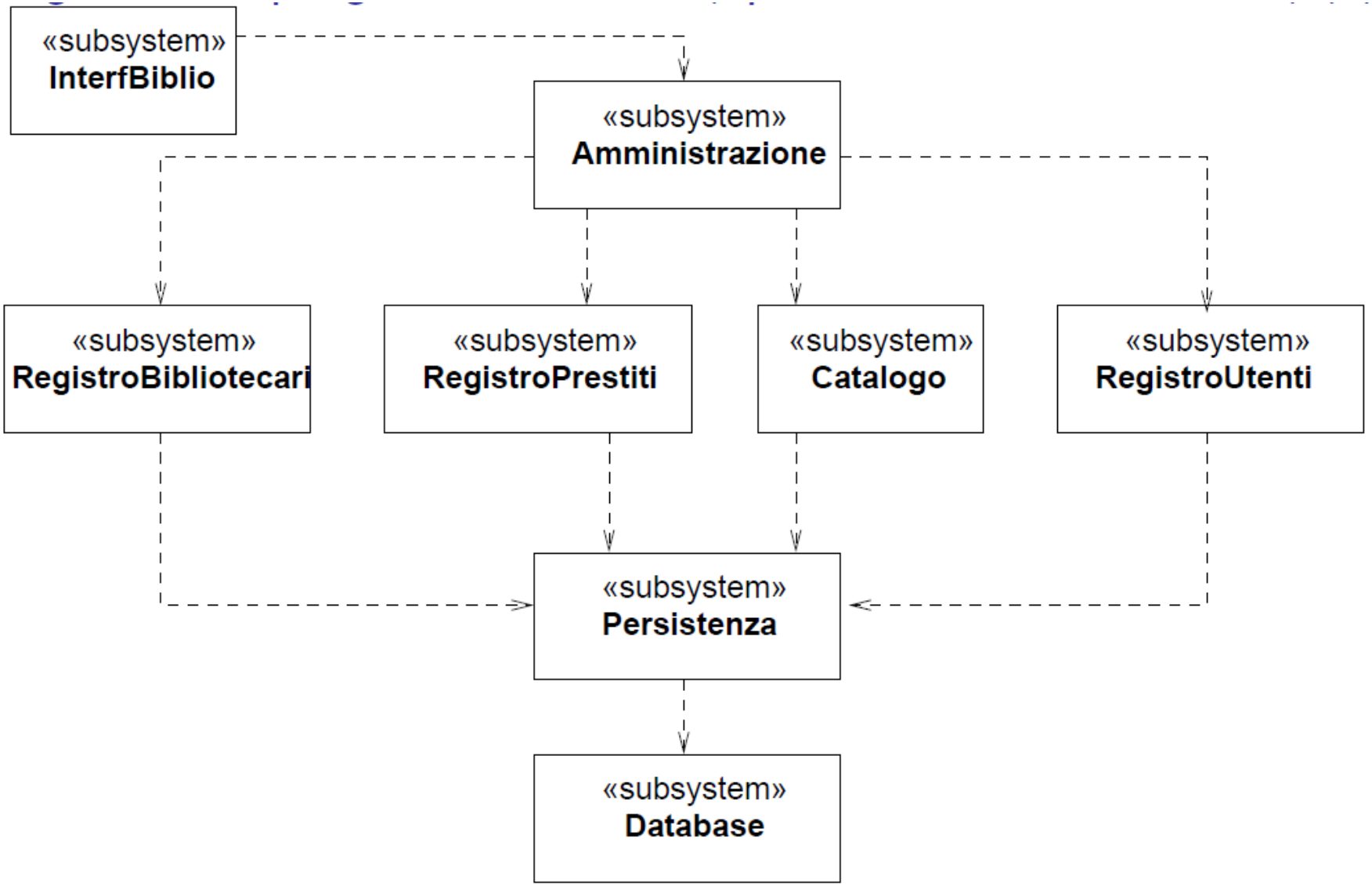
rappresentazione in C++ dell'elemento ≪interface≫

```
class IShape {  // classe virtuale pura
public:
    virtual void draw() = 0;
    virtual void move(Point p) = 0;
};

class Shape : public IShape { // classe astratta
    Point position;
public:
    virtual void draw() = 0;
    virtual void move(Point p) { position = p; };
};

class Square : public Shape {
  //...
public:
  void draw()
    { /* impl. */ };
};

class Circle : public Shape {
  //...
public:
  void draw()
    { /* impl. */ };
};

void drawall(IShape** shps)
 {
   // metodo immutato ANCHE SE AGGIUNGO NUOVE CLASSI DERIVATE
 }

 main()
 {
   IShape* shapes[2];
   // algoritmo immutato ANCHE SE AGGIUNGO NUOVE CLASSI DERIVATE
```

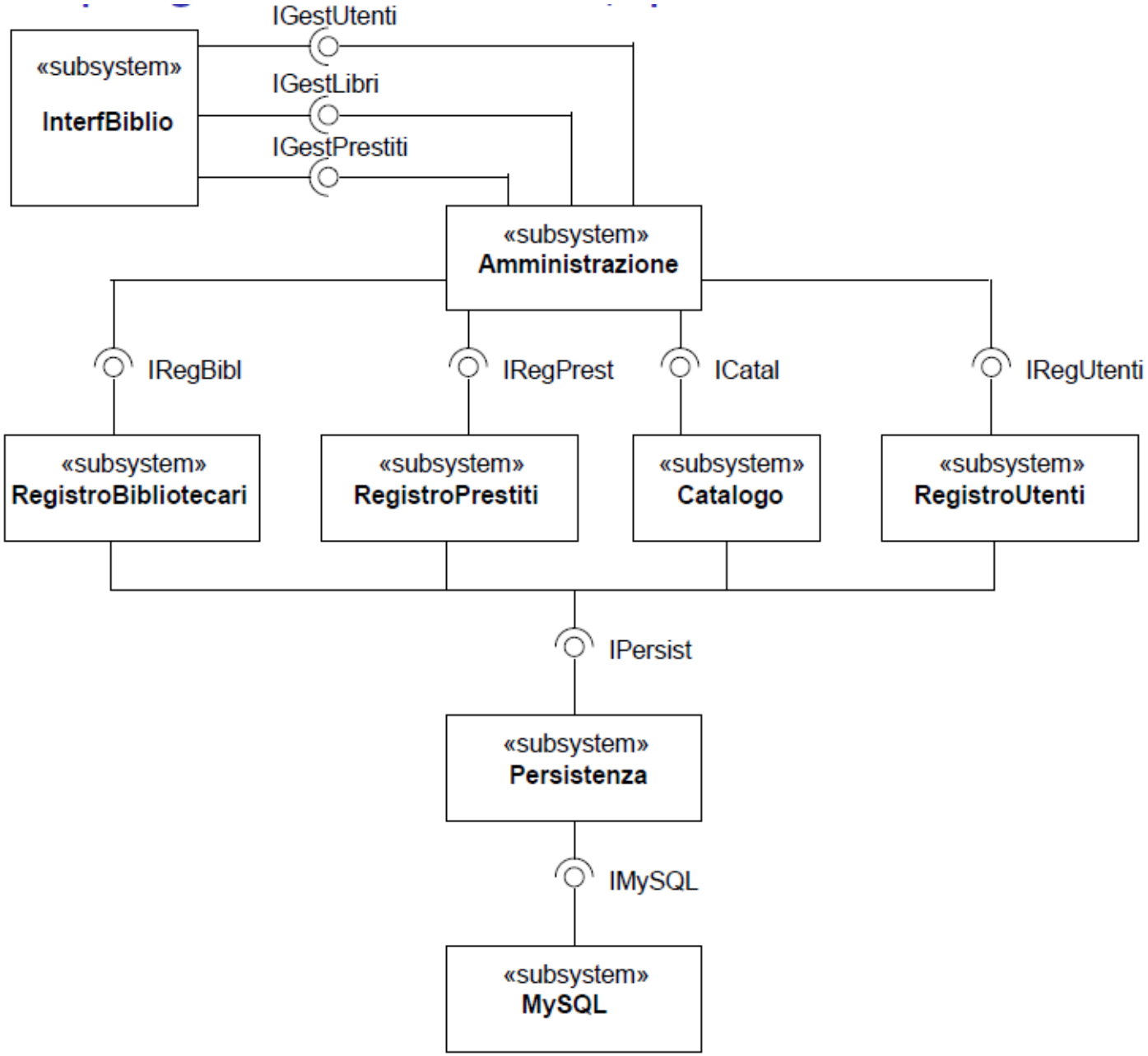


ogni componente di tipo JetLiner offre le stesse interfacce, ma le può realizzare per diversi tipi di aereo.

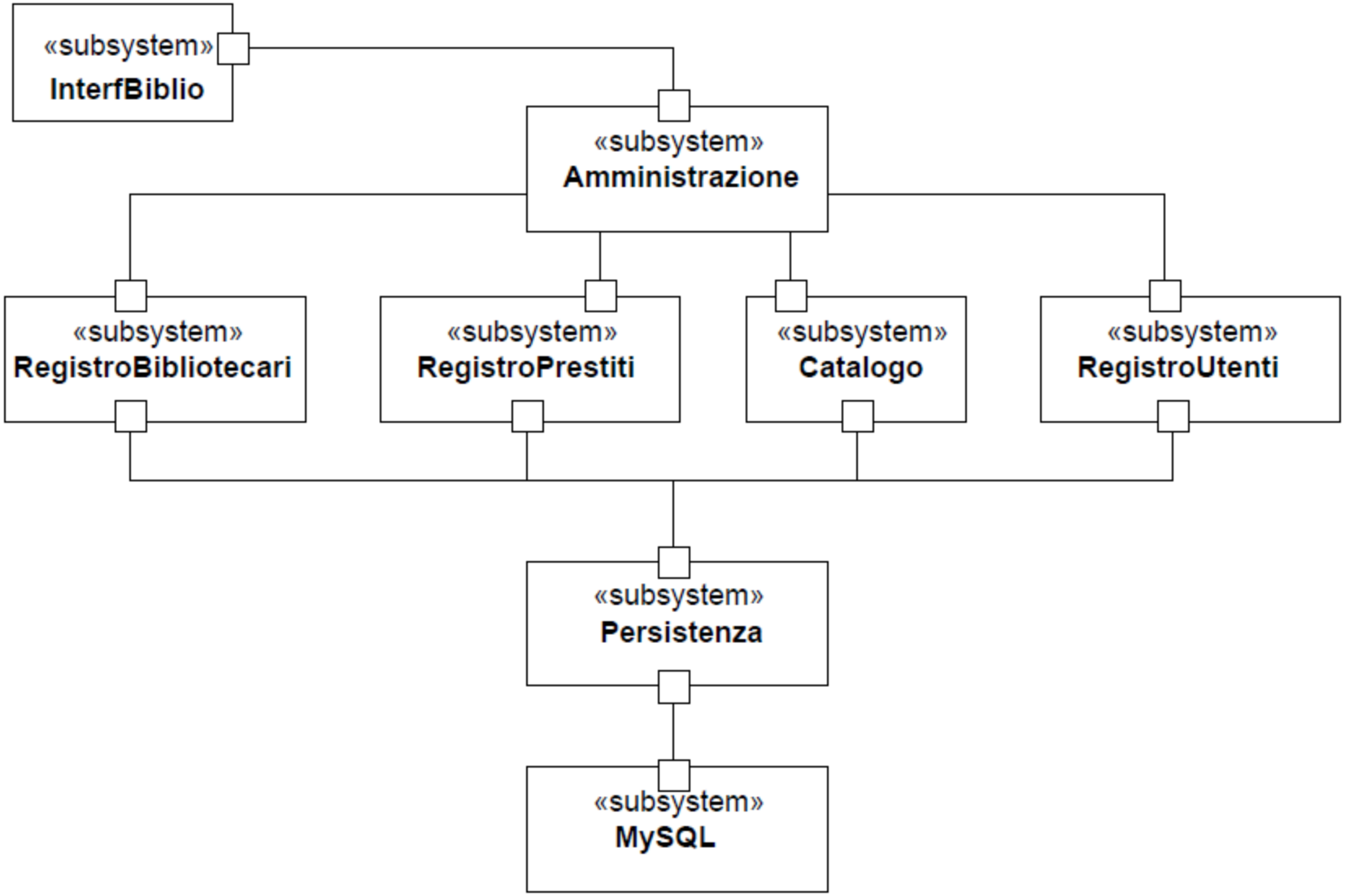
UML - Ripartizione in sottosistemi



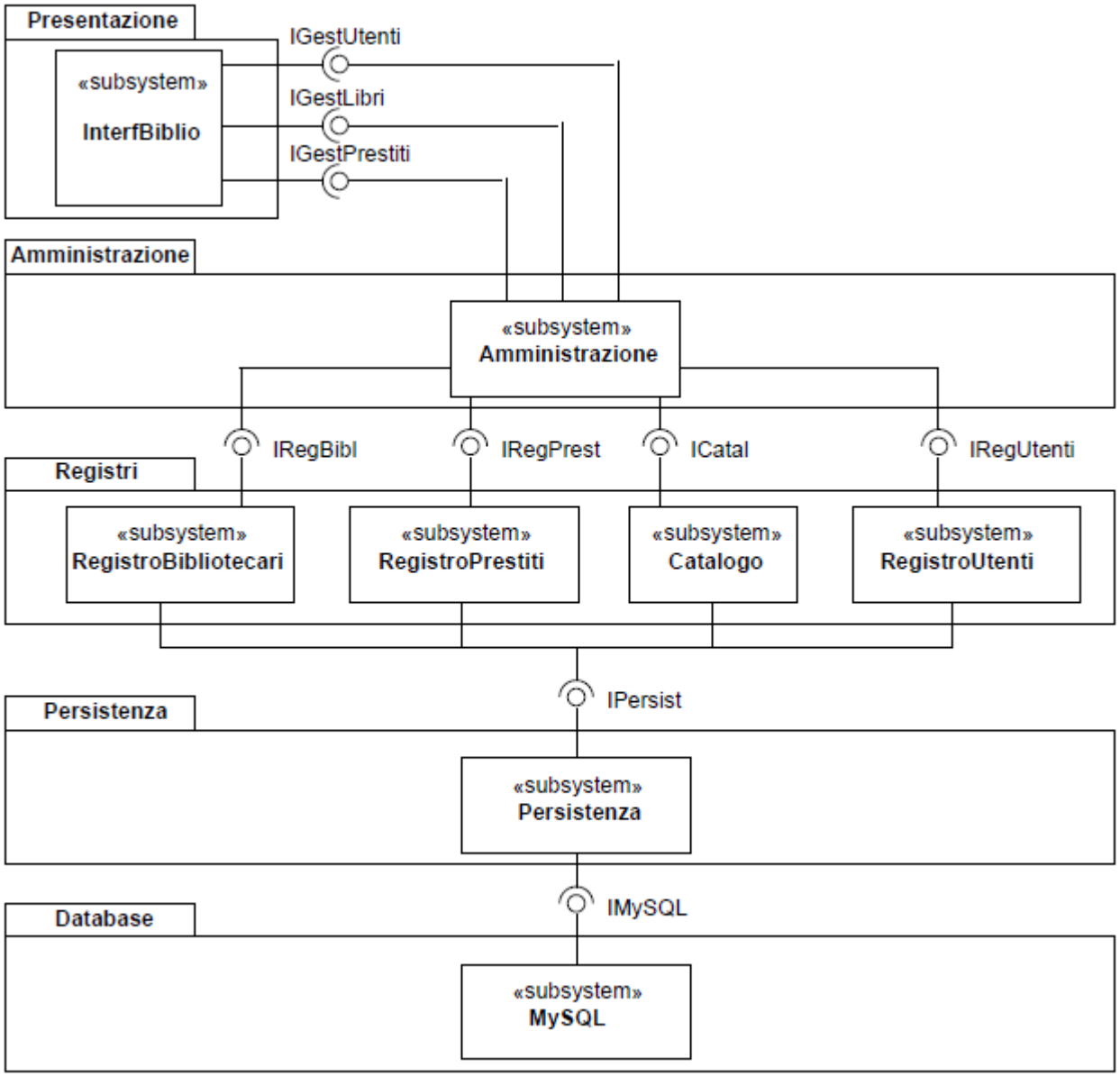
UML - Ripartizione in sottosistemi



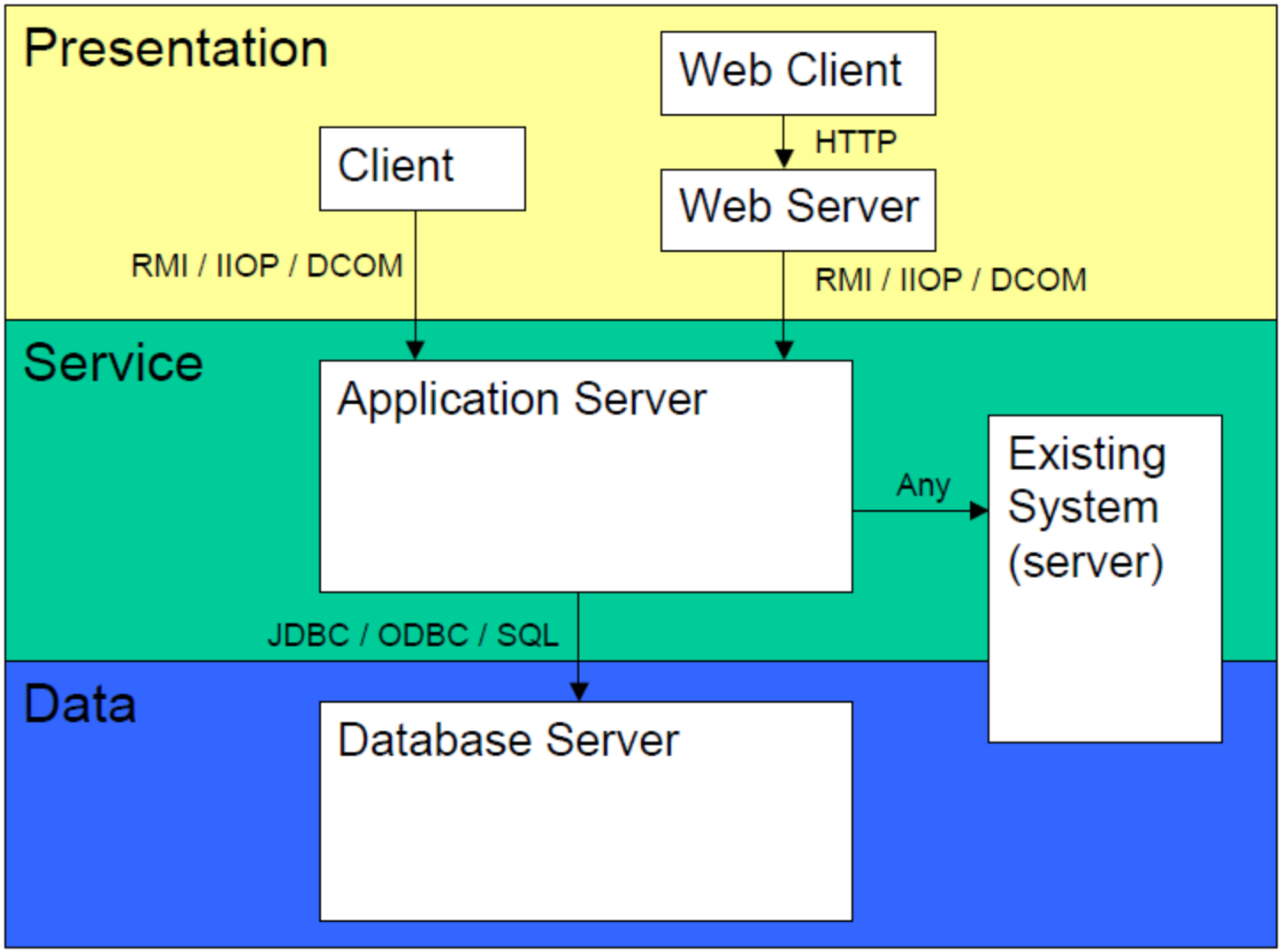
UML - Ripartizione in sottosistemi



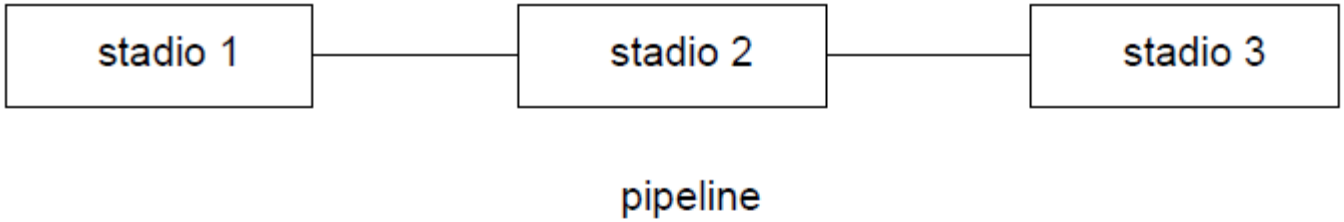
UML - Strati e partizioni



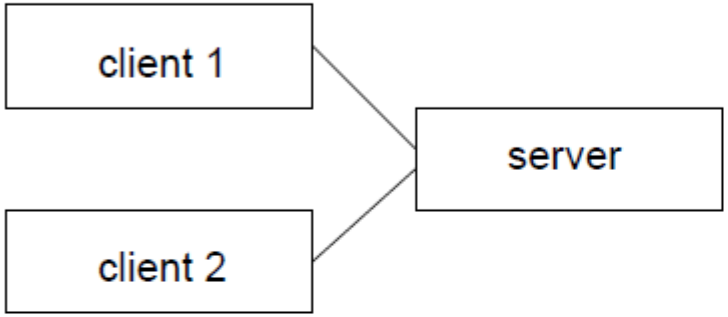
UML - Strati e partizioni



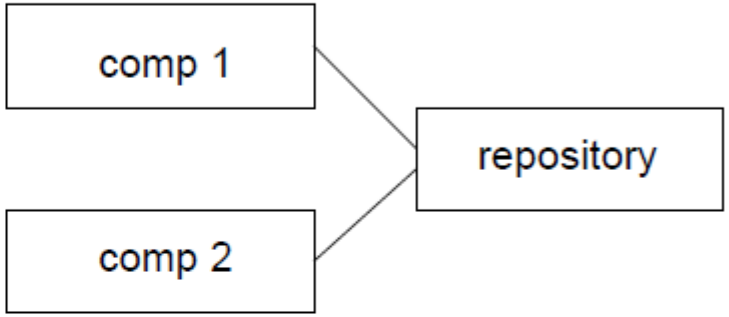
UML - Architetture standard



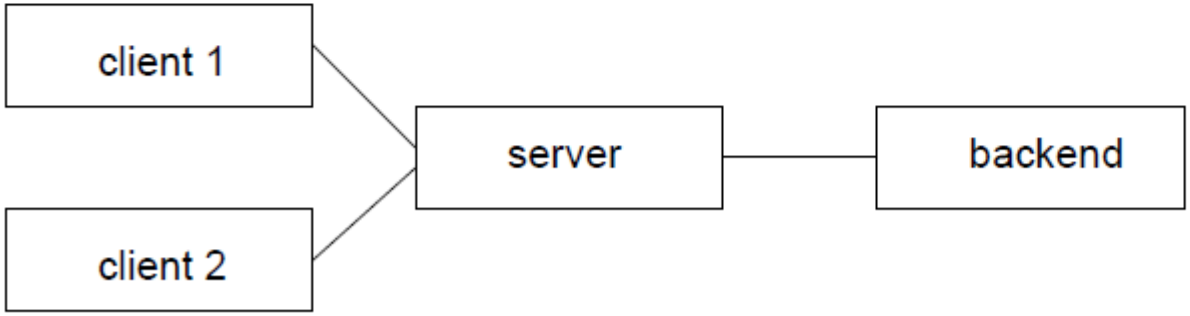
pipeline



client−server a 1 livello



repository



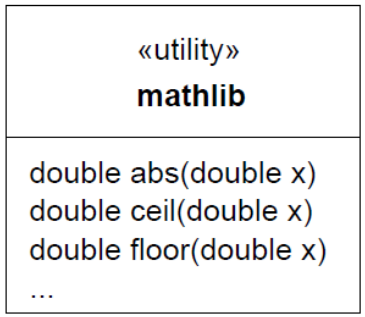
client−server a 2 livell1 (two−tier)

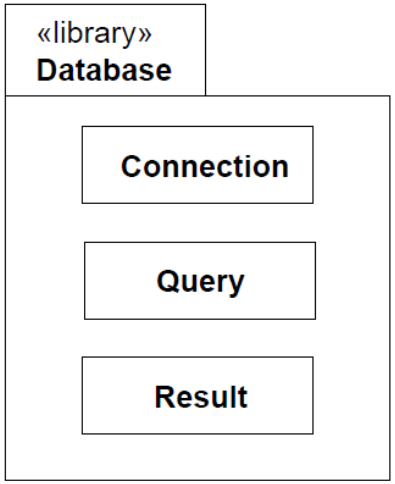
UML - Librerie

- Una libreria logica è una raccolta di componenti che offrono servizî ad un livello di astrazione piuttosto basso.

- Le librerie si usano “dal basso verso l’alto”, assemblando componenti semplici e predefiniti per ottenere strutture complesse specializzate.

- Considerando, p.es., una tipica libreria matematica, possiamo usare le sue funzioni per costruire un sw che risolva un particolare problema di equazioni differenziali, oppure un sw che risolva un problema di geometria.



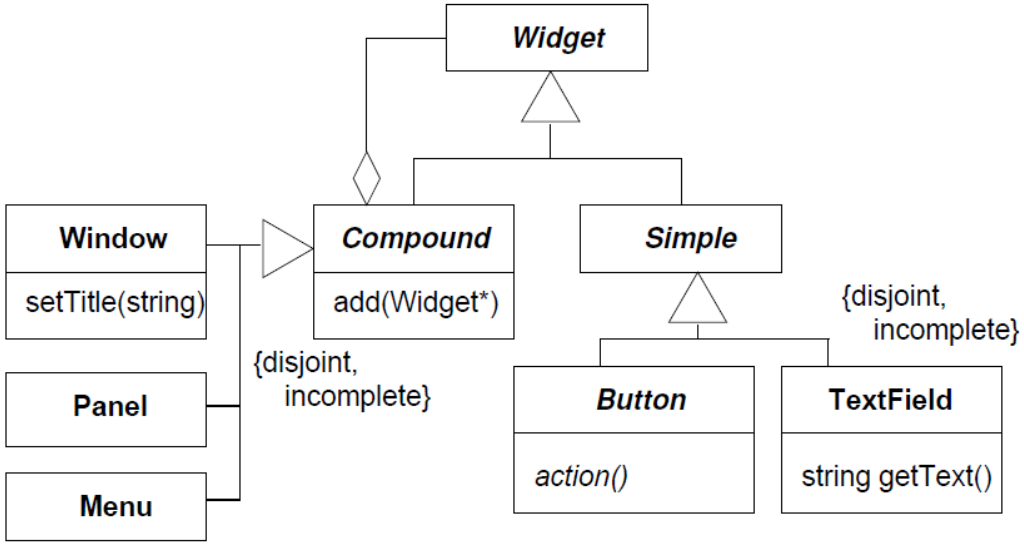


- Un framework contiene invece dei componenti ad alto livello di astrazione che offrono uno schema di soluzione preconfezionato per un determinato tipo di problema.

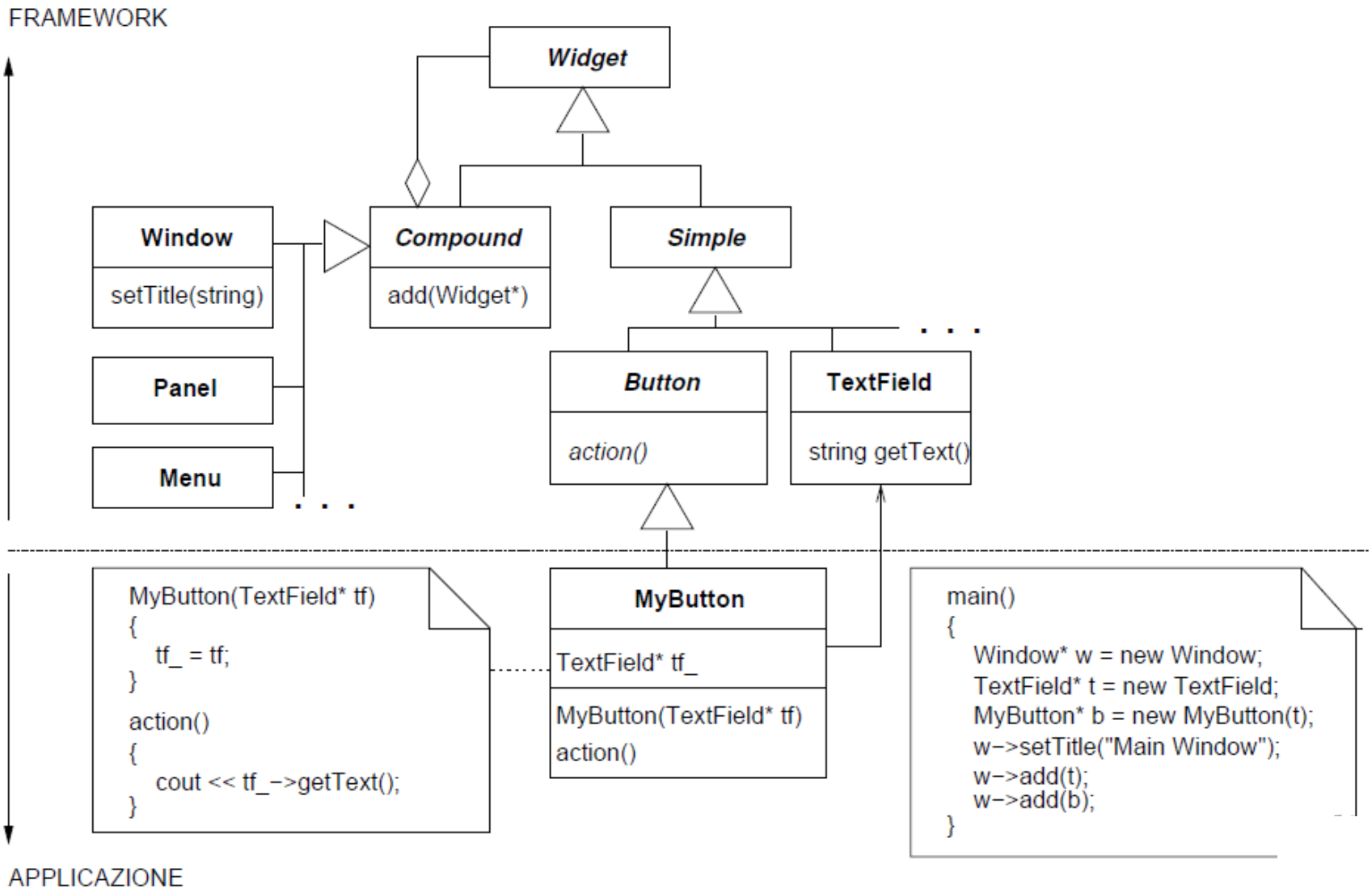
- I framework si usano “dall’alto verso il basso”, riempiendo delle strutture complesse predefinite (delle “intelaiature”) con dei componenti semplici specializzati.

- Per esempio, un framework per risolvere generici problemi di equzioni differenziali può essere applicato ad un problema particolare, aggiungendovi moduli specifici sviluppati ad hoc. Similmente si può specializzare un framework per problemi geometrici.

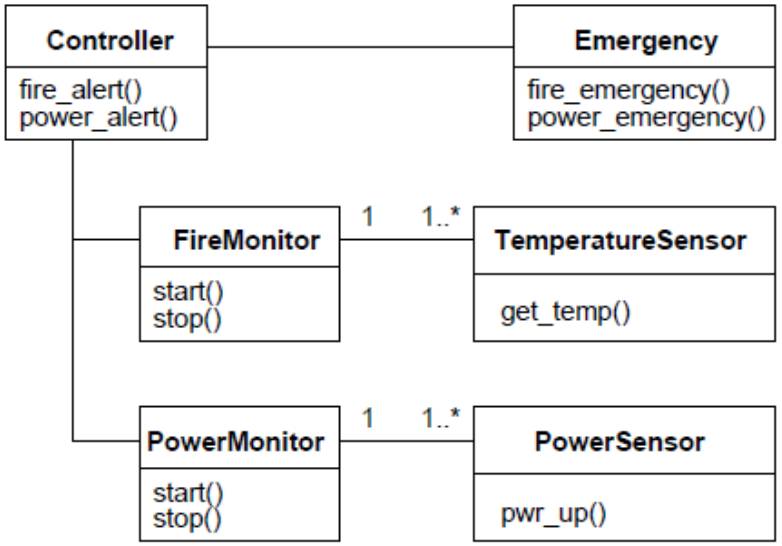
Esempio: framework per costruire interfacce grafiche

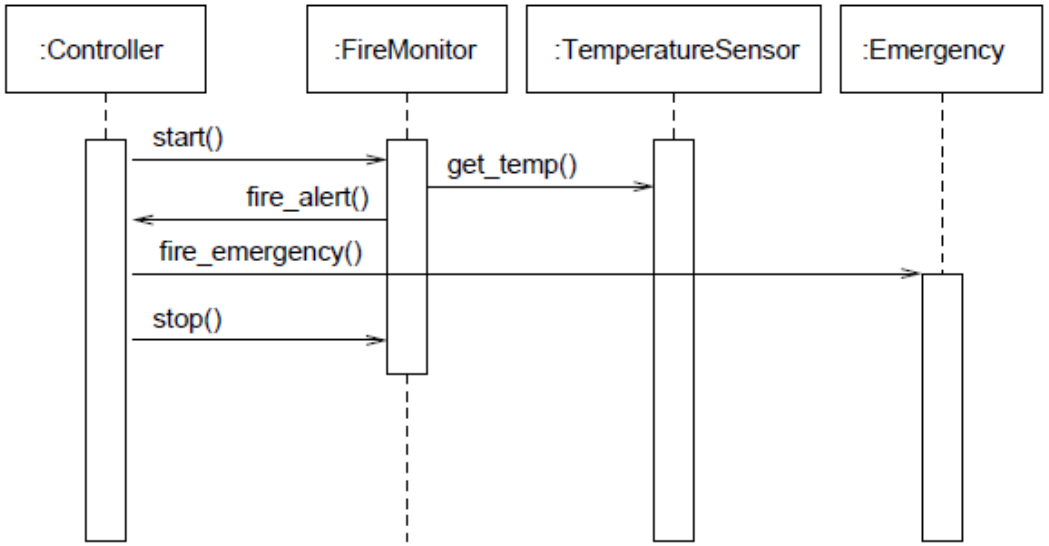


UML - Framework



Consideriamo un modello iniziale (di analisi o di progetto) che non tenga conto della concorrenza.

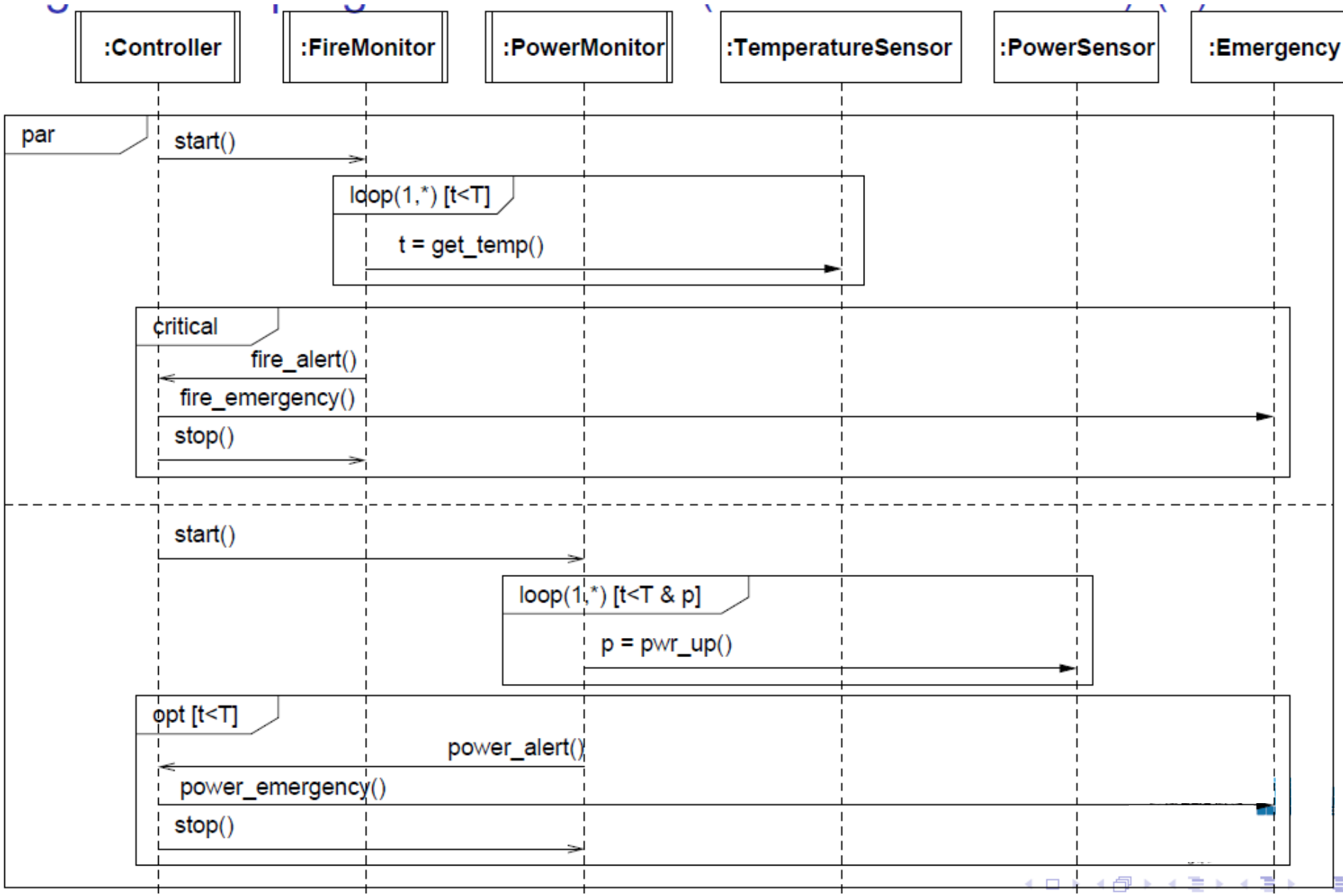




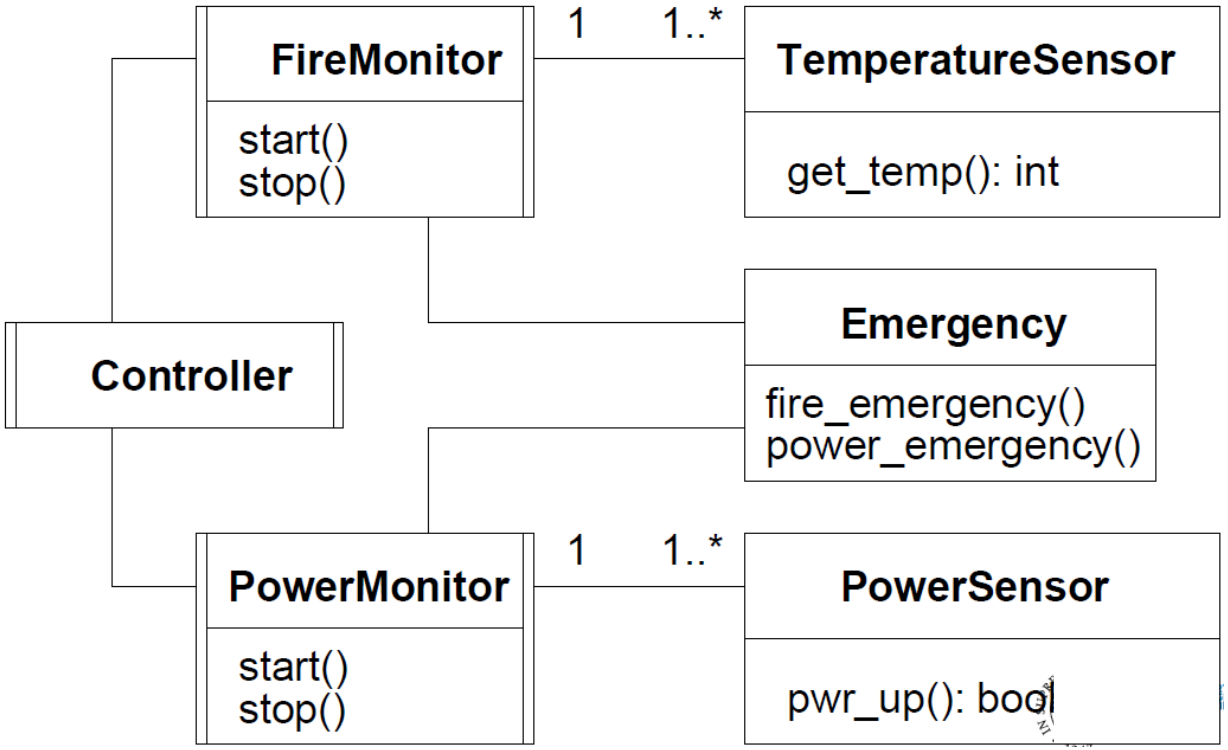
Rappresentazione di flussi di controllo nei diagrammi di sequenza

- In un diagramma di sequenza, un **frammento** è una sequenza d'interazioni che può essere ripetuta o eseguite condizionalmente, eseguita in parallelo ad altre sequenze, etc.;
- ogni frammento è delimitato da un rettangolo etichettato da una **parola chiave**;
- frammento **par**: sequenze parallele, in regioni distinte del frammento;
- frammento **loop**: sequenza ripetuta un certo numero (anche indeterminato) di volte e finché vale la condizione specificata;
- frammento **critical**: sequenza non interrompibile;
- frammento **opt**: sequenza eseguita solo se vale la condizione specificata.

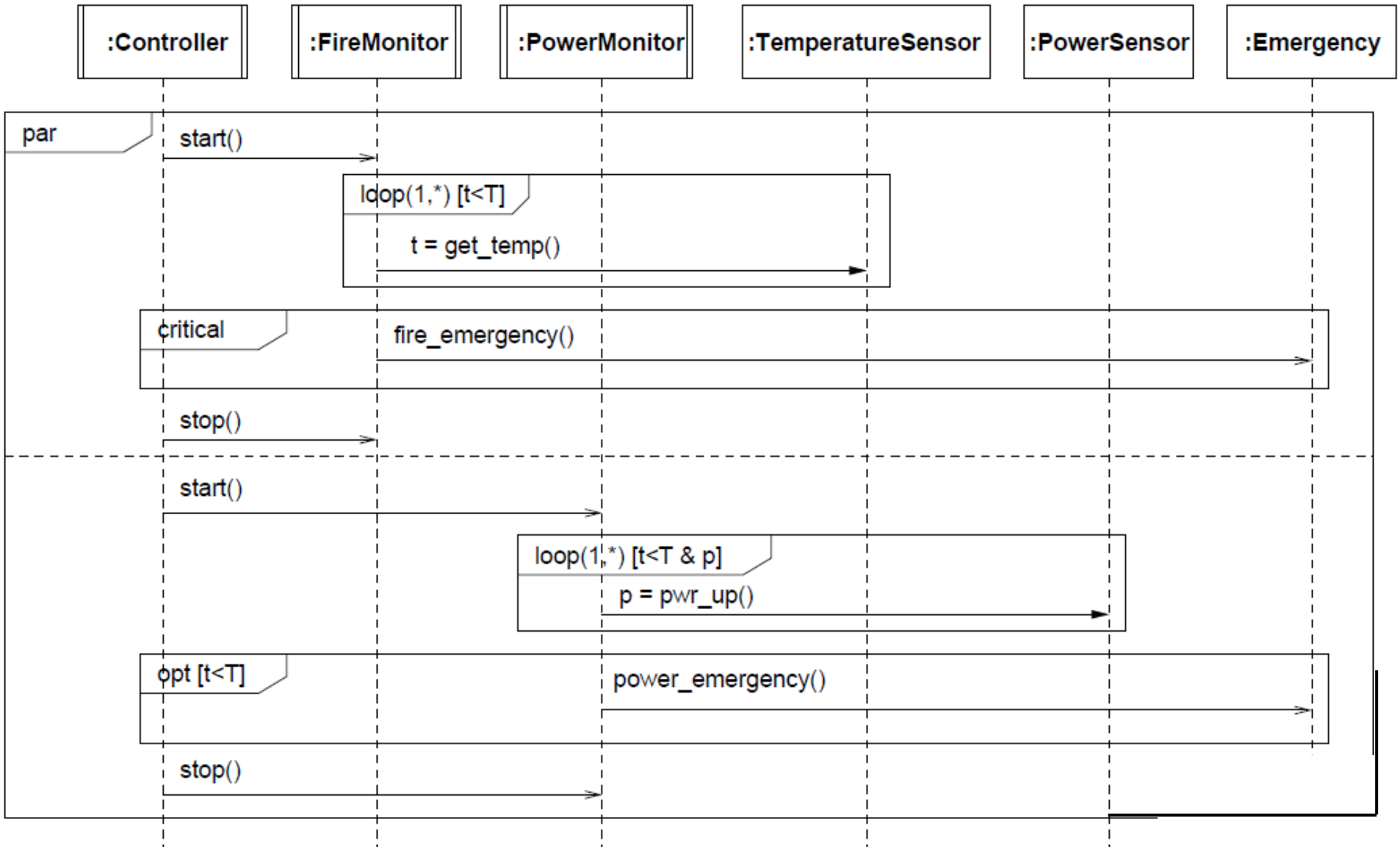
UML – Sistemi concorrenti



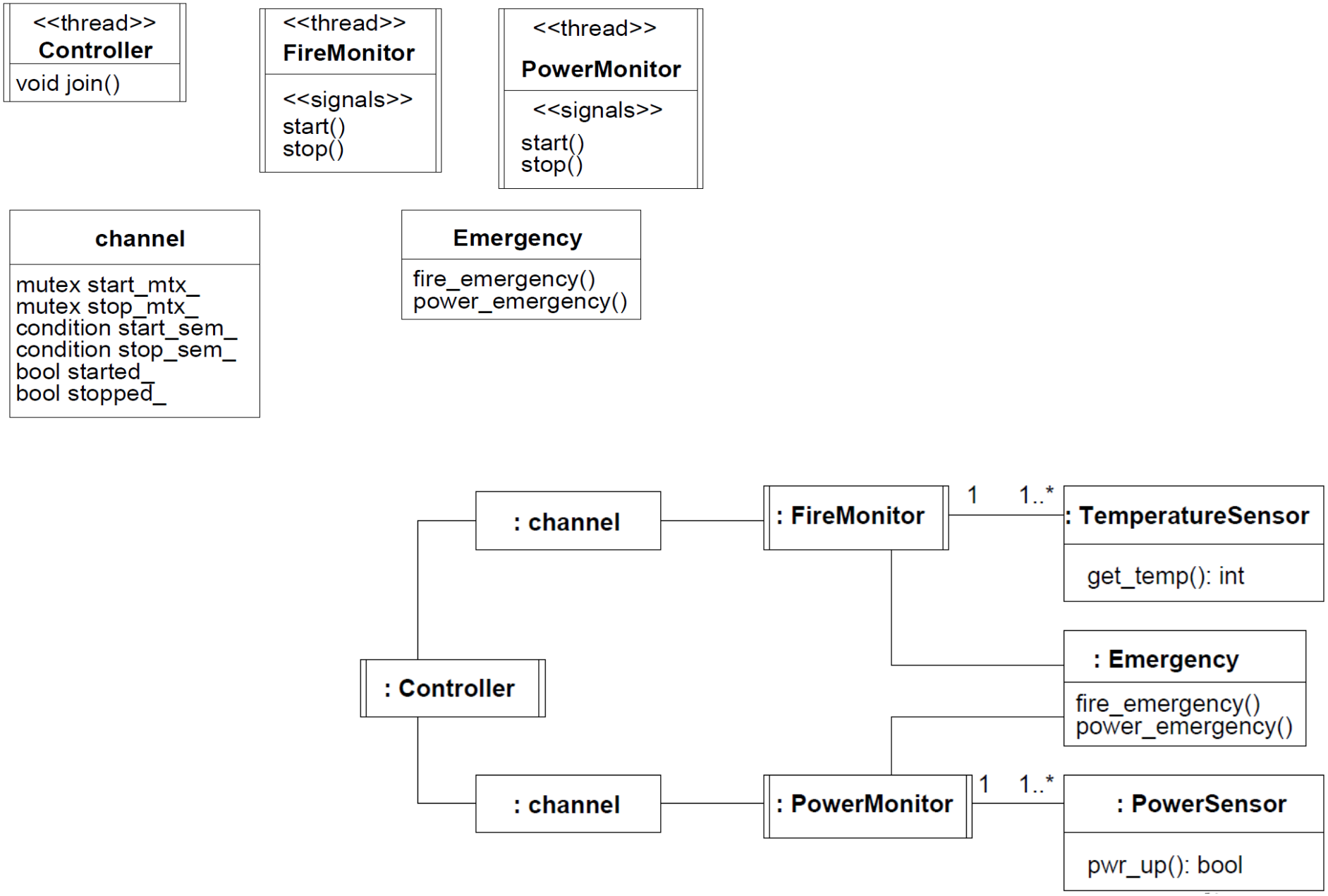
- Nell'architettura alternativa mostrata di séguito, vediamo che il sottosistema Emergency è interessato dai flussi di controllo dei due monitor (cioè, ciascuno dei due monitor può invocare operazioni di Emergency) ed è quindi condiviso.
- Nel progetto del sistema bisogna tener conto della condivisione di risorse per evitare problemi come il blocco (o deadlock) o l'inconsistenza delle informazioni causata da accessi concorrenti a dati condivisi.
- In questo caso particolare, bisogna rispettare il vincolo costituito dalla priorità dell'attività fire_emergency() rispetto all'altra.

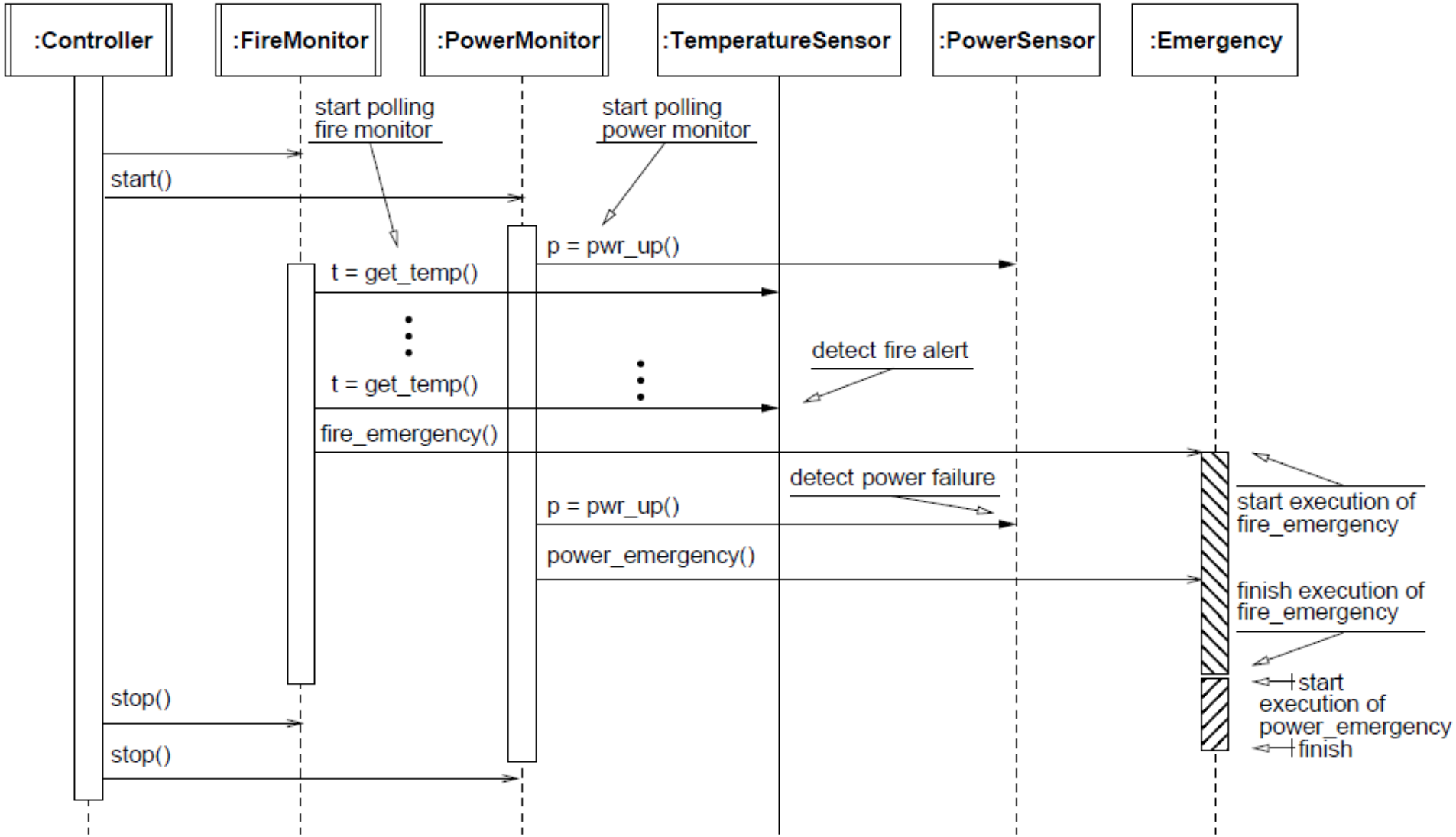


UML – Sistemi concorrenti



UML – Sistemi concorrenti




:Controller
:FireMonitor
:PowerMonitor
:TemperatureSensor
:PowerSensor
:Emergency
start polling fire monitor
start polling power monitor
start()
p = pwr_up()
t = get_temp()
detect fire alert
t = get_temp()
fire_emergency()
detect power failure
p = pwr_up()
power_emergency()
start execution of fire_emergency
finish execution of fire_emergency
start execution of power_emergency
finish
stop()
stop()

Progetto delle classi

- Se necessario, scomporre classi complesse in classi piú semplici;
- specificare completamente attributi ed operazioni già presenti, indicando visibilità, modificabilità, tipo e direzione dei parametri;
- aggiungere operazioni implicite nel modello, per esempio costruttori e distruttori;
- aggiungere operazioni ausiliarie, se necessario.

- Completezza
- sufficienza
- primitività
- coesività
- disaccoppiamento.

- Evitare ottimizzazioni inutili (usare strumenti di profilazione).

Progetto delle associazioni

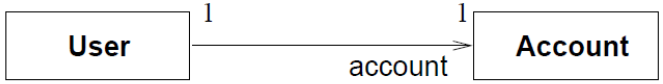
- navigabilità
  - le istanze di A hanno un puntatore verso istanze di B?
  - le istanze di B hanno un puntatore verso istanze di A?

- molteplicità
  - quante istanze di B sono associate ad una istanza di A, e viceversa?

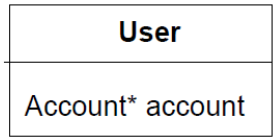
- ordinamento
  - le istanze di B associate ad una istanza di A sono un insieme (non ci sono ripetizioni) o un multiinsieme?
  - se sono un insieme, è ordinato?

Progetto delle associazioni

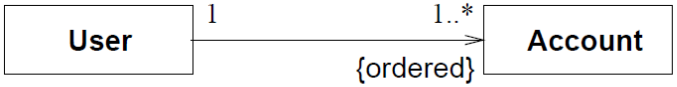
- Uno a uno
  - un'associazione uno a uno si implementa con un puntatore se navigabile in un solo verso, o due (uno per classe) altrimenti.



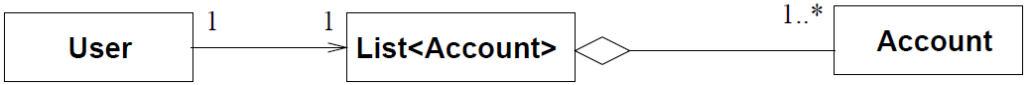
uno a uno



- Uno a molti
  - Un'associazione uno a molti si implementa con una classe contenitore, scelta in base alle caratteristiche dell'insieme di istanze associate (in particolare l'ordinamento) ed ai modi di accesso previsti.



uno a molti

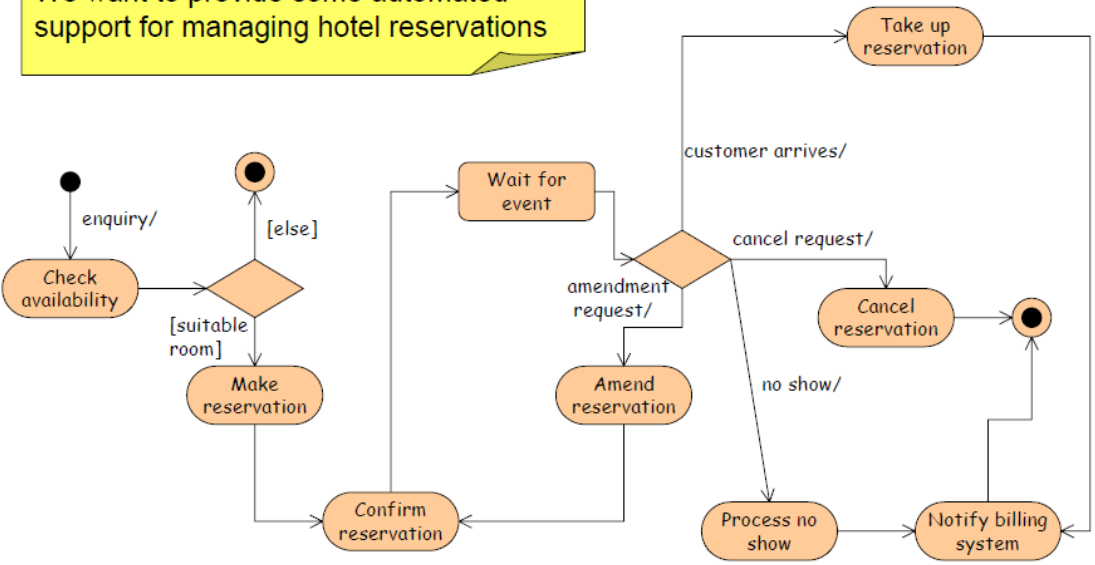


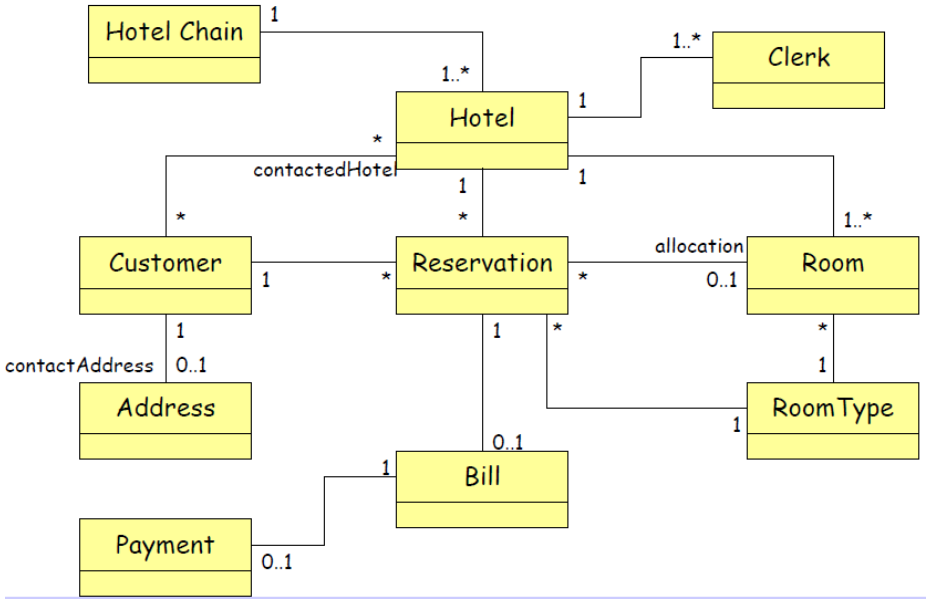
- Molti a molti
  - Un'associazione molti a molti si implementa con una classe intermedia che scomponga l'associazione originale in una associazione da uno a molti ed una da molti a uno.

molti a molti

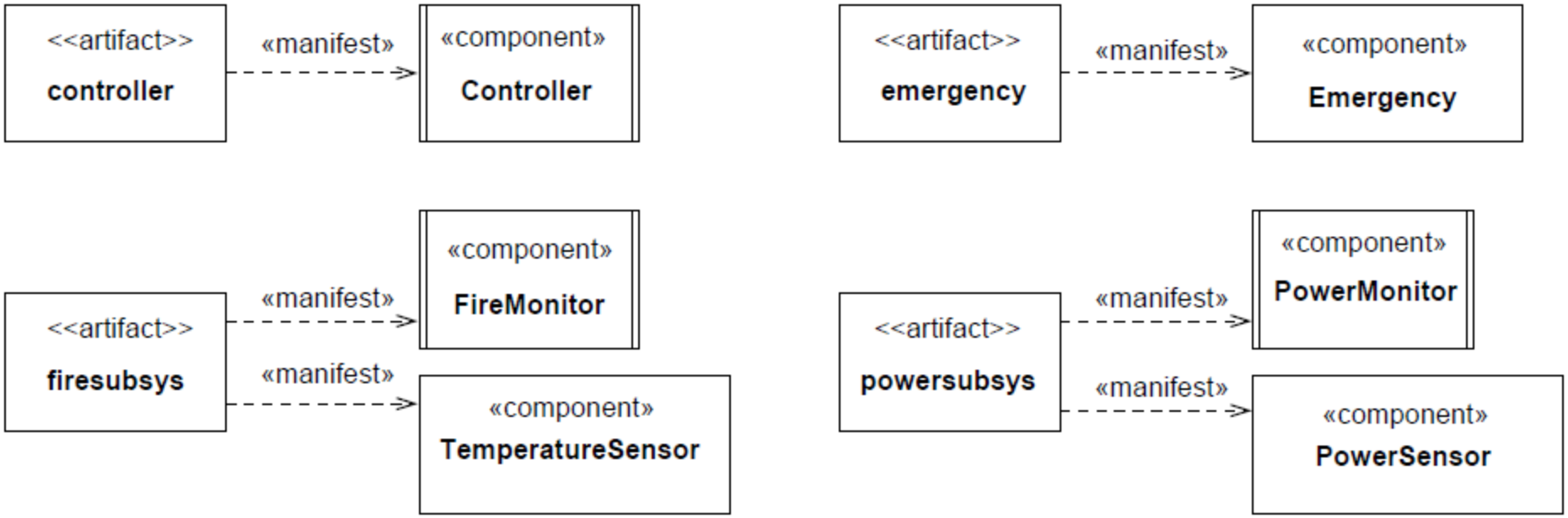
Processo e modello concettuale di business

We want to provide some automated support for managing hotel reservations

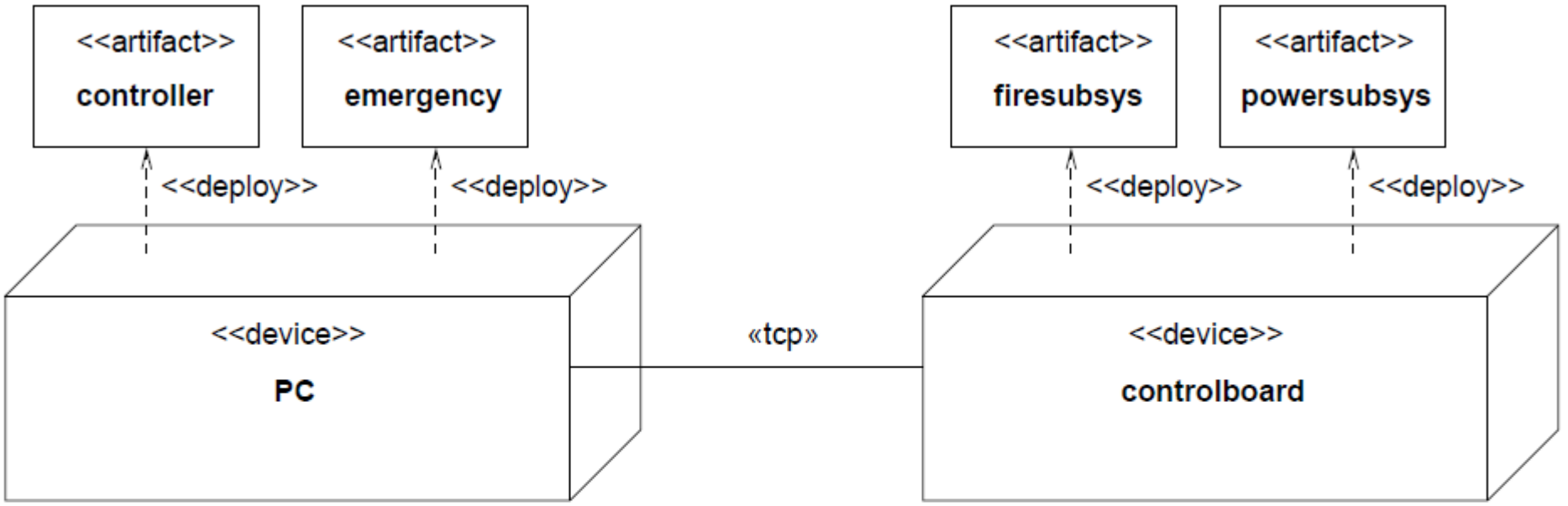




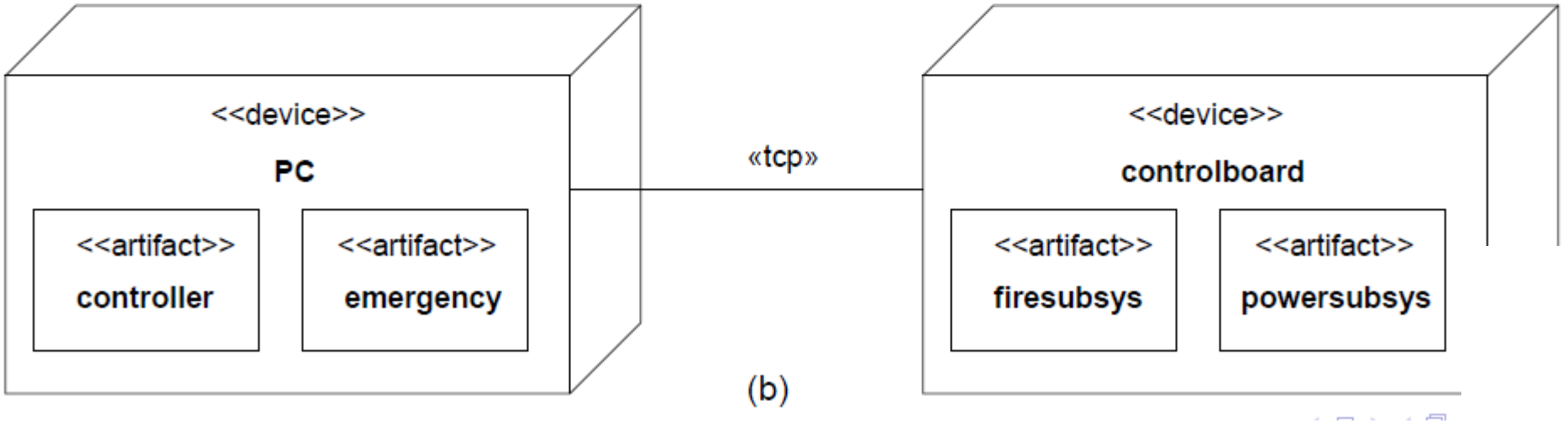
Relazione fra moduli e artefatti (manifest)



Relazione fra moduli e artefatti (deploy)



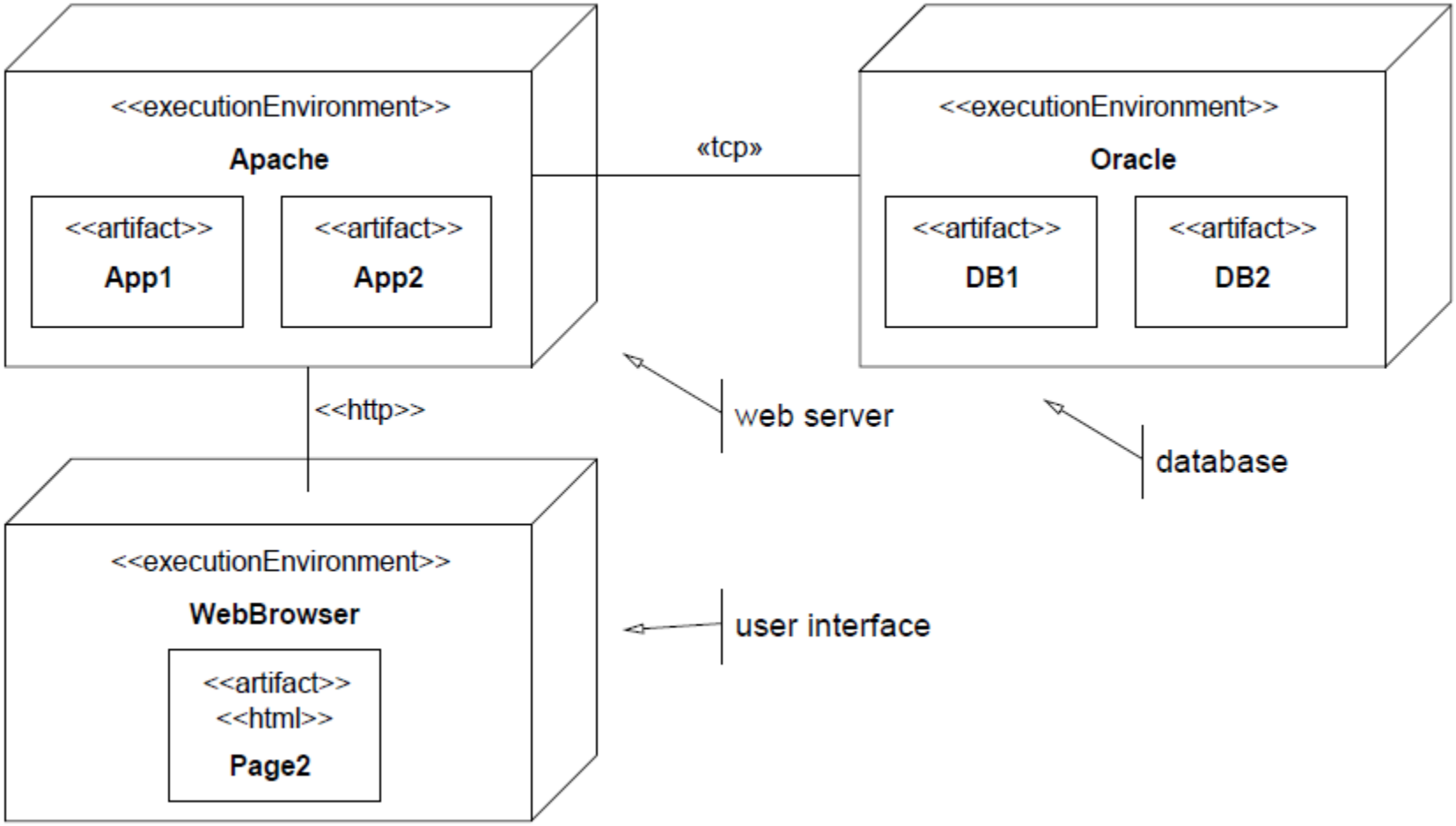
(a)

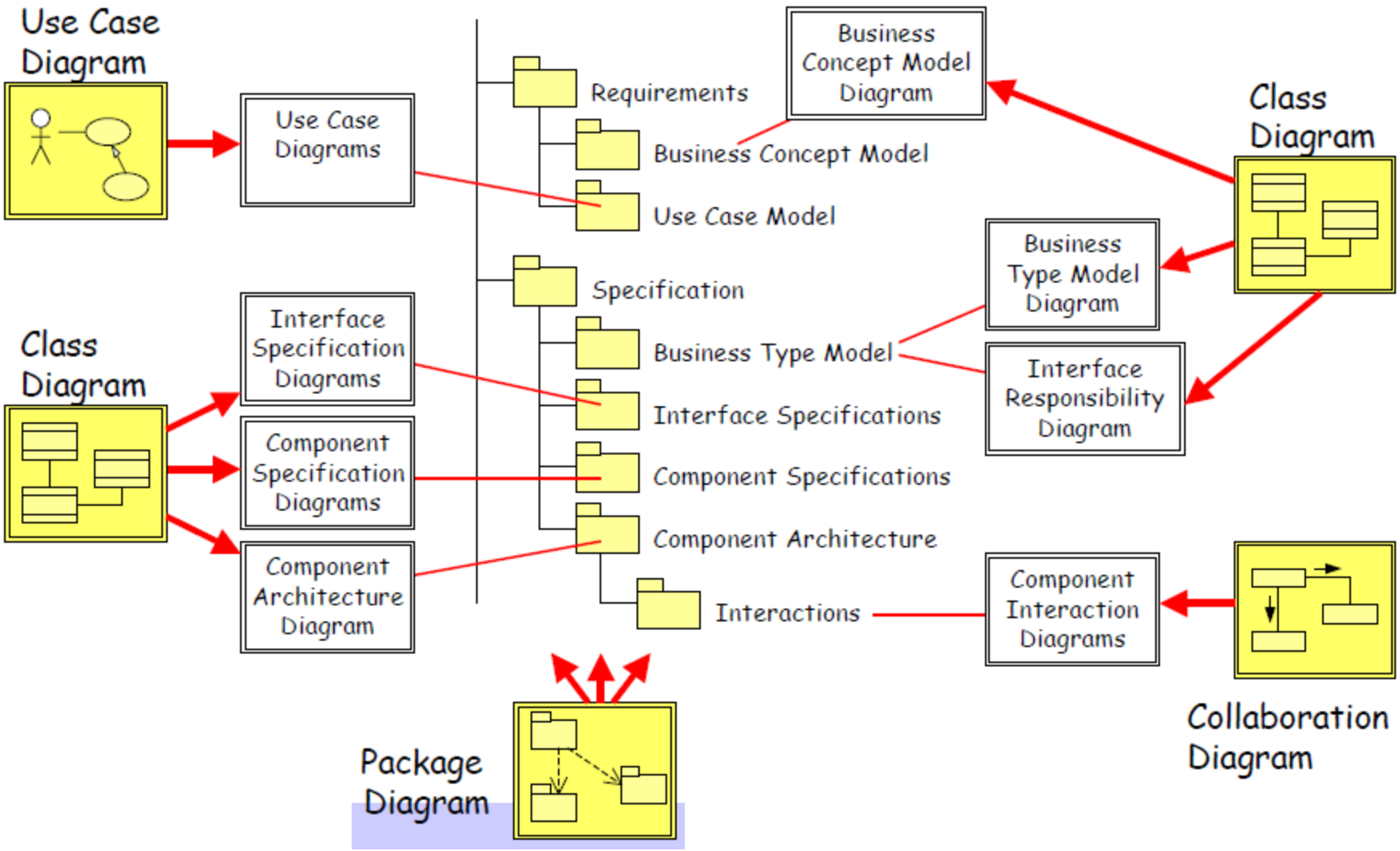


(b)

Ambienti di esecuzione

Sistemi software, esterni all'applicazione sviluppata, entro cui viene eseguita l'applicazione.




Use Case Diagram
Use Case Diagrams
Requirements
Business Concept Model Diagram
Business Concept Model
Use Case Model
Class Diagram
Business Type Model Diagram
Interface Responsibility Diagram
Specification
Business Type Model
Interface Specifications
Component Specifications
Component Architecture
Interactions
Class Diagram
Interface Specification Diagrams
Component Specification Diagrams
Component Architecture Diagram
Component Interaction Diagrams
Collaboration Diagram
Package Diagram

	Problem domain	S/W spec	Implementation
Use case		boundary interactions	
Class diagram	information models	component structures	component structures
Seq/collab diagram		required object interactions	designed object interactions
Activity diagram	business processes		algorithms
Statechart		object lifecycles	object lifecycles

Punti di vista del software

- **Funzionale**: descrive gli elementi funzionali del sistema, le loro responsabilità e interfacce, e le interazioni principali
- **delle Informazioni**: descrive il modo in cui l'architettura memorizza, manipola, gestisce e distribuisce informazioni – in termini di strutture di dati statiche e di flussi di informazioni
- **della Concorrenza**: escrive l'organizzazione della concorrenza e mappa gli elementi funzionali su unità di concorrenza, nonché le parti concorrenti del sistema e le loro necessità e modalità di sincronizzazione
- **dello Sviluppo**: descrive l'architettura che supporta il processo di sviluppo
- **del Deployment**: descrive l'ambiente in cui il sistema sarà rilasciato, comprese le dipendenze dall'ambiente runtime
- **Operazionale**: descrive come il sistema sarà usato, amministrato e supportato quando sarà in esecuzione nell'ambiente di produzione

Relazioni e dipendenza tra le viste