Lezioni di Ingegneria del software
A.A. 2021–2022
Corso di Laurea in Ingegneria informatica

Andrea Domenici

Dipartimento di Ingegneria dell'informazione
Università di Pisa

12 dicembre 2021

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Indice

Informazioni

Docente: Ing. Andrea Domenici andrea.domenici@unipi.it

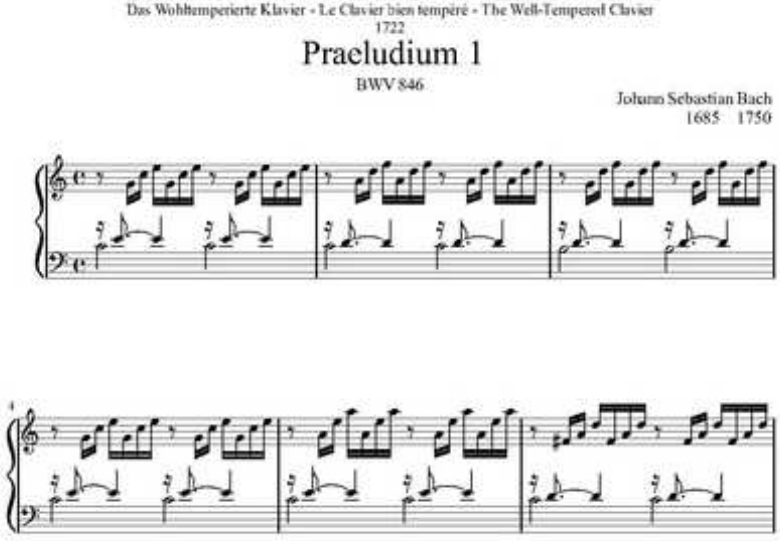
Pagina web:
http://www2.ing.unipi.it/~a009435/issw/isw.html

Orario effettivo:

lunedí	14:15–15:45
giovedí	15:15–16:45
venerdí	8:45–10:15

Le lezioni del venerdí mattina inizieranno il giorno 8/10.

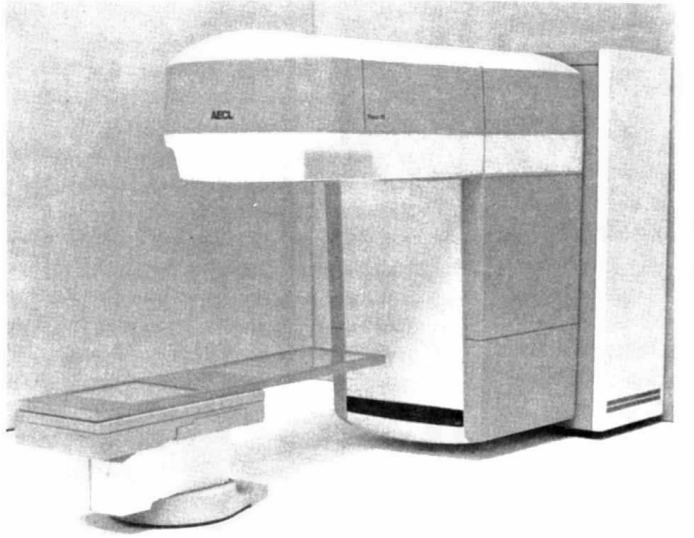
PRELUDIO


Das Wohltemperierte Klavier - Le Clavier bien tempéré - The Well-Tempered Clavier
1722
Praeludium 1
BWV 846
Johann Sebastian Bach
1685 1750

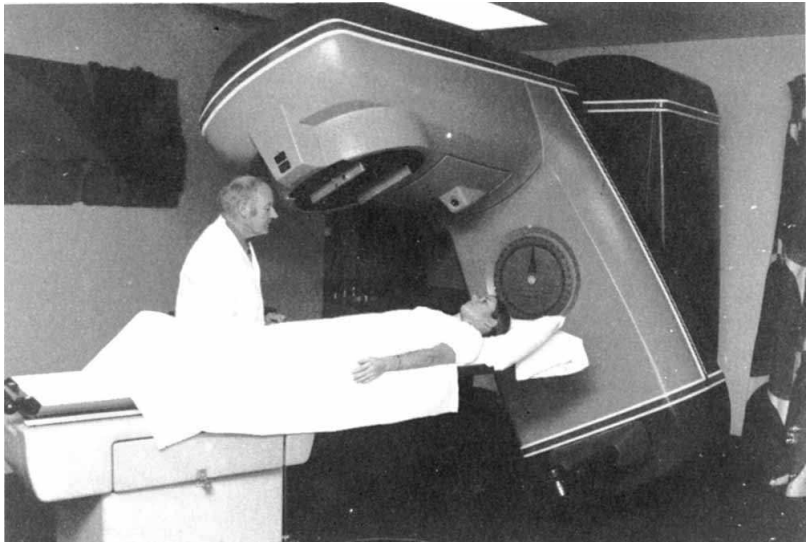
Indice

Preludio: il caso Therac-25 (17)

Il Therac-25 era una macchina per radioterapia che provocò almeno sei gravi incidenti di sovradosaggio, di cui tre mortali, fra il 1985 e il 1987.
http://www.ing.unipi.it/~a009435/issw/extra/therac.pdf



Preludio: il caso Therac-25 (2)



Il predecessore Therac-6, con blocchi di sicurezza (*interlock*) elettromeccanici ed operazione manuale e software.

Preludio: il caso Therac-25 (3)

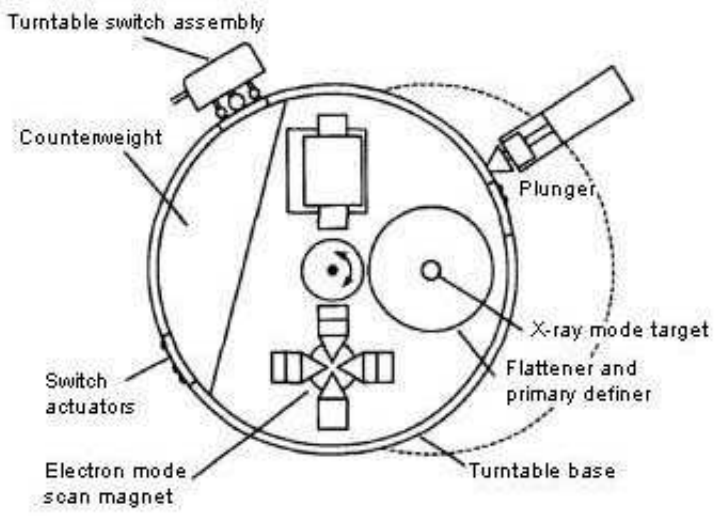


Figure B. Upper turntable assembly

Il vassoio portaaccessori girevole.

Preludio: il caso Therac-25 (4)

Il vassoio girevole aveva tre posizioni, corrispondenti a tre modi di operazione:

- *specchio*: per il puntamento con un raggio di luce, **a fascio spento**.
- *elettromagneti e camera di ionizzazione*: per diffondere e misurare il fascio di **elettroni, ad energia bassa o alta ed a bassa intensità di corrente**.
- *bersaglio, diffusore e camera di ionizzazione*: per generare, diffondere e misurare il fascio di raggi **X, ad alta energia ed alta intensità di corrente**.

Il sovradosaggio avveniva quando il fascio di elettroni per raggi X (25 MeV ed alta intensità di corrente) era attivo senza che gli appositi accessori fossero posizionati.

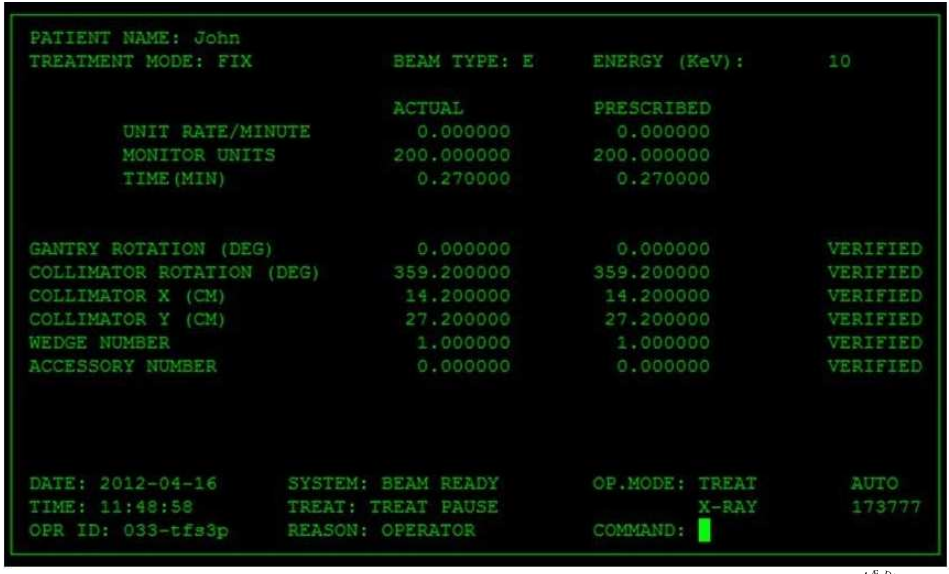
Preludio: il caso Therac-25 (5)

Genealogia:

- Therac-6 (raggi X): blocchi di sicurezza elettromeccanici, comando manuale (interruttori, potenziometri...) e software (terminale VT100).
- Therac-20 (raggi X ed elettroni): blocchi di sicurezza elettromeccanici, comando manuale e software.
- Therac-25 (raggi X ed elettroni): blocchi di sicurezza software, comando prevalentemente software, *stato dei sensori non controllato*.

Terminale Digital Equipment Corporation VT100.

Preludio: il caso Therac-25 (6)



Interfaccia utente del Therac-25.

Preludio: il caso Therac-25 (7)

L'operatore può inserire dei valori di default con un ritorno carrello.

Portando il cursore sulla linea `COMMAND`, l'operatore segnala il completamento dell'inserimento o modifica dei dati.

Il monitor mostra i parametri di funzionamento in due colonne:

- ACTUAL: valori rilevati dai sensori;
- PRESCRIBED: valori impostati dall'operatore.

Se i valori non corrispondono, sono possibili due condizioni di arresto:

- SUSPEND: richiede un reset completo (**reimpostando i parametri**) per riprendere il funzionamento;
- PAUSE: basta il comando **P** per riprendere, dopo aver corretto qualche parametro e **lasciando gli altri invariati**.

Preludio: il caso Therac-25 (7)

Gestione degli errori;

- molti messaggi consistevano solo in un numero, p.es., "`MALFUNCTION 54`";
- nessuna spiegazione nella documentazione;
- messaggi di significato ambiguo (e comunque non documentato):
  - `MALFUNCTION 54` $\rightarrow$ delivered dose **either too high or too low**;
- nessuna indicazione di livello di gravità;
- condizioni irrilevanti segnalate **frequentemente** come malfunzionamenti ("*al lupo, al lupo!*").

Preludio: il caso Therac-25 (8)

L'incidente dell'East Texas Cancer Center, Tyler, Texas, marzo 1986

- L'operatrice imposta velocemente i dati e porta il cursore su `COMMAND`;
- si accorge di aver scritto 'X' (raggi X) invece di 'E' (elettroni);
- corregge l'errore e usa il ritorno carrello per confermare gli altri dati;
- il display mostra `VERIFIED` e `BEAM READY`;
- l'operatrice accende il fascio (tasto **B**);
- la macchina va in `PAUSE` col messaggio `MALFUNCTION 54` e mostra una dose somministrata di 6 unità invece delle 202 richieste;
- l'operatrice preme **P**;
- `PAUSE`, `MALFUNCTION 54`, 6 unità;
- il paziente si alza dal tavolo e cerca di uscire dalla stanza.

Il paziente morí cinque mesi dopo.

Preludio: il caso Therac-25 (9)

L'incidente dell'East Texas Cancer Center, Tyler, Texas, aprile 1986

- simile al precedente, ma l'operatrice non cercò di ripetere il trattamento dopo il primo arresto della macchina.

Il paziente comunque morí tre settimane dopo.

Il problema SW (a grandi linee):

- ci sono diversi processi concorrenti, fra cui uno per gestire l'interfaccia e impostare i parametri ed uno per attivare i magneti di deflessione del fascio;
- i processi comunicano per mezzo di **variabili condivise**, ma **senza meccanismi di sincronizzazione** (semafori etc.).
- quando la sequenza *impostazione-correzione-accensione* viene completata prima che finisca l'attivazione dei magneti, il sistema mantiene i valori non corretti,
- risultando in un fascio alla massima energia ed intensità con la macchina predisposta per il trattamento a elettroni.

Preludio: il caso Therac-25 (10)

Un problema HW (a grandi linee):

- La camera di ionizzazione, esposta ad un fascio per raggi X senza l'attenuazione del bersaglio e del diffusore, si satura e dà valori inattendibili (6 dosi);
- il sistema non ha meccanismi per controllare che i sensori funzionino correttamente.

Preludio: il caso Therac-25 (11)

L'incidente dello Yakima Valley Memorial Hospital, gennaio 1987

- L'operatore esegue due trattamenti a raggi X per un totale di 7 rad;
- imposta i dati per un trattamento a raggi X da 79 rad;
- usa lo specchio di puntamento e dà il comando **set** per posizionare correttamente il vassoio;
- il monitor mostra `BEAM READY` e l'operatore preme **B**;
- la macchina va in pausa e non mostra alcuna dose somministrata;
- l'operatore preme **P** e la macchina torna in pausa, mostrando una dose di 7 rad;
- il paziente si lamenta.

Il paziente morí in aprile.

Preludio: il caso Therac-25 (12)

Il problema SW (a grandi linee):

- due processi concorrenti comunicano attraverso variabili condivise;
- la variabile `Class3` uguale a zero significa che i parametri sono corretti per il tipo di trattamento;
- la variabile `F$mal` uguale a zero significa che la macchina è pronta;
- il processo `SetUpTest` incrementa `Class3` finché i parametri non sono corretti;
- poi legge `F$mal` e passa alla fase successiva se `F$mal` è uguale a zero.
- il processo `Lmtchk` legge `Class3` e se questa è uguale a zero **non** controlla la posizione del vassoio; **ma**
- `Class3` va a zero per **overflow** ogni 127 volte che viene incrementata; **quindi**
- può accadere che il fascio venga abilitato anche se il vassoio non è posizionato correttamente.

Preludio: il caso Therac-25 (13)

Errori di programmazione

Sono stati individuati con certezza due particolari errori:

- *overflow*:
  - si usa un intero per rappresentare valori booleani, con la convenzione "*diverso da zero*" $\rightarrow$ FALSO;
  - per assegnare "FALSO" si incrementa la variabile invece di assegnare un valore costante.
- *sincronizzazione*:
  - non si usano meccanismi di protezione per l'accesso alle variabili condivise.

Preludio: il caso Therac-25 (14)

Problemi di progetto e di sistema

- sistema operativo real-time (RTOS) sviluppato ad hoc
  - quando era disponibile il DEC RT-11;
- sistema operativo scritto interamente in assembler
  - quando era disponibile il linguaggio C;
  - probabilmente monolitico;
- mancanza di *programmazione difensiva*
  - eccezioni, asserzioni, “trucchi” vari:
- interfaccia utente piú amichevole che sicuro;
- gestione di errori e malfunzionamenti inadeguata
  - vedi lucidi precedenti.

Preludio: il caso Therac-25 (15)

Errori di processo

- Documentazione di specifica e di progetto:
- controllo di qualità;
- valutazione dei rischi
  - malfunzionamenti SW non considerati;
- collaudo del SW
  - test di unità;
  - test di integrazione;
  - SW troppo complesso;
- risposta inadeguata alle segnalazioni fatte dagli utenti.

Preludio: il caso Therac-25 (16)

Errori di metodologia

- considerare solo errori HW
- valutazione pseudoquantitativa dei rischi
  - espressi come (bassi) valori di probabilità senza giustificazione sperimentale o matematica:
- riuso acritico del SW
  - in parte ripreso da Therac-6 e Therac-20
  - che avevano blocchi di sicurezza HW;
- confusione fra *affidabilità* e *sicurezza*;
  - *affidabilità*: capacità di funzionare a lungo senza guasti:
  - *sicurezza*: incapacità di fare danni:
- confusione fra *uso* e *collaudo*;
  - *uso*: operazione del sistema nel suo ambiente di applicazione, allo scopo di fornire i servizi richiesti:
  - *collaudo*: esercizio del sistema in ambiente controllato, allo scopo di individuare malfunzionamenti.

Preludio: l'incidente Ariane 5 Flight 501 (1)

Il razzo Ariane 5 si distrusse 37 secondi dopo il lancio nel giugno del 1997.
http://www.ing.unipi.it/~a009435/issw/extra/esa-x-1819eng.pdf
http://www.ing.unipi.it/~a009435/issw/extra/ariane5-benari.pdf

Malfunzionamento

- overflow in una conversione da 64 a 16 bit;
- arresto del sistema di riferimento inerziale (SRI);
- il "messaggio di errore" dell'SRI interpretato dal processore principale come un dato valido, **quindi**
- il processore principale comanda un brusco cambiamento di rotta.

Preludio: l'incidente Ariane 5 Flight 501 (2)

Il colpevole?

```
... codice Ada
declare
  horizontal_veloc_sensor: float;
  horizontal_veloc_bias: integer;
  ...
begin
  declare    -- "suppress" ELIMINA IL CONTROLLO A RUNTIME
    pragma suppress(numeric_error, horizontal_veloc_bias);
  begin
    sensor_get(horizontal_veloc_sensor);
    horizontal_veloc_bias := integer(horizontal_veloc_sensor);
    ...
  exception  -- L'ECCEZIONE "numeric_error" NON VIENE GESTITA
    when numeric_error => calculate_vertical_veloc();
    when others => use_irs1();
  end;
```

Preludio: l'incidente Ariane 5 Flight 501 (3)

Problemi di progetto e di sistema

- mancanza di defensive programming
  - nella versione precedente (Ariane 4) alcune operazioni non erano protette delle eccezioni
  - **per motivi di efficienza**
  - perché le protezioni erano considerate inutili;
- gestione di errori e malfunzionamenti inadeguata
  - arresto totale dell'SRI per eccezione HW;
- interfacciamento fra sottosistemi
  - flag di errore scambiata per dato normale;

Preludio: l'incidente Ariane 5 Flight 501 (4)

Errori di processo

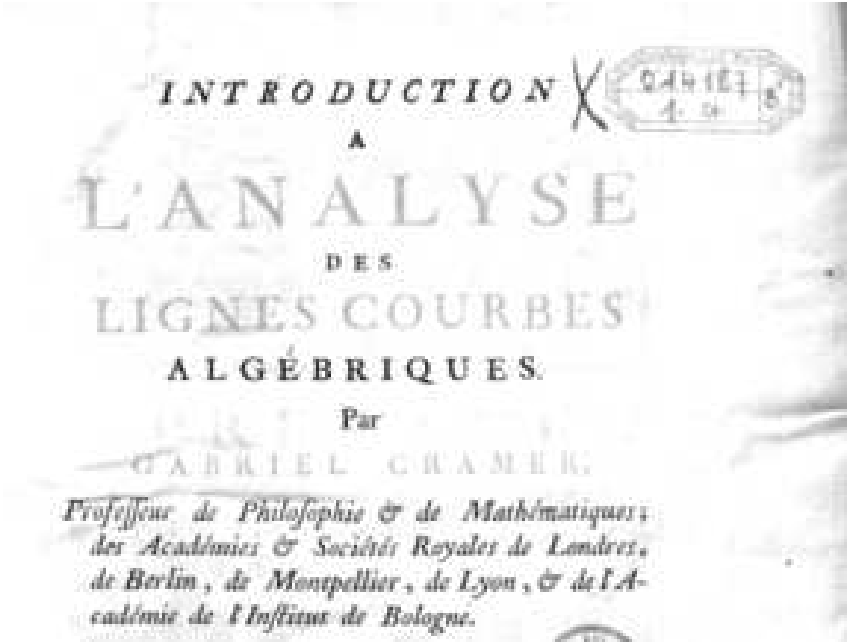
- specifiche incomplete
  - fra i requisiti dell'SRI manca la traiettoria della missione;
  - mancano i vincoli fisici del sistema;
- collaudi incompleti;

Preludio: l'incidente Ariane 5 Flight 501 (5)

Errori di metodologia

- considerare solo errori HW
  - il sistema aveva doppia ridondanza (2 SRI, 2 processori principali), **ma**
  - una sola versione di SW;
- riuso acritico del SW
  - overflow causato dalle diverse caratteristiche della traiettoria fra Ariane 4 e Ariane 5.
  - la funzione che causò l'overflow era inutile nell'Ariane 5!

INTRODUZIONE



INTRODUCTION
A
L'ANALYSE
DES
LIGNES COURBES
ALGÉBRIQUES.
Par
GABRIEL CRAMER,
Professeur de Philosophie & de Mathématiques; des Académies & Sociétés Royales de Londres, de Berlin, de Montpellier, de Lyon, & de l'Académie de l'Institut de Bologne.

Indice

Introduzione: prime definizioni

L'ingegneria del software non mi piace. Mi piacciono le cose pratiche, mi piace programmare in assembler.
– confessione autentica di uno studente

Ein Architekt ist ein Maurer, der Latein gelernt hat.
(un architetto è un muratore che ha studiato il latino.)
– Adolf Loos

- L'ingegneria del SW è una cosa pratica?
- Che differenza c'è fra *programmare* e *sviluppare SW*?
- Un ingegnere del SW è un programmatore che ha studiato il latino?

(*L'ingegneria del software non piaceva nemmeno al Prof. Dijkstra:* `http://www.ing.unipi.it/~a009435/issw/extra/EWD1036.pdf`.)

Introduzione: concetti generali

> *"L'ingegneria del software è il settore dell'informatica che si occupa della creazione di sistemi software talmente **grandi o complessi** da dover essere realizzati da piú **squadre** di ingegneri. Di solito questi sistemi esistono in varie **versioni** e rimangono in servizio per parecchi anni. Durante la loro vita subiscono numerose **modifiche**.*
>
> *— C. Ghezzi et al., Ingegneria del software: fondamenti e principi*

- In realtà, l'ISW è molto utile anche in progetti piccoli, fatti da un solo sviluppatore.

Introduzione: le parti in causa

Sviluppatore: chi partecipa direttamente allo sviluppo del SW, come *analisti*, *progettisti*, *programmatori*, o *collaudatori*. In alcuni casi il termine "sviluppatore" verrà contrapposto a "collaudatore".

Produttore: per "produttore" del SW si intende un'organizzazione che produce SW, o una persona che la rappresenta. Uno sviluppatore generalmente è un dipendente del produttore.

Committente: organizzazione o persona che chiede del SW al produttore.

Utente: una persona che usa il SW. Generalmente il committente è distinto dagli utenti, ma spesso si userà il termine "utenti" per riferirsi sia agli utenti propriamente detti che al committente.

- Spesso il SW non viene sviluppato per un committente particolare (applicazioni *dedicate*, o *custom*, *bespoke*), ma viene messo in vendita (o anche distribuito liberamente) come un prodotto di consumo (applicazioni *generiche*, o *shrink-wrapped*).

Introduzione: specifica e implementazione

Specifica: una descrizione precisa dei *requisiti* (proprietà o comportamenti richiesti) di un sistema o di una sua parte.

- descrive una certa entità “dall’esterno”, cioè dice quali servizi devono essere forniti o quali proprietà devono essere esibite da tale entità.

implementazione: un sistema che *soddisfa* (implementa) la specifica.

- generalmente, una specifica può avere piú implementazioni;
- nei rami tradizionali dell’ingegneria (civile, meccanica. . .), l’implementazione è un sistema fisico;
- nell’ingegneria informatica, l’implementazione è un *codice eseguibile*, **cioè**
- **un modello a basso livello** di un processo di calcolo.

Introduzione: modelli e linguaggi

- il processo di sviluppo del SW si può vedere come la costruzione di una serie di *modelli*.
- Un modello è una descrizione astratta di un sistema, che serve a studiarlo prendendone in considerazione soltanto quelle caratteristiche che sono necessarie al conseguimento di un certo scopo.
- Ogni sistema, quindi, dovrà essere rappresentato per mezzo di piú modelli, ciascuno dei quali ne mostra solo alcuni aspetti, spesso a diversi livelli di dettaglio.

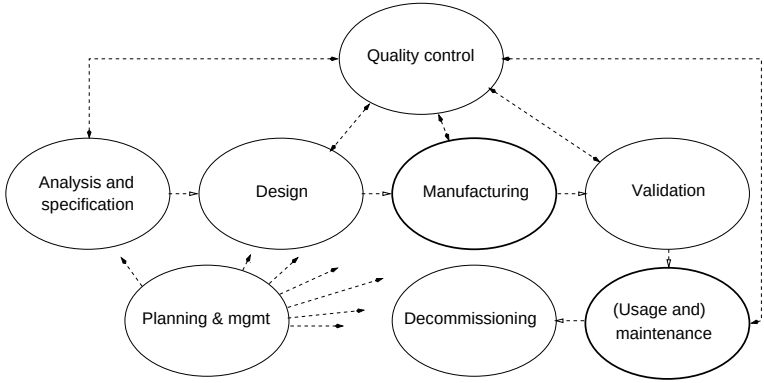
Un linguaggio offre

- un *vocabolario* di simboli (parole o segni grafici) che rappresentano i concetti necessari a descrivere certi aspetti di un sistema,
- una *sintassi* che stabilisce in quali modi si possono costruire delle espressioni (anche espressioni grafiche, cioè diagrammi), e
- una *semantica* che definisce il significato delle espressioni.

PROCESSI DI SVILUPPO

Indice

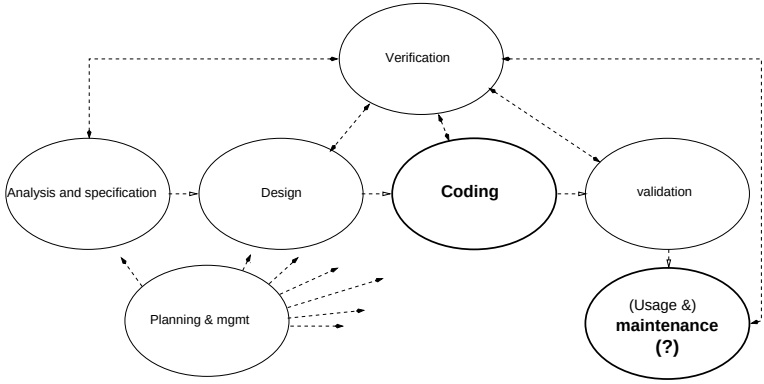
Processi di sviluppo: il ciclo di vita (1)



The *lifecycle* of a product is the set of *activities* affecting it from conception to retirement.

The lifecycle of software products is *similar* to the general lifecycle, **but** there are important differences.

Processi di sviluppo: il ciclo di vita (2)

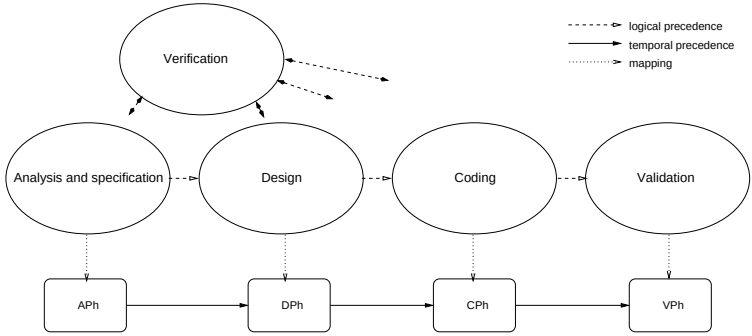


Coding is actually the final phase of design.

Software maintenance is actually redesign/recoding to correct or improve deployed software components.

In this course we ignore issues about software decommissioning.

Processi di sviluppo: i processi (1)

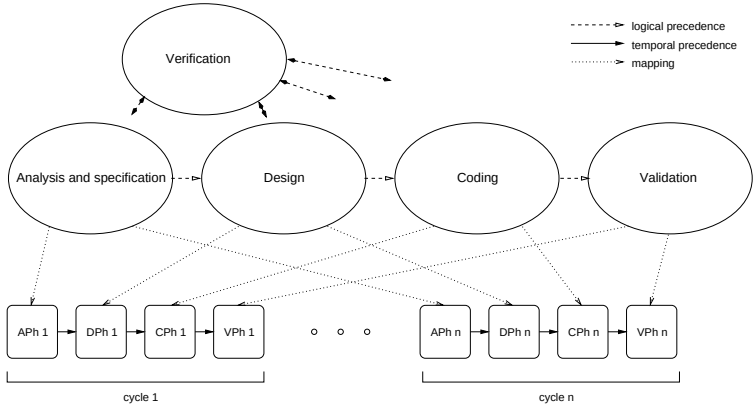


A *software development process* maps activities (i.e., things to be done) into *phases* (i.e., periods wherein activities are carried out).

Each phase has well-defined *milestones* (important events at planned dates) and produces some *deliverables* (documents or code).

The *waterfall* family of processes maps each activity to a distinct phase.

Processi di sviluppo: i processi (2)



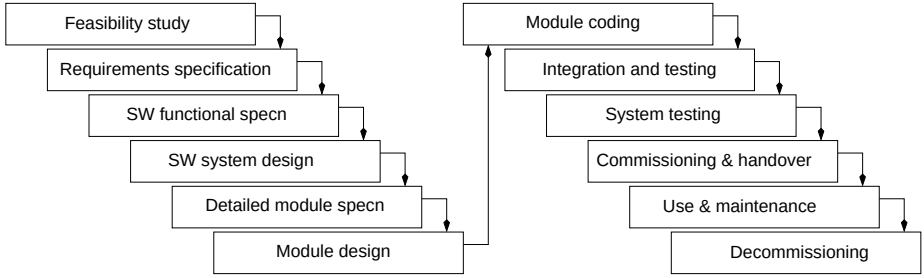
Other families of processes, such as the *iterative* processes, have more complex mappings from activities to phases.

The waterfall processes, however, are usually preferred (or mandatory) for software with safety requirements.

Processi di sviluppo: il modello a cascata (1)

(brunnur.stjr.is/embassy/strasb.nsf/pages/index.html)

Processi di sviluppo: il modello a cascata (2)



International Atomic Energy Agency, *Manual on Quality Assurance for Computer Software Related to the Safety of Nuclear Power Plants*, Tech. Reports Series No. 282, IAEA, Vienna, 1988.

Processi di sviluppo: il modello a cascata (3)

Nel séguito ci riferiremo ad un processo articolato nelle seguenti fasi:

- *studio di fattibilità*;
- *analisi e specifica dei requisiti*, suddivisa in
  - *analisi (definizione e specifica) dei requisiti dell'utente*, e
  - *specifica dei requisiti del software*;
- *progetto*, suddiviso in
  - progetto *architetturale*, e
  - progetto *in dettaglio*;
- *programmazione e test di unità*;
- *integrazione e test di sistema*;
- *manutenzione*.

Processi di sviluppo: il modello a cascata (4)

Studio di fattibilità

- stabilire se il prodotto può essere realizzato;
- stabilire se il è conveniente realizzarlo;
- possibili strategie di realizzazione alternative;
- tipo e quantità di risorse necessarie, quindi stima dei costi.

In base allo studio di fattibilità, **il committente decide se firmare o no il contratto** per la fornitura del software.

Gli errori di valutazione possono causare ritardi e inadempienze contrattuali, con perdite economiche per il fornitore o per il committente.

lo studio di fattibilità può essere fornito come prodotto finito, indipendente dall'eventuale prosecuzione del progetto:

- A chiede lo studio a B e affida il progetto a C.

Processi di sviluppo: il modello a cascata (5)

Analisi e specifica dei requisiti

- capire e descrivere nel modo piú completo e preciso possibile *che cosa vuole* il committente;
- risultati usati da persone con diversi ruoli; quindi
- diversi livelli di astrazione;
- *analisi del dominio*: descrive il contesto in cui opera il SW;
- *definizione dei requisiti* e *specifica dei requisiti*: descrivono i servizi richiesti a diverso livello di dettaglio;
- *specifica dei requisiti del software*: descrive le caratteristiche del SW;
  - **non l'implementazione**!
  - rappresentazione esterna dell'informazione, interfaccia utente, comportamento osservabile...

Processi di sviluppo: il modello a cascata (6)

Analisi e specifica dei requisiti: semilavorati

Documento di specifica dei requisiti (DSR): fondamentale; ha valore legale se incluso nel contratto;

- analisi del dominio: parti in causa, entità, concetti, relazioni;
- gli scopi dell'applicazione;
- i requisiti funzionali;
- i requisiti non funzionali;
- i requisiti sulla gestione del processo di sviluppo.

Manuale utente: Descrive il comportamento del sistema dal punto di vista dell'utente;

Piano di test di sistema: Definisce come verranno eseguiti i test finali per convalidare il prodotto rispetto ai requisiti; può avere valore legale.

Esempi di DSR su

http://www2.ing.unipi.it/~a009435/issw/esercitazioni/laboratorio.html

Processi di sviluppo: il modello a cascata (7)

Progetto

- decidere *come* deve essere fatto il sistema definito dai documenti di specifica;
- *scelte* fra le soluzioni possibili;
- *architettura software*:
  - **moduli**;
  - funzioni;
  - relazioni;
- *progetto architetturale*: moduli ad alto livello (**sottosistemi**);
- *progetto in dettaglio*: moduli a basso livello (**moduli unità**);

Progetto: semilavorati

Documento delle specifiche di progetto (DSP): definizione testuale e grafica dell'architettura SW;

Piano di test di integrazione: come collaudare l'interfacciamento fra i moduli nel corso della costruzione del sistema.

Processi di sviluppo: il modello a cascata (8)

Programmazione (codifica) e test di unità (a)

```
:programming: n. 1. The art of debugging a blank sheet of
paper (or, in these days of on-line editing, the art of
debugging an empty file). 2. A pastime similar to banging
one's head against a wall, but with fewer opportunities for
reward. 3. The most fun you can have with your clothes on
(although clothes are not mandatory).
```

– The Jargon File, v. 2.9.9, 01 APR 1992
(update: The Jargon File, v. 4.4.7)

Processi di sviluppo: il modello a cascata (9)

Programmazione (codifica) e test di unità (b)

- implementazione e collaudo dei singoli moduli;
  - scelta di strutture dati ed algoritmi;
- gestione delle versioni: *Subversion* (SVN), *git*;
- configurazione, compilazione e collegamento automatici (in ambienti “a linea di comando”): *Make*, *Automake*, *Autoconf*, *Libtool*;
- ambienti di sviluppo integrati: *MS Studio*, *Eclipse*, . . . ;
- documentazione del codice: *Doxygen*, *Javadoc*, . . . ;
- test di unità: *CppUnit*, *mockpp*, ...

Programmazione (codifica) e test di unità: semilavorati

Codice sorgente: dei moduli e dei programmi di prova;

Documentazione del codice sorgente: dei moduli e dei programmi di prova;

Documentazione dei test: compresi i dati di test.

Processi di sviluppo: il modello a cascata (10)

Integrazione e test di sistema

- vengono assemblati i sottosistemi a partire dai moduli componenti;
- effettuando parallelamente il *test di integrazione*;
- pianificazione e coordinamento fra gruppi di lavoro;
- *test di sistema*;
- *alfa test*;
- *beta test*.

Processi di sviluppo: il modello a cascata (11)

Manutenzione

- correzione di errori presenti nel prodotto consegnato al committente; oppure
- aggiornamento del codice allo scopo di fornire nuove versioni;
- **riprogettazione** del codice;
- *design for change*: anticipare la necessità di modificare il codice;
- tipi di manutenzione:

  correttiva: individuare e correggere errori;
  adattativa: cambiamenti di ambiente operativo (*porting*) (piattaforma hardware, leggi, lingue, ...);
  perfettiva: aggiunte e miglioramenti.

Processi di sviluppo: il modello a cascata (12)

Attività di supporto

gestione: pianificazione, sviluppo, allocazione delle risorse, flussi di informazione . . . ;

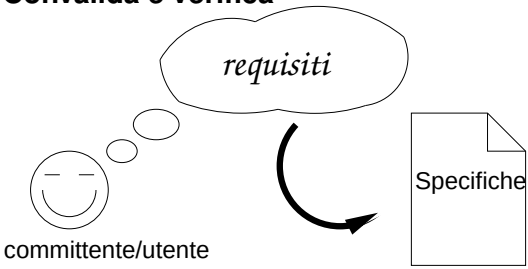
documentazione: semilavorati, rapporti sull'avanzamento dei lavori, linee guida per gli sviluppatori, minute delle riunioni, . . . ;

convalida e verifica: accertare che il prodotto ed i semilavorati soddisfino i requisiti;

controllo di qualità: standard (p.es., ISO 9000) e procedure.

Processi di sviluppo: il modello a V (1)

Convalida e verifica



- requisiti: il sistema immaginato dal committente;
- specifiche: descrizione (semi)formale del sistema.

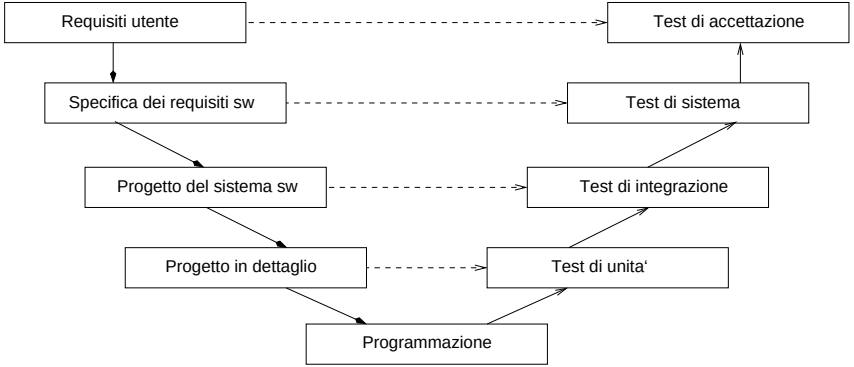
(N.B.: anche le singole clausole delle specifiche sono chiamate *requisiti*!)

- **Definizione 1**:
  - convalida = valutare **prodotto e semilavorati** rispetto ai **requisiti**;
  - verifica = valutare **prodotto e semilavorati** rispetto alle **specifiche**;
- **Definizione 2**:
  - convalida = valutare **solo il prodotto** rispetto ai **requisiti**;
  - verifica = valutare **i semilavorati** rispetto alle **specifiche**;

Il grado di formalità della verifica dipende da quello delle specifiche.

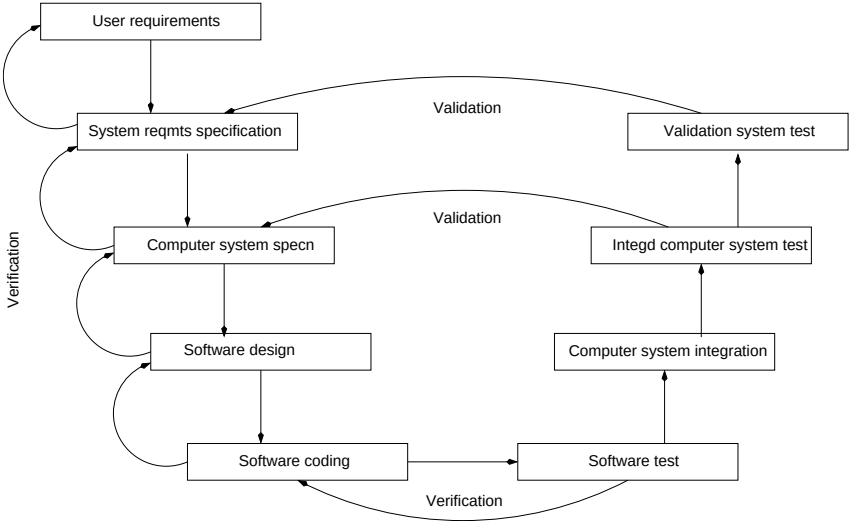
Processi di sviluppo: il modello a V (2)

Il *modello di processo a V* è una variante del modello a cascata in cui si mettono in evidenza le fasi di collaudo e la loro relazione con le fasi di sviluppo precedenti.



(http://homepages.laas.fr/waeselyn/papers/Testing-Lecture.pdf)

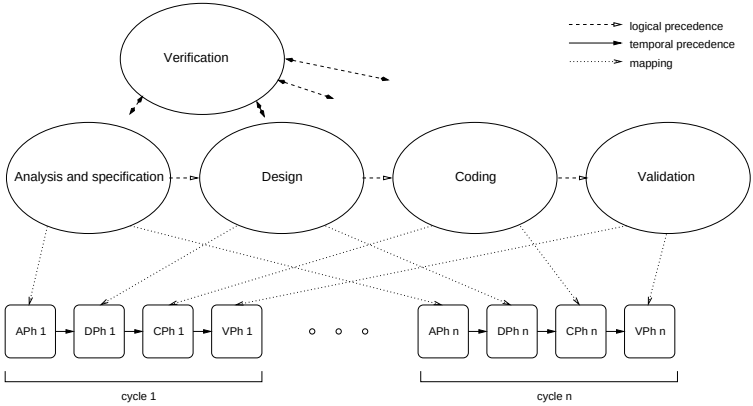
Processi di sviluppo: il modello a V (3)



The V-model (as in IAEA TRS 384).

Processi di sviluppo: lo Unified Process (1)

I processi evolutivi



- Rilevare tempestivamente la necessità di cambiamenti:
  - applicazioni o tecnologie nuove;
  - requisiti instabili;
- sviluppo **incrementale**.

Processi di sviluppo: lo Unified Process (2)

- l'arco temporale del processo di sviluppo è suddiviso in quattro *fasi* successive;
- ogni fase ha un obiettivo e produce un insieme di semilavorati chiamato *milestone* ("pietra miliare");
- ogni fase è suddivisa in un numero variabile di *iterazioni*;
- nel corso di ciascuna iterazione possono essere svolte tutte le attività richieste (analisi, progetto...), anche se, a seconda della fase e degli obiettivi dell'iterazione, alcune attività possono essere predominanti ed altre possono mancare;
- ciascuna iterazione produce una versione provvisoria (*baseline*) del prodotto, insieme alla documentazione associata.

Processi di sviluppo: lo Unified Process (3)

Attività (*workflow*)

- **raccolta dei requisiti** (*requirement capture*);
  - *modello dei casi d'uso*;
- **analisi** (*analysis*);
  - *modello di analisi*;
- **progetto** (*design*);
- **implementazione** (*implementation*);
- **collaudo** (*test*).

Processi di sviluppo: lo Unified Process (4)

Fasi (a)

Inizio (*inception*):
- studio di fattibilità;
- analisi dei rischi di varia natura (tecnica, economica, organizzativa...)
- prototipi
- modello dei casi d'uso: descrizione sintetica delle possibili interazioni degli utenti col sistema, espressa mediante la notazione UML;

Elaborazione (*elaboration*):
- estendere e perfezionare i risultati della fase precedente
- *baseline architetturale eseguibile*: prima versione eseguibile anche se parziale;
- non è un prototipo, ma una base per lo sviluppo successivo.
- milestone: codice della baseline, modello UML, versioni aggiornate dei documenti.

Processi di sviluppo: lo Unified Process (5)

Fasi (b)

Costruzione (*construction*):
- produce il sistema finale, partendo dalla baseline architetturale;
- completa le attività di raccolta dei requisiti, analisi e progetto;
- milestone: sistema completo, documentazione in UML, *test suite*, manuali utente. . . ;
- beta-test.

Transizione (*transition*):
- correzione degli errori trovati in corso di beta-test, consegna e messa in opera;
- milestone: versione definitiva del sistema, manuali utente, piano di assistenza tecnica.

Processi di sviluppo: lo Unified Process (6)

Distribuzione delle attività nelle fasi

Inizio:

- **raccolta e analisi dei requisiti**;
- progetto di architettura ad alto livello, non eseguibile;
- implementazione, se usato un prototipo.

Elaborazione:

- sforzo maggiore per raccolta e analisi dei requisiti, decrescente verso la fine;
- sforzo maggiore per elaborazione ed implementazione, crescente verso la fine.

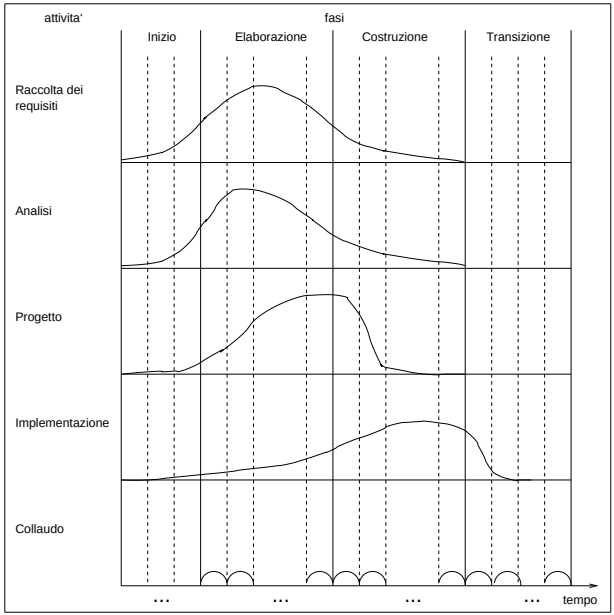
Costruzione:

- **progetto ed implementazione**;
- raccolta e analisi dei requisiti, per aggiornamenti dei requisiti.

Transizione:

- progetto ed implementazione per correggere errori.

Processi di sviluppo: lo Unified Process (7)



Processi di sviluppo: i modelli trasformazionali (1)

- Modelli **formali**;
- trasformazioni successive dal modello di analisi ad un modello di progetto dettagliato;
- generazione automatica di codice dal modello dettagliato;
- analogía con soluzione di equazioni:

$$f(x, y) = g(x, y)$$

$$\ldots$$

$$y = h(x)$$

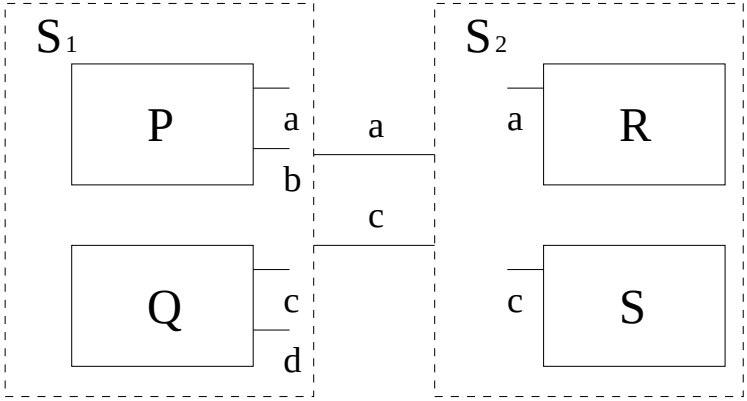
Processi di sviluppo: i modelli trasformazionali (2)

Esempio: LOTOS (a)

- **Algebra di processo**;
- un processo è un'espressione che rappresenta un'insieme di sequenze di **azioni**;
- i processi si possono combinare mediante **operatori**, p.es.:
  - *interleaving* ($|||$): esecuzione indipendente senza vincoli reciproci;
  - *sincronizzazione* ($|[\cdots]|$): esecuzione sincronizzata su azioni (i processi devono eseguire insieme le azioni su cui si sincronizzano);
- p.es., $(P[a,b] \;|||\; Q[c,d]) \;\; |[a,c]| \;\; (R[a] \;|||\; S[c])$ **analoga** (isomorfa) a $(P+Q)\cdot(R+S)$, cioè:
- l'algebra di ($|||, |[]|$) è isomorfa a quella di $(+,\cdot)$.

Processi di sviluppo: i modelli trasformazionali (3)

Esempio: LOTOS (b)



$(P[a, b] \;|||\; Q[c, d]) \;|[a, c]|\; (R[a] \;|||\; S[c])$

Processi di sviluppo: i modelli trasformazionali (4)

Esempio: LOTOS (c)

Subsystem S_1 is composed of processes P and Q. Subsystem S_2 is composed of processes R and S.

P can execute actions of type a and b, Q can execute actions of type c and d.

R can execute actions of type a, S can execute actions of type c.

P and Q execute their actions independently of each other: An *unconstrained* parallel composition. Similarly for R and S.

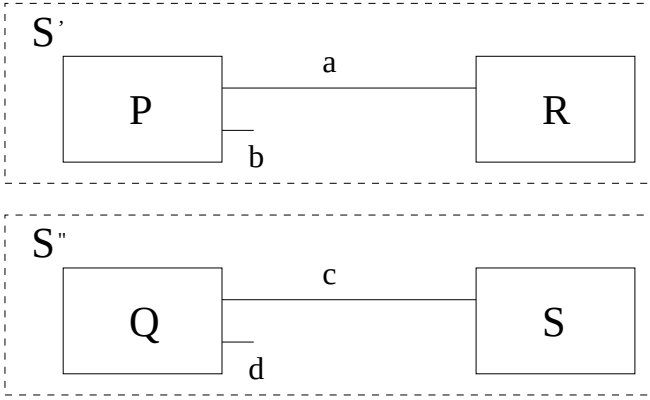
Subsystems S_1 and S_2 must execute "simultaneously" actions of type a or c: A *synchronized* parallel composition.

Processi di sviluppo: i modelli trasformazionali (5)

Esempio: LOTOS (d)

Within the LOTOS algebra, the previous expression can be transformed in an equivalent one, where the processes are rearranged.

The resulting system is equivalent to the initial one, with a simpler structure.



$$(P[a,b] \,||[a]|\, R[a]) \,|||\, (Q[c,d] \,||[c]|\, S[c])$$

Processi di sviluppo: i processi agili (1)

Materiale tratto da
http://www.ing.unipi.it/~a009435/issw/extra/viviani.pdf

Il *Manifesto agile* (http://agilemanifesto.org):

Persone e interazioni	sono più importanti dei processi e degli strumenti;
Un software funzionante	è più importante della documentazione;
Collaborare con i clienti	è più importante del contratto;
Aderire ai cambiamenti	è più importante che aderire al progetto.

*O*vvero, fermo restando il valore (e quindi la necessità) delle entità a destra, consideriamo più importanti le entità a sinistra.

Processi di sviluppo: i processi agili (2)

- Soddisfazione del cliente;
- Accogliere favorevolmente i cambiamenti anche se si è avanti con lo sviluppo;
- Usare scale temporali brevi (feedback piú rapido possibile);
- Eccellenza tecnica e buon design;
- Team che si auto regolano e persone motivate;
- Lavorare insieme quotidianamente (giocare per vincere);
- Semplicità del progetto e software funzionante;
- Iterazioni regolari e valutazione dell'attività (Sprint: 2 Settimane);
- Fornire a chi lavora al progetto l'ambiente, il supporto e la fiducia necessari al completamento del lavoro.

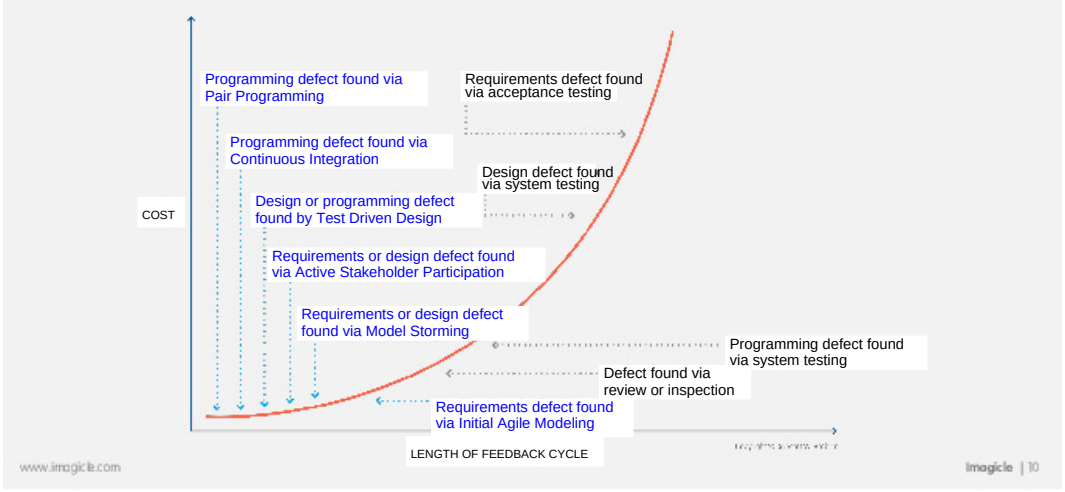
Processi di sviluppo: i processi agili (3)

Principali metodologie agili:

- Extreme Programming (XP);
- SCRUM;
- CRYSTAL;
- LEAN;
- KANBAN;
- FDD.

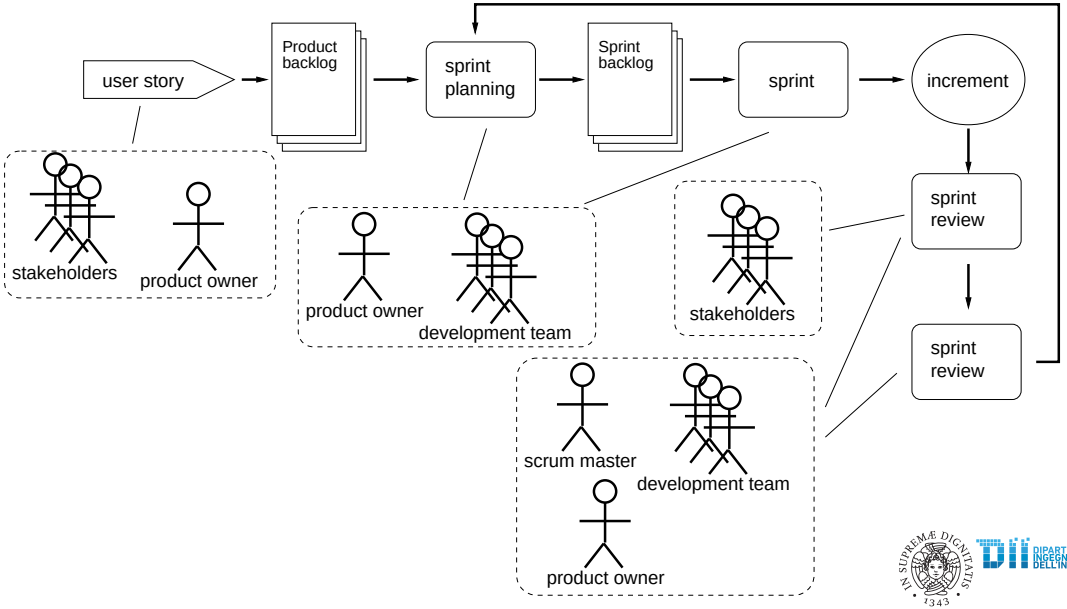
Processi di sviluppo: i processi agili (4)

Costo dei difetti in funzione del tempo di identificazione:



Processi di sviluppo: i processi agili (5)

Il processo *I'm Agicle*



Processi di sviluppo: i processi agili (7)

Il processo illustrato si basa sulle metodologie *Scrum* ed *XP*
Ogni **ciclo** (sprint) produce un **incremento**.

La durata (massima?) di ogni ciclo è **due settimane**.

- **sprint planning**: scelta degli obiettivi (dal *backlog*) da realizzare con l'incremento;
- **sprint review**: presentazione dell'incremento agli stakeholder;
- **sprint (development)**: realizzazione dell'incremento;
- **sprint retrospective**: discussione dello sprint concluso per preparare lo sprint successivo.

Processi di sviluppo: i processi agili (8)

Gli *artefatti*:

- **user story**: raccolta di documenti simili a requisiti o casi d'uso, ma diversi;
- **product backlog**: lista di user story da realizzare o bug da eliminare;
- **sprint backlog**: lista di elementi del product backlog scelti come obiettivi dello sprint corrente.

Processi di sviluppo: i processi agili (9)

I *partecipanti*:

- **stakeholders**: rappresentanti del committente;
- **product owner**: rappresentante del committente che *partecipa a tutte le attività insieme agli sviluppatori*;
- **scrum master**: facilitatore delle comunicazioni nel team e fra team e stakeholder, *partecipa alle attività di valutazione dello sprint*.

Processi di sviluppo: i processi agili (10)

User Story: esempi

come: utente
Voglio che ad ogni tipo di file sia associata una specifica icona
Al fine di riconoscere il tipo di file in modo veloce senza necessità di aprirlo

Business Value:
Story Point:

come: utente
Voglio he ad ogni tipo di file sia associato un programma in grado di aprirlo eseguendo il doppio click sul file stesso
Al fine di poter aprire direttamente il file senza dovermi ricordare quale programma usare.

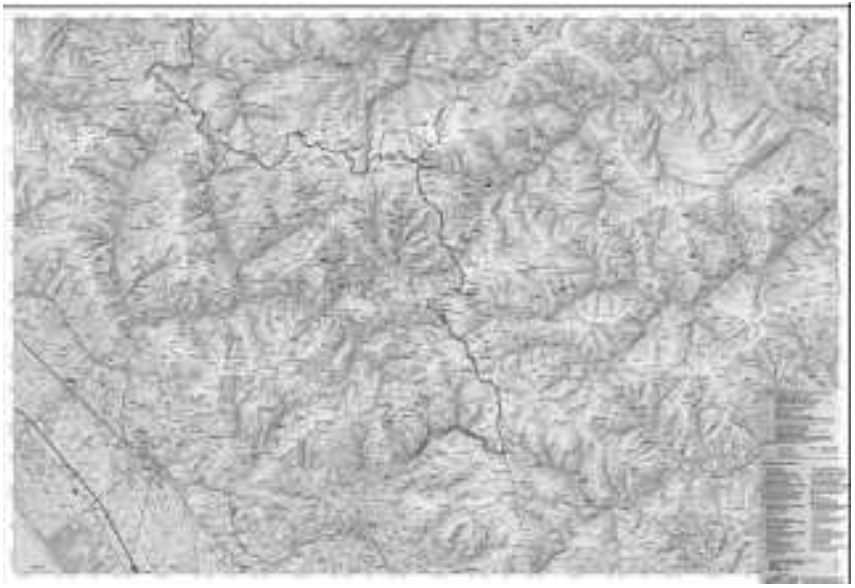
Business Value:
Story Point:

Processi di sviluppo: i processi agili (11)

Proprietà delle user story:

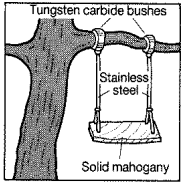
- **Independent** (per quanto possibile)
- **Negotiable** (Non sono contratti, possono essere aggiunte, rimosse modificate, unite, divise)
- **Valuable** (Devono avere valore per l'utente, l'utente in genere non è il committente)
- **Estimable** (Per gli sviluppatori è importante poter stabilire la dimensione della storia)
- **Small** (Sufficientemente piccola da poter rientrare in un'iterazione del team da 1 a 10 gg)
- **Testable** (Devono essere forniti i criteri di accettazione della storia)

ANALISI E SPECIFICA

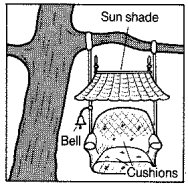


Indice

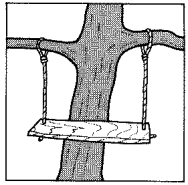
Analisi e specifica dei requisiti (ASR) (1)

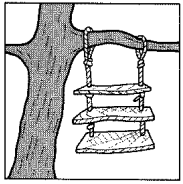


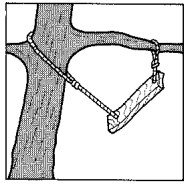
What Product Marketing specified

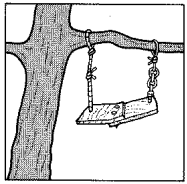


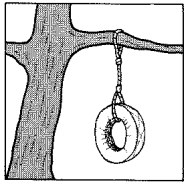
What the salesman promised


Design group's initial design


Corp. Product Architecture's modified design


Pre-release version

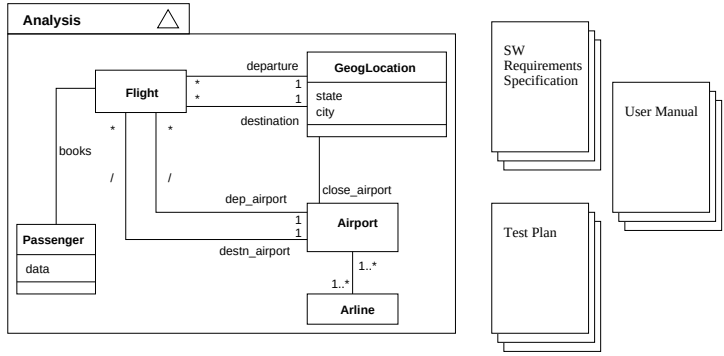

General release version


What the customer actually wanted

Analisi e specifica dei requisiti (2)

- Requirements *analysis* is the process of understanding what is required of a system.
- Requirements analysis involves studying the application domain, so that software developers may understand the relationship of the software with its environment.
- A requirements *specification* is the result of the requirements analysis process, i.e., a document that gives a precise and complete description of the requirements.
- The analysis process results in the construction of a *model*, i.e., an abstraction of the relevant aspects of the system and its environment.
- Using a specific analysis method helps making this model and its underlying assumptions more *explicit* and *verifiable*.

Analisi e specifica dei requisiti (3)



The analysis phase produces an *analysis* model.

The analysis model is defined by the *Software Requirements Specification* document.

Other deliverables include the *Test Plan* and *User Manual(s)*.

ASR: requisiti

- A *requirement* is a statement about a relevant aspect of a system that must be developed: a service it must deliver, a property or constraint it must satisfy, and so on.
- Some requirements may concern the development process itself, rather than the system; for example, it may be a requirement that the system is developed following certain standard procedures.
- *Functional* requirements describe the services offered by the system in terms of relationships between inputs and outputs, or between stimuli and responses (*reactive* systems).
- A system that satisfies its functional requirements is said to be *correct*.
- *Non-functional* requirements describe properties of, or constraints on, the system or process.
- Some non-functional requirements are *reliability*, *robustness*, *safety*, and *performance*.

ASR: requisiti, tracciabilità

> *Requirements should be traceable to design; design should be traceable to code; requirements, design and code should be traceable to tests.*
>
> *Traceability should be maintained when changes are made.*
>
> *There should be traceability in the reverse direction, to ensure that no unintended functions have been created.*

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

ASR: requisiti, testabilità

> *Each requirement and each design feature should be expressed in such a manner that a test can be done to determine whether that feature has been implemented correctly.*
>
> *Both functional and non-functional requirements should be testable.*
>
> *Test results should be traceable back to the associated requirements.*

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

ASR: vincoli

- *Constraints* are non-functional requirements that impose limitations on design choices.
- *Time constraints* are particularly important in *concurrent* systems, such as control systems, transaction systems (distributed databases), etc..
  - *Synchronization* constraints impose some *ordering* among events, without specifying bounds on time intervals between events.
    - E.g.: "*Valve B must not be opened before valve A*".
  - *Real-time* constraints impose *bounds* on time intervals between events.
    - E.g.: "*Valve B must be opened between 10 and 20 seconds after valve A*".

Note that a real-time system is not necessarily a fast system. *Predictability* is more important than speed.

ASR: requisiti non funzionali (1)

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.

ASR: requisiti non funzionali (2)

Dependability (a)

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.

ASR: requisiti non funzionali (3)

Dependability (b)

- **Fault avoidance**: *Software quality* is the first line of defense.
  - Rigorous development process.
  - Testing, verification, and validation.
  - Competence and experience.
- **Fault tolerance**: Specific provisions designed into the software are the second line of defense, against undetected design (or coding) errors and unforeseen accidents.
  - Compliance with system-level safety requirements and design.
  - System-level exception handling.
  - Choice of fault-tolerant mechanisms (redundancy, diversity, voting schemes. . .).
  - Issues specific to computer systems (interrupts, memory management, clocks. . .).

ASR: 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.

ASR: classi di sistemi secondo il 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.

ASR: linguaggi di specifica, grado di formalità (1)

formali: sintassi (testuale o grafica) e semantica definite rigorosamente;

semiformali: sintassi (generalmente grafica) rigorosa, semantica approssimativa;

informali: sintassi e semantica approssimative (linguaggio naturale).

ASR: linguaggi di specifica, grado di formalità (2)

- I linguaggi formali
  - non sono ambigui;
  - sono verificabili;
  - tradurre da linguaggio naturale a linguaggio formale chiarisce le idee (anche la programmazione è una traduzione).
- I linguaggi semiformali
  - aiutano a schematizzare i concetti;
  - sono espressivi.
- I linguaggi informali
  - sono ambigui,

ASR: linguaggi di specifica, grado di formalità (3)

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

ASR: linguaggi di specifica, stile di rappresentazione (1)

descrittivi: entità, proprietà, relazioni, senza riferimento al tempo;

operazionale: stati e transizioni, evoluzione del sistema.

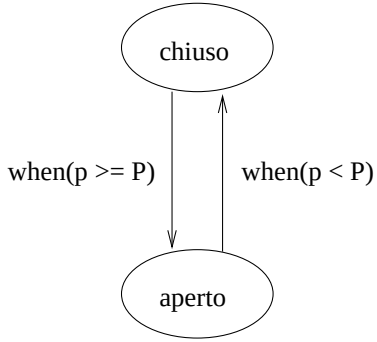
ASR: linguaggi di specifica, stile di rappresentazione (2)

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:



chiuso: valvola−chiusa and p < P

aperto: not valvola−chiusa and p

ASR: linguaggi di specifica, automi a stati finiti (1)

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*).

ASR: linguaggi di specifica, automi a stati finiti (2)

Un ASF (di Mealy) è definito da una sestupla

$$< S, I, U, d, t, s_0 >$$

- S insieme finito degli stati
- I insieme finito degli ingressi
- U insieme finito delle uscite
- $d : S \times I \rightarrow S$ funzione di transizione
- $t : S \times I \rightarrow U$ funzione di uscita
- $s_0 \in S$ stato iniziale

ASR: linguaggi di specifica, automi a stati finiti (3)

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.

ASR: linguaggi di specifica, automi a stati finiti (4)

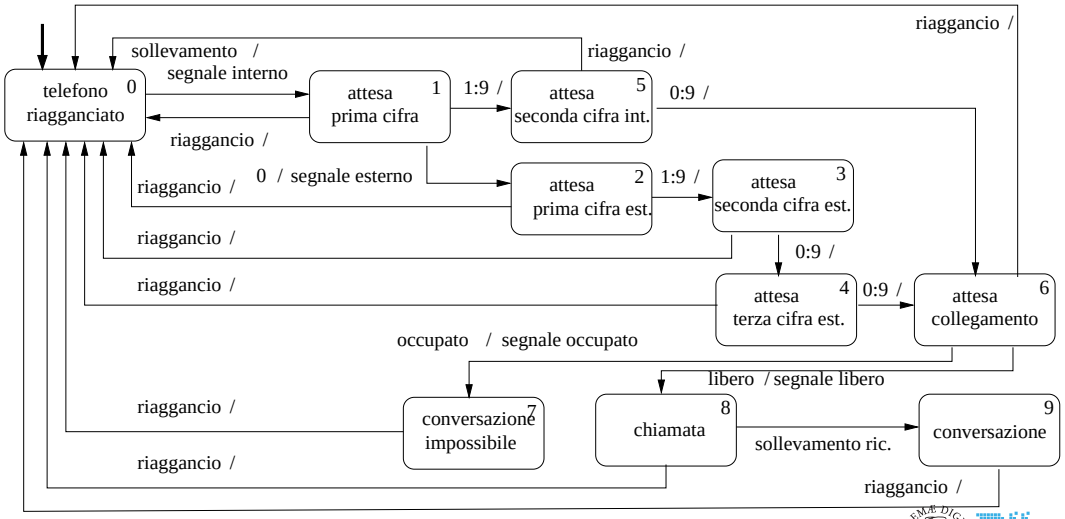
Esempio: centralino

- 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;
- s_0 = riagganciato.

Non si distingue fra riaggancio del chiamante e del chiamato.

ASR: linguaggi di specifica, automi a stati finiti (5)

Esempio: centralino



ASR: linguaggi di specifica, automi a stati finiti (6)

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.

ASR: linguaggi di specifica, automi a stati finiti (7)

Esempio: centralino
Domande:

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

ASR: linguaggi di specifica, automi a stati finiti (8)

Esempio: centralino, rappresentazione tabulare

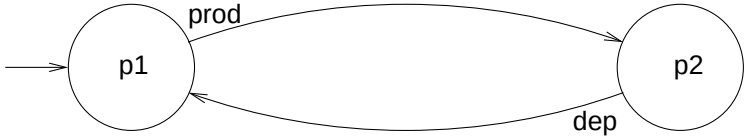
	0	1	2	3	4	5	6	7	8	9
0		S/int.								
1	R/		0/est.			1:9/				
2	R/			1:9/						
3	R/				0:9/					
4	R/						0:9/			
5	R/						0:9/			
6	R/							O/occ.	L/lib.	
7	R/									
8										Sr/
9	R/									

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

ASR: linguaggi di specifica, automi a stati finiti (9)

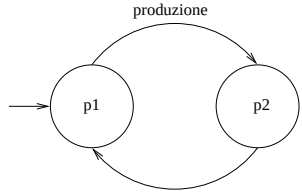
Esempio di linguaggio per sistemi concorrenti (SAL):



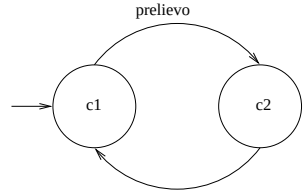

```
esempio: CONTEXT = BEGIN
  TStato: TYPE = {p1, p2}; TIngresso: TYPE = {prod, dep};
  produttore: MODULE = BEGIN
      INPUT azione: TIngresso
      OUTPUT stato: TStato      % vediamo la var. "stato" come uscita
      INITIALIZATION stato = p1 % stato iniziale
      TRANSITION
      (    stato = p1 AND azione = prod            % guardia
           --> stato' = p2                         % stato successivo
        [] stato = p2 AND azione = dep --> stato' = p1
        [] ELSE --> stato' = stato )
  END;
END
```

ASR: linguaggi di specifica, automi a stati finiti (10)

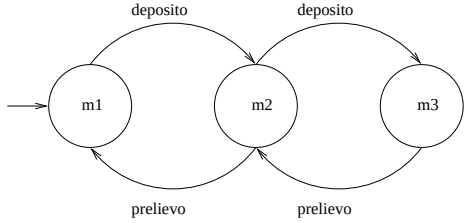
Composizione di ASF: produttore – magazzino – consumatore



produttore



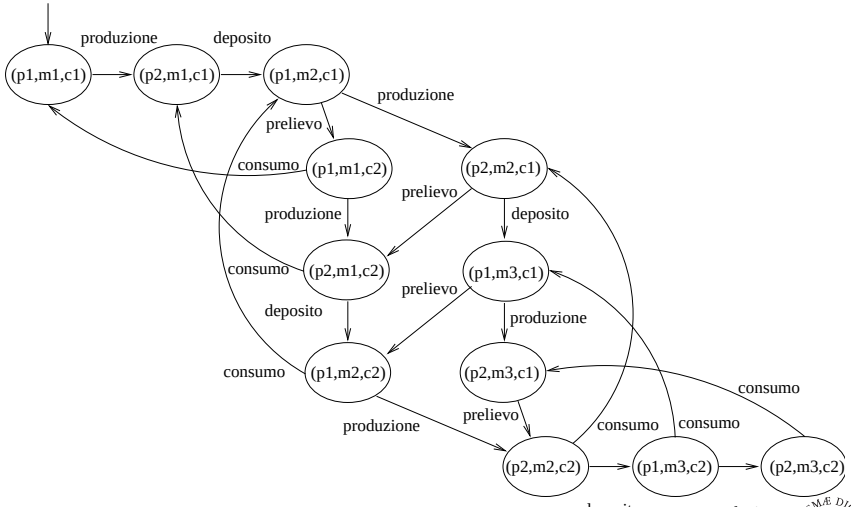
consumatore



magazzino

ASR: linguaggi di specifica, automi a stati finiti (11)

Composizione di ASF: produttore – magazzino – consumatore



ASR: linguaggi di specifica, automi a stati finiti (12)

Composizione di ASF: produttore – magazzino – consumatore

In questo tipo di modello, le transizioni rappresentano *azioni* eseguite dai diversi automi.

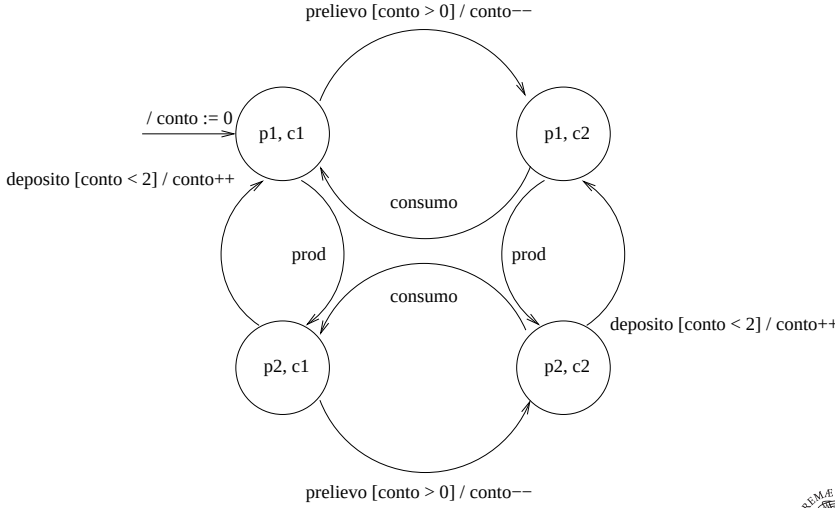
Le transizioni di automi diversi con la stessa etichetta rappresentano azioni *condivise* che devono essere eseguite insieme dagli automi (**sincronizzazione**).

Ad ogni passo viene eseguita una sola transizione. Questo *sequenzializza* un sistema intrinsecamente *concorrente*.

L'insieme degli stati risultante dalla composizione è il prodotto cartesiano degli insiemi di stati dei singoli automi. Per ridurre la complessità grafica, si ricorre a *estensioni* degli ASF, p.es., introducendo *variabili* associate all'automa e *guardie* associate alle transizioni.

ASR: linguaggi di specifica, automi a stati finiti (13)

Composizione di ASF: produttore – magazzino – consumatore



ASR: linguaggi di specifica, logica

Ramon Llull (1232–1315), *Ars magna*.

Formal logic

Formal logic is the conceptual framework that explicitly sets out the rules of sound reasoning. Formal logic enables us to make sure that a given line of reasoning (e.g., the demonstration of a theorem) is correct, i.e., the conclusions indeed follow from the premises.

Mathematics and the physical sciences are the classical fields of application for logic, but logic has become an important tool in technical applications, particularly in computer engineering.

Families of logics and formal systems
While the term *logic* refers in general to the science of formal reasoning, we speak of *a logic* or another to refer in particular to some specific way of using the general concepts of logic (just as we have different geometries, Euclidean, Riemannian, etc., within the field of geometry).

There exist several families of logics, with different purposes and expressiveness.

Within each logic, *formal systems* (or *theories*) are defined. Formal systems will be introduced later.

ASR: linguaggi di specifica, logica (3)

Languages

> *Was sich überhaupt sagen lässt, lässt sich klar sagen. (What can be said at all can be said clearly).* *L. Wittgenstein, Tractatus*

A logic language defines *what* we want to talk about, *the expressions* that we use to say what we mean, and how expression are given a meaning:

- *Domain*: the *individual entities* we talk about, and their reciprocal relationships.
  - E.g., the set of natural numbers, operations, ordering, equality.
- *Syntax*: the *symbols* denoting entities and relationships, and the *well-formedness rules* that say how correct *expressions* can be formed out of symbols. An expression that can be true or false is a (*declarative*) *sentence*, or *formula*.
  - E.g., the symbols $1, 2, 3, \ldots, +, -, \ldots, <, >, =, \ldots$. "$1 + 1$" is correct, "$1 + -2$" is not. "$1 < 3$" and "$1 > 3$" are sentences.
- *Semantics*: the rules that relate symbols to entities and relationships, and that decide which sentences are true.

ASR: linguaggi di specifica, logica (4)

Formal Systems

ASR: linguaggi di specifica, logica (5)

Formal Systems (1): interpretation
Given a language and its semantics, we can *interpret* any sentence of the language to see if it is true or false.

E.g., given the sentence "$1 + 1 = 2$", the semantics of the arithmetics language tell us that the symbols "1" and "2" correspond to the concepts of *number one* and *number two*, "$+$" corresponds to *sum*, and "$=$" corresponds to *equality*.

We can then verify (perhaps by counting on our fingers) that the sentence is true.

Things get more complicated when sentences refer to infinite sets, e.g., "*all primes greater than two are odd*"...

Formal Systems (2): definition

A *formal system* (or *theory*) is a "machine" that we use to *prove* the truth or falsehood of sentences by *deductions*, i.e., by showing that a sentence follows through a series of reasoning steps from some other sentences that are known (or assumed) to be true.

A formal system consists of:

- A *language*;
- a set of *axioms*, selected sentences taken as true[1].
- a set of *inference rules*, saying that a sentence of a given *structure* can be deduced from sentences of the appropriate structure, *independently of the meaning* (*semantics*) of the sentences.
  - E.g., if *A* and *B* stand for any two sentences, a well-known inference rule says that from *A* and "*A implies B*" we can deduce *B*.

[1] Or, more precisely, *valid*.

Formal Systems (3): inference

More precisely, an inference rule is a relationship between a set of (one or more) formulae called the rule's *premises*, and a formula called the (*direct*) *consequence* of the premises.

E.g., the rule mentioned in the previous slide (the *modus ponens*) is usually written as:

$$\frac{A \quad A \Rightarrow B}{B}$$

or

$$\frac{\begin{array}{c} A \\ A \Rightarrow B \end{array}}{B}$$

Note that this inference rule is a template that is matched by any pair of formulae, since A and B are placeholders for any formula.

Formal Systems (4): deduction and theorems

We have a formal system $\mathcal{F}$ with axioms $\mathcal{A}$ and inference rules $\mathcal{R}$

We want to prove that a formula *S* follows from a set $\mathcal{H}$ of hypotheses.

A *deduction* of *S* from $\mathcal{H}$ within $\mathcal{F}$ is a sequence of formulae such that *S* is the last one and each other formula either:

1. Belongs to $\mathcal{A}$; or
2. belongs to $\mathcal{H}$; or
3. is a direct consequence of some preceding formula in the sequence by some rule belonging to $\mathcal{R}$.

Formula *S* is then a *theorem*.

The application of an inference rule is a basic step in a formal line of reasoning (or *argument*).

ASR: linguaggi di specifica, logica (9)

Il calcolo proposizionale

- **simboli proposizionali**: rappresentano *sentenze non analizzabili*:
  - p.es. la frase "*il tempo è bello*", può essere vera o falsa, ma il linguaggio del CP non possiede dei simboli per denotare sostantivi (*tempo*) o proprietà (*essere bello*);
  - la frase può essere rappresentata da una lettera qualsiasi, o da una qualsiasi stringa di caratteri.
- **connettivi**: congiunzione, disgiunzione, negazione ...

ASR: linguaggi di specifica, logica (10)

Il calcolo proposizionale: sintassi

- un insieme numerabile $\mathcal{P}$ di *simboli proposizionali* (A,B,C, ...);
- un insieme finito di *connettivi*; per esempio: $\neg$ (*negazione*), $\wedge$ (*congiunzione*), $\vee$ (*disgiunzione*), $\Rightarrow$ (*implicazione*) ...
- un insieme di *simboli ausiliari* (parentesi e simili);
- un insieme $\mathcal{W}$ di *formule* (dette anche *formule ben formate*, *well-formed formulas*, o *wff*), definito dalle seguenti regole:
  1. un simbolo proposizionale è una formula;
  2. se $\mathcal{A}$ e $\mathcal{B}$ sono formule, allora sono formule anche $(\neg\mathcal{A})$, $(\mathcal{A} \wedge \mathcal{B})$, $(\mathcal{A} \vee \mathcal{B})$, ...
  3. solo le espressioni costruite secondo le due regole precedenti sono formule.

I connettivi vengono applicati (di solito) in quest'ordine:

$$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$$

per cui, ad esempio, la formula

$$A \Leftrightarrow \neg B \vee C \Rightarrow A$$

equivale a

$$A \Leftrightarrow (((\neg B) \vee C) \Rightarrow A)$$

Semantica del calcolo proposizionale: interpretazione

- l'insieme *booleano* $\mathbf{B} = \{\mathbf{T}, \mathbf{F}\}$, contenente i valori *vero* (**T**, *true*) e *falso* (**F**, *false*).
- una *funzione di valutazione* $v : \mathcal{P} \to \mathbf{B}$;
- le *funzioni di verità* di ciascun connettivo

$$\begin{array}{lcl} H_{\neg} & : & \mathbf{B} \to \mathbf{B} \\ H_{\wedge} & : & \mathbf{B}^2 \to \mathbf{B} \\ & \ldots & \end{array}$$

- una *funzione di interpretazione* $S_v : \mathcal{W} \to \mathbf{B}$ cosí definita:

$$\begin{array}{rcl} S_v(A) & = & v(A) \\ S_v(\neg\mathcal{A}) & = & H_{\neg}(S_v(\mathcal{A})) \\ S_v(\mathcal{A} \wedge \mathcal{B}) & = & H_{\wedge}(S_v(\mathcal{A}), S_v(\mathcal{B})) \\ & \ldots & \end{array}$$

dove $A \in \mathcal{P}$, $\mathcal{A} \in \mathcal{W}$, $\mathcal{B} \in \mathcal{W}$.

Semantica del calcolo proposizionale: funzione di valutazione
La funzione di valutazione è **arbitraria**, nel senso che viene scelta da chi si vuole servire di un linguaggio logico per rappresentare un certo dominio, in modo da riflettere ciò che si considera vero in tale dominio.

Per esempio, consideriamo le proposizioni A (*Aldo è bravo*), S (*Aldo passa l'esame di Ingegneria del Software*) e T (*il tempo è bello*). A ciascuna proposizione si possono assegnare valori di verità scelti con i criteri ritenuti piú adatti a rappresentare la situazione, per esempio la valutazione di A può essere fatta in base a un giudizio soggettivo sulle capacità di Aldo, la valutazione di S e T può essere fatta in base all'osservazione sperimentale o anche assegnata arbitrariamente, considerando una situazione ipotetica.

ASR: linguaggi di specifica, logica (13)

Semantica del calcolo proposizionale:
Di solito le funzioni di verità vengono espresse per mezzo di tabelle di verità come le seguenti:

x	y	$H_{\neg}(x)$	$H_{\wedge}(x,y)$	$H_{\vee}(x,y)$	$H_{\Rightarrow}(x,y)$	$H_{\Leftrightarrow}(x,y)$
F	**F**	**T**	**F**	**F**	**T**	**T**
F	**T**	**T**	**F**	**T**	**T**	**F**
T	**F**	**F**	**F**	**T**	**F**	**F**
T	**T**	**F**	**T**	**T**	**T**	**T**

Le funzioni di verità, a differenza della funzione di valutazione, sono parte integrante del linguaggio: poiché definiscono la semantica dei connettivi adottati dal linguaggio, non si possono modificare.

Semantica del calcolo proposizionale: soddisfacibilità
Se, per una formula $\mathcal{F}$ ed una valutazione v, si ha che $S_v(\mathcal{F}) = \mathbf{T}$, si dice che v *soddisfa* $\mathcal{F}$, e si scrive $v \models \mathcal{F}$.
Il simbolo $\models$ non appartiene al linguaggio del calcolo proposizionale, poiché non serve a costruire delle formule, ma è solo un'abbreviazione della frase "soddisfa", che appartiene al metalinguaggio, cioè al linguaggio di cui ci serviamo per parlare del calcolo proposizionale.
Una formula si dice *soddisfacibile* o *consistente* se esiste almeno una valutazione che la soddisfa. Per esempio:

$$A \Rightarrow \neg A \quad \text{per } v(A) = \mathbf{F}$$

Una formula si dice *insoddisfacibile* o *inconsistente* se non esiste alcuna valutazione che la soddisfi. Si dice anche che la formula è una *contraddizione*. Per esempio:

$$A \wedge \neg A$$

Semantica del calcolo proposizionale: validità
Una formula soddisfatta da tutte le valutazioni è una *tautologia*, ovvero è *valida*. La verità di una tautologia non dipende quindi dalla verità delle singole proposizioni che vi appaiono, ma unicamente dalla struttura della formula. Esempi di tautologie sono:

$$A \vee \neg A$$
$$A \Rightarrow A$$
$$\neg\neg A \Rightarrow A$$
$$\neg(A \wedge \neg A)$$
$$(A \wedge B) \Rightarrow A$$
$$A \Rightarrow (A \vee B)$$

Se $\mathcal{A} \Rightarrow \mathcal{B}$ (ove $\mathcal{A}$ e $\mathcal{B}$ sono formule) è una tautologia, si dice che $\mathcal{A}$ *implica logicamente* $\mathcal{B}$, ovvero che $\mathcal{B}$ è *conseguenza logica* di $\mathcal{A}$.

Il calcolo proposizionale: un sistema formale

- Il linguaggio $\mathcal{L}$ è costituito dalle formule ottenute a partire dai simboli proposizionali, dai connettivi $\neg$ e $\Rightarrow$, e dalle parentesi.
- Gli assiomi sono tutte le espressioni date dai seguenti *schemi*:

$$\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{A}) \tag{1}$$

$$(\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{C})) \Rightarrow ((\mathcal{A} \Rightarrow \mathcal{B}) \Rightarrow (\mathcal{A} \Rightarrow \mathcal{C})) \tag{2}$$

$$(\neg\mathcal{B} \Rightarrow \neg\mathcal{A}) \Rightarrow ((\neg\mathcal{B} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{B}) \tag{3}$$

  Questa teoria formale ha quindi un insieme infinito di assiomi; osserviamo anche che qualsiasi assioma ottenuto da questi schemi è una tautologia.
- L'unica regola d'inferenza è il *modus ponens* (MP): una formula $\mathcal{B}$ è conseguenza diretta di $\mathcal{A}$ e $\mathcal{A} \Rightarrow \mathcal{B}$. Si scrive anche

$$\frac{\mathcal{A} \qquad \mathcal{A} \Rightarrow \mathcal{B}}{\mathcal{B}}$$

La logica del primo ordine: sintassi

A *First-order logic* (FOL) is based on a language consisting of:

- A countable set $\mathcal{C}$ of *constant* symbols, denoting individual entities of the domain;
- a countable set $\mathcal{F}$ of *function* symbols, denoting functions in the domain;
- a countable set $\mathcal{V}$ of *variable* symbols, i.e., placeholders that stand for unspecified *individual entities*;
- a countable set $\mathcal{P}$ of *predicate* symbols, denoting relationships in the domain.
- a finite set of *logical connectives*, e.g. $\neg, \wedge, \vee, \Rightarrow, \ldots$;
- a finite set of *quantifiers*, e.g. $\forall, \exists$.

La logica del primo ordine: sintassi
A *term* is a constant, a variable, or a function symbol applied, recursively, to an *n*-tuple of terms.

A term is an expression that denotes an individual entity.

An *atomic formula* (or *atom*) is a predicate symbol applied to an *n*-tuple of terms.

An atom is an expression whose semantics is *true* iff the entities denoted by its terms satisfy the relationships denoted by the predicate symbol.

A *formula* is an atom, or an expression obtained by recursively combining expressions with quantifiers and connectives.

ASR: linguaggi di specifica, logica (19)

La logica del primo ordine: semantica

- l'insieme $\mathbf{B} = \{\mathbf{T}, \mathbf{F}\}$;
- le *funzioni di verità* di ciascun connettivo;
- un insieme non vuoto $\mathcal{D}$, detto *dominio* dell'interpretazione;
- una *funzione di interpretazione delle funzioni* $\Phi : \mathcal{F} \to \mathcal{F}_{\mathcal{D}}$, dove $\mathcal{F}_{\mathcal{D}}$ è l'insieme delle funzioni su $\mathcal{D}$. Φ assegna a ciascun simbolo *n*-ario di funzione una funzione $\mathcal{D}^n \to \mathcal{D}$;
- una *funzione di interpretazione dei predicati* $\Pi : \mathcal{P} \to \mathcal{R}_{\mathcal{D}}$, dove $\mathcal{R}_{\mathcal{D}}$ è l'insieme delle relazioni su $\mathcal{D}$; Π assegna a ciascun simbolo *n*-ario di predicato una funzione $\mathcal{D}^n \to \mathbf{B}$;
- la terna $I = (\mathcal{D}, \Phi, \Pi)$, detta *interpretazione*;
- un *assegnamento di variabili* $\xi : \mathcal{V} \to \mathcal{D}$;
- un *assegnamento di termini* $\Xi : \mathcal{T} \to \mathcal{D}$ cosí definito:

$$\begin{aligned} \Xi(x) &= \xi(x) \\ \Xi(f(t_1, \ldots, t_n)) &= \Phi(f)(\ \Xi(t_1), \ldots, \Xi(t_n)\ ) \end{aligned}$$

dove $x \in \mathcal{V}$, $t_i \in \mathcal{T}$, $f \in \mathcal{F}$;

Semantica della logica del primo ordine: interpretazione

- una *funzione di interpretazione* $S_{I,\xi} : \mathcal{W} \to \mathbf{B}$ cosí definita:

$$
\begin{aligned}
S_{I,\xi}(p(t_1,\ldots,t_n)) &= \Pi(p)(\,\Xi(t_1),\ldots,\Xi(t_n)\,) \\
S_{I,\xi}(\neg\mathcal{A}) &= H_\neg(S_{I,\xi}(\mathcal{A})) \\
S_{I,\xi}(\mathcal{A}\wedge\mathcal{B}) &= H_\wedge(S_{I,\xi}(\mathcal{A}), S_{I,\xi}(\mathcal{B})) \\
&\ldots \\
S_{I,\xi}(\exists x\mathcal{A}) &= \mathbf{T} \text{ se e solo se esiste un } d \in \mathcal{D} \text{ tale che } [S_{I,\xi}]_{x/d}(\mathcal{A}) = \mathbf{T} \\
S_{I,\xi}(\forall x\mathcal{A}) &= \mathbf{T} \text{ se e solo se per ogni } d \in \mathcal{D} \text{ si ha } [S_{I,\xi}]_{x/d}(\mathcal{A}) = \mathbf{T}
\end{aligned}
$$

dove $p \in \mathcal{P}$, $t_i \in \mathcal{T}$, $\mathcal{A}, \mathcal{B} \in \mathcal{W}$, $x \in \mathcal{V}$, e $[S_{I,\xi}]_{x/d}$ è la funzione di interpretazione uguale a $S_{I,\xi}$ eccetto che assegna alla variabile x il valore d.

ASR: linguaggi di specifica, logica (21)

Semantica della logica del primo ordine: soddisfacibilità e validità
Una formula $\mathcal{A}$ è *soddisfacibile in un'interpretazione* I se e solo se esiste un assegnamento di variabili ξ tale che $S_{I,\xi}(\mathcal{A}) = \mathbf{T}$.

L'interpretazione I *soddisfa* $\mathcal{A}$ con assegnamento di variabili ξ, e si scrive $I \overset{\xi}{\models} \mathcal{A}$.

Una formula $\mathcal{A}$ è *soddisfacibile* (tout-court) se e solo se esiste un'interpretazione I in cui $\mathcal{A}$ è soddisfacibile.

Una formula $\mathcal{A}$ è *valida in un'interpretazione* (o *vera in un'interpretazione*) I se e solo se $S_{I,\xi}(\mathcal{A}) = \mathbf{T}$ per ogni un assegnamento di variabili ξ. Si dice quindi che I è un *modello* di $\mathcal{A}$, e si scrive $I \models \mathcal{A}$.

Una formula *aperta* $\mathcal{A}$, cioè contenente variabili non quantificate (dette *libere*), è valida in un'interpretazione I se e soltanto se è valida la sua *chiusura universale*, cioè la formula ottenuta da $\mathcal{A}$ quantificando universalmente le sue variabili libere.

Una formula $\mathcal{A}$ è (*logicamente*) *valida* se e solo se è valida per ogni interpretazione I, e si scrive $\models \mathcal{A}$.

La logica del primo ordine: un sistema formale

- A first-order language with just two connectives ($\neg$ and $\Rightarrow$) and one quantifier ($\forall$);
- The following *axiom schemata*:

$$A \Rightarrow (B \Rightarrow A) \tag{4}$$
$$(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C)) \tag{5}$$
$$(\neg B \Rightarrow \neg A) \Rightarrow ((\neg B \Rightarrow A) \Rightarrow B) \tag{6}$$
$$\forall x A(x) \Rightarrow A(t) \tag{7}$$
$$\forall x(A \Rightarrow B) \Rightarrow (A \Rightarrow \forall x B) \tag{8}$$

- The following rules of inference:

$$\frac{A \quad A \Rightarrow B}{B}$$

$$\frac{\mathcal{A}}{\forall x \mathcal{A}}$$

La logica del primo ordine: un sistema formale

Allo schema di assioma 7 bisogna aggiungere la restrizione che t sia *libero per* x *in* $\mathcal{A}$, cioè che non contenga variabili che in $\mathcal{A}$ abbiano un quantificatore universale il cui campo d'azione includa occorrenze di x; questo vincolo impedisce che variabili libere nel termine t diventino quantificate quando t viene sostituito a x.

Per esempio, la formula

$$\forall x(\neg\forall y(x = y)) \Rightarrow \neg(\forall y(y = y))$$

("*se per ogni x esiste un y diverso da x, allora non è vero che ogni y è uguale a se stesso*") **sembra** un'istanza di 7 con $\neg\forall y(x = y)$ al posto di $A(x)$ e y al posto di t, ma questa sostituzione non è ammissibile perché y non è libero per x nella formula $(\neg\forall y(x = y))$.

La logica del primo ordine: un sistema formale

Allo schema di assioma 8 bisogna aggiungere la restrizione che $\mathcal{A}$ non contenga occorrenze libere di x. Per esempio, se $\mathcal{A}$ e $\mathcal{B}$ corrispondono tutte e due alla formula $p(x)$, dove $p(x)$ viene interpretato come "*x è pari*", l'applicazione dell'assioma 8 porterebbe alla formula $\forall x(p(x) \Rightarrow p(x)) \Rightarrow (p(x) \Rightarrow \forall x\ p(x))$, che è falsa: l'antecedente dell'implicazione principale $(\forall x(p(x) \Rightarrow p(x)))$ significa che "*per ogni x la parità implica la parità*", e il conseguente $((p(x) \Rightarrow \forall x\ p(x)))$ significa che "*la parità di un numero implica la parità di tutti i numeri*"

Verifica del software: esempio

Vogliamo specificare la relazione fra un vettore arbitrario x di $N > 2$ elementi ed il vettore y ottenuto ordinando x in ordine crescente, supponendo che gli elementi del vettore abbiano valori distinti. Possiamo esprimere questa relazione cosí:

$$
\begin{aligned}
ord(x, y) &\Leftrightarrow permutazione(x, y) \wedge ordinato(y) \\
permutazione(x, y) &\Leftrightarrow \forall k((1 \leq k \wedge k \leq N) \Rightarrow \\
&\quad \exists i(1 \leq i \wedge i \leq N \wedge y_i = x_k) \wedge \\
&\quad \exists j(1 \leq j \wedge j \leq N \wedge x_j = y_k)) \\
ordinato(x) &\Leftrightarrow \forall k(1 \leq k \wedge k < N \Rightarrow x_k \leq x_{k+1})
\end{aligned}
$$

Possiamo verificare la correttezza, rispetto alla specifica, di un programma che ordina un vettore di N elementi. Trascureremo la parte di verifica relativa alla permutazione di un vettore.

Verifica del software: esempio

Annotiamo il codice con le condizioni che devono valere per ogni iterazione (*invarianti*) e quelle che devono valere all'uscita di ciascun blocco (*postcondizioni*).

```
for (i = 0; i < M; i++) {                    // 0 ≤ i < M
    for (j = 0; j < M-i; j++) {              // 0 ≤ j < M − i
        if (v[j] > v[j+1]) {
            t = v[j];
            v[j] = v[j+1];
            v[j+1] = t;
        } // v_j ≤ v_{j+1}                                   (i)
    } // ∀k(M − i − 1 ≤ k < M ⇒ v_k ≤ v_{k+1})               (ii)
} // ∀k(0 ≤ k < M ⇒ v_k ≤ v_{k+1})                          (iii)
```

Verifica del software: esempio

L'asserzione (i) vale dopo l'esecuzione dell'istruzione `if`, per il valore corrente di `j`.

Le asserzioni (ii) e (iii) valgono, rispettivamente, all'uscita del loop interno (quando sono stati ordinati gli ultimi $i + 2$ elementi) e del loop esterno (quando sono stati ordinati tutti gli elementi).

Le altre due asserzioni esprimono gli intervalli dei valori assunti da `i` e `j`. La forma $A \leq B < C$ è un'abbreviazione di $A \leq B \wedge B < C$.

L'asserzione (i) può essere verificata informalmente considerando le operazioni svolte nel corpo dell'istruzione `if` (una verifica formale richiede una definizione della semantica dell'`if` e dell'assegnamento).

Verifica del software: esempio

Verifichiamo la (ii) per induzione sul valore di i, che controlla il ciclo esterno.

Per $i = 0$, la (ii) diventa

$$\forall k(M - 1 \leq k < M \Rightarrow v_k \leq v_{k+1})$$

La variabile k assume solo il valore $M - 1$, e j varia fra 0 ed $M - 1$, pertanto all'uscita del ciclo si ha dalla (i) che $v_{M-1} \leq v_M$, e la (ii) è verificata per $i = 0$.

Per un $\bar{\imath}$ arbitrario purché minore di $M - 1$, la (ii) (che è vera per l'ipotesi induttiva) assicura che gli ultimi $\bar{\imath} + 2$ elementi del vettore sono ordinati. Sono state eseguite $M - \bar{\imath}$ iterazioni del ciclo interno, e (dalla (i)) $v_{M-\bar{\imath}-1} < v_{M-\bar{\imath}}$.

Per $i = \bar{\imath} + 1$, il ciclo interno viene eseguito $M - \bar{\imath} - 1$ volte, e all'uscita del ciclo $v_{M-\bar{\imath}-2} < v_{M-\bar{\imath}-1}$. Quindi gli ultimi $i + 2$ elementi sono ordinati.

La formula (iii) si ottiene dalla (ii) ponendovi $i = M - 1$, essendo $M - 1$ l'ultimo valore assunto da i. Risulta quindi che all'uscita del ciclo esterno il vettore è ordinato, q.e.d.

Sequent calculus

The sequent calculus is a class of formal systems, each based on a specific logic language. The sequent calculus works on (meta)expressions of a special form, called *sequents*, such as:

$$A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m$$

where the A_i's are the *antecedents* and the B_i's are the *consequents*.

Each antecedent or consequent, in turn, is a formula of any form (it may contain subformulae with quantifiers and connectives, but not "sub-sequents") in the underlying language.

The symbol in the middle ($\vdash$) is called a *turnstile* and may be read as "*yields*".

Informally, a sequent can be seen as another notation for

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n \Rightarrow B_1 \vee B_2 \vee \ldots \vee B_m$$

ASR: linguaggi di specifica, logica (30)

Sequent calculus

A sequent is **proved** if:

- Any formula occurs both as an antecedent and as a consequent; or
- any antecedent is false; or
- any consequent is true.

In the PVS prover interface (*infra*), a sequent is represented as:

```
{-1}   A1
 ...
[-n]   An
  |-------
{1}    B1
 ...
[m]    Bm
```

Sequent calculus: inference rules

The sequent calculus has one axiom: $\Gamma, A \vdash A, \Delta$ where Γ and Δ are (multi)sets of formulae.

$$\frac{}{\Gamma, A \vdash A, \Delta}\,\text{axm} \qquad \frac{\Gamma \vdash \Delta, A \quad A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\,\text{cut} \qquad \frac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\,\text{ctr L} \qquad \frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}\,\text{ctr R}$$

$$\frac{\Gamma \vdash \Delta, A}{\neg A, \Gamma \vdash \Delta}\,\neg L \qquad \frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg A}\,\neg R \qquad \frac{A, B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta}\,\wedge L \qquad \frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}\,\wedge R$$

Inference rules:

$$\frac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta}\,\vee L \qquad \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B}\,\vee R \qquad \frac{\Gamma \vdash \Delta, A \quad B, \Gamma \vdash \Delta}{A \Rightarrow B, \Gamma \vdash \Delta}\,\Rightarrow L \qquad \frac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash A \Rightarrow B, \Delta}\,\Rightarrow R$$

$$\frac{A[x \leftarrow t], \Gamma \vdash \Delta}{\forall x.A, \Gamma \vdash \Delta}\,\forall L \qquad \frac{\Gamma \vdash \Delta, A[x \leftarrow y]}{\Gamma \vdash \forall x.A, \Delta}\,\forall R \qquad \frac{A[x \leftarrow y], \Gamma \vdash \Delta}{\exists x.A, \Gamma \vdash \Delta}\,\exists L \qquad \frac{\Gamma \vdash \Delta, A[x \leftarrow t]}{\Gamma \vdash \exists x.A, \Delta}\,\exists R$$

axm: the axiom
cut: the cut rule
ctr: the contraction rules

The quantifier rules have caveats as in FOL.

Sequent calculus: proofs
Proofs are constructed backwards from the *goal* sequent, of the form $\vdash F$, where F is the formula we want to prove.

Inference rules are applied backwards, i.e., given a formula, we find a rule whose consequence matches the formula, and the premises become the new *subgoals*.

Since a rule may have two premises, proving a goal produces a tree of sequents, rooted in the goal, called the *proof tree*.

The proof is completed when (and if!) all branches terminate with an instance of the axiom.

Sequent calculus: proofs

Suppose we want to prove that $\neg A \vee \neg B \Rightarrow \neg(A \wedge B)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A, B \vdash A}\,\text{axm} \quad \dfrac{}{A, B \vdash B}\,\text{axm}}{\neg A, A, B \vdash}\,\neg L \quad \dfrac{}{\neg B, A, B \vdash}\,\neg L}{(\neg A \vee \neg B), A, B \vdash}\,\vee L}{(\neg A \vee \neg B), (A \wedge B) \vdash}\,\wedge L}{(\neg A \vee \neg B) \vdash \neg(A \wedge B)}\,\neg R}{\vdash (\neg A \vee \neg B) \Rightarrow \neg(A \wedge B)}\,\Rightarrow R$$

The root goal is at the bottom.

At the top we have two branches that end with empty formulae by the axiom rule.

The goal has then been proved.

Higher-order logic
In a FOL, variables may range only over individual entities.

In a FOL, we may say "*For all x's such that x is a real number, $x^2 = x \cdot x$*".

We *cannot* say "*For all f's such that f is a function over real numbers, $f^2(x) = f(x) \cdot f(x)$*".

In *higher-order* logics, variables may range over functions and predicates.

In higher-order logics, we can make statements about functions and predicates: e.g., we may say "*if x and y are real numbers and $x = y$, then for all P's such that P is a predicate, $P(x) = P(y)$*".

Computer-assisted theorem proving
Two main aproaches exist to automatic verification of system properties:

- *Theorem proving*: A *theorem prover* is a computer program that implements a formal system. It takes as input a formal definition of the system that must be verified and of the properties that must be proved, and tries to construct a proof by application of inference rules, according to a built-in strategy.
- *Model checking*: A *model checker* is a computer program that extracts a *model* of the system to be verified from its formal description. The model is a graph whose nodes are the states of the system, connected by transitions. The model checker examines each state and checks if the desired properties hold in that state.

Theorem proving may be fully *automatic*, or *interactive*.

ASR: linguaggi di specifica, logica (36)

Computer-assisted theorem proving: PVS

The PVS (*Prototype verification system*) is an interactive theorem prover developed at Computer Science Laboratory, SRI International, Menlo Park (California), by S. Owre, N. Shankar, J. Rushby, and others.

The formal system of PVS consists of a higher-order language and the *sequent calculus* axioms and inference rules.

PVS has many applications, including formal verification of hardware, algorithms, real-time and safety-critical systems. (http://pvs.csl.sri.com)

Computer-assisted theorem proving: PVS

- EMACS-based user interface. **Update**: Visual Studio UI available.
- The user writes definitions and formulae.
- The user selects a formula and enters the *prover* environment.
- Prover commands apply single inference rules or pre-packaged sequences of rules (*strategies*), transforming formulae or producing new formulae.
- The user examines the formulae resulting for each prover command, and decides what to do next.
- The prover finds out when a proof has been successfully completed.

Computer-assisted theorem proving: PVS

- Logical connectives: `NOT`, `AND`, `OR`, `IMPLIES`, ...
- Complex operators: `IF-THEN-ELSE`, `COND`..
- Quantifiers: `EXISTS`, `FORALL`.
- Notation for records, tuples, lists...
- Notation for definitions, abbreviations...
- Rich higher-order type system. Each variable is defined to range over a type, including function and predicate types (predicates are functions that return a Boolean value).
- *Theories*: named collections of definitions and formulae. A theory may be *imported* (and referred to) by another theory.
- A large number of pre-defined theories is available in the *prelude* library.

Computer-assisted theorem proving: PVS types

- Every variable or constant belongs to a type, i.e., denotes elements of a given set.
- *Pre-defined base types*: `bool`, `nat`, `real`...
- *Uninterpreted types*: we just say that a type with a given name exists, e.g., `perfectsw: TYPE`.
- *Interpreted types*: we define a type in terms of other types, or by explicit enumeration of its members.
  - *Enumerations*: `flag: TYPE = {red, black, white, green}`
  - *Tuples*: `triple: TYPE = [nat, flag, real]`
  - *Records*: `point: TYPE = [# x: real, y: real #]`
  - *Subtypes*: `posnat: TYPE = {x: nat | x > 0}`
  - *Functions*: `int2int: TYPE = [int -> int]`

ASR: linguaggi di specifica, logica (40)

Computer-assisted theorem proving: PVS declarations

- *Constants*:
  - `n0: nat` (*uninterpreted constant*)
  - `lucky: nat = 13`
  - `a_triple: triple = (lucky, red, 3.14)`
  - `origin: point = (# x := 0.0, y:= 0.0 #)`
  - `inc: int2int = (lambda (x: int): x + 1)`
  - `inc: [int -> int] = (lambda (x: int): x + 1)`
  - `inc(x: int): int = x + 1`
- *Variables*: add `VAR` to type expression: `m: VAR nat`
- *Formulae*:
  - `plus_commutativity: AXIOM forall(x, y: nat): x + y = y + x`
  - `a_theorem: THEOREM forall(n: nat): n < n + 1`

Keyword `lambda` introduces the parameters of a function.

Instead of `THEOREM` we may use `LEMMA`, `CONJECTURE`...

An `AXIOM` is assumed to be proved.

ASR: linguaggi di specifica, logica (41)

Computer-assisted theorem proving: PVS example theory

```
group : THEORY
BEGIN
  G: TYPE+       % uninterpreted, nonempty
  e: G           % neutral element
  i: [G -> G]    % inverse
  *: [G,G -> G] % binary operation
  x,y,z: VAR G
  associative: AXIOM
    (x*y)*z = x*(y*z)
  id_left: AXIOM
    e*x = x
  inverse_left: AXIOM
    i(x)*x = e
  inverse_associative : THEOREM
    i(x)*(x*y) = y
END group
```

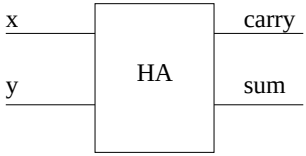
Computer-assisted theorem proving: PVS rules

- *Control* rules to control proof execution and proof tree exploration.
- *Structural* rules to implement the contraction rules and to hide unused formulae in the sequent.
- *Propositional* rules implement the inference rules for connectives, and complex operators, and for the cut. They also apply various simplification laws.
- *Quantifier* rules implement the inference rules for quantifiers.
- *Equality* rules implement various inference rules, including rules for equality, records, tuples, and function definitions.
- *Definition and lemma handling* rules invoke and apply lemmas and definitions.
- *Strategies* apply pre-defined sequences of rules.
- . . . and more.

Computer-assisted theorem proving: PVS verification example

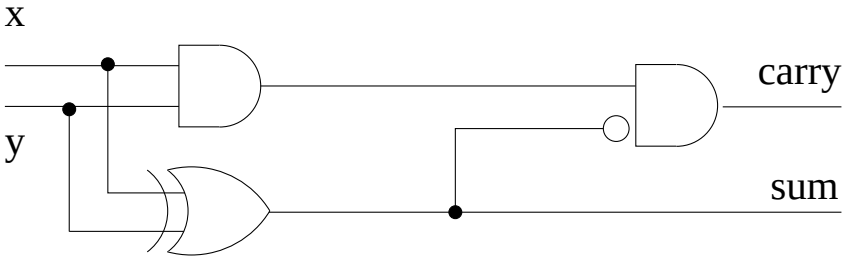
Specification:

x	y	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Is the implementation correct?

An implementation:



Computer-assisted theorem proving: PVS verification example
A half adder is a device whose input is the 1-digit binary encoding of two natural numbers, and whose output is the 2-digit binary encoding of their sum.

A half adder must correctly execute the sums $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, and $1 + 1 = 2$.

In PVS, the property of correctness for the half adder can be expressed as:

```
2*b2n(carry) + b2n(sum) = b2n(x) + b2n(y)
```

where `b2n` is a function that translates a Boolean value into a natural number in $\{0, 1\}$.

ASR: linguaggi di specifica, logica (45)

Computer-assisted theorem proving: PVS verification example

```
HA: THEORY
BEGIN
  x, y: VAR bool

  HA(x,y): [bool, bool] =
     ((x AND y) AND (NOT (x XOR y)),       % carry
      (x XOR y))                           % sum

  % convert Boolean to natural
  b2n(x): nat = IF x THEN 1 ELSE 0 ENDIF

  HA_corr: THEOREM        % correctness
    LET (carry, sum) = HA(x, y) IN
      b2n(sum) + 2*b2n(carry)
        = b2n(x) + b2n(y)
END HA
```

ASR: linguaggi di specifica, logica (46)

Computer-assisted theorem proving: PVS verification example

Initial goal:

```
HA_corr :

  |-------
{1}   FORALL (x, y: bool):
        LET (carry, sum) = HA(x, y) IN
          b2n(sum) + 2 * b2n(carry)
            = b2n(x) + b2n(y)
```

Get rid of quantifiers:

```
Rule? (skolem*)
HA_corr :

  |-------
{1}   LET (carry, sum) = HA(x!1, y!1) IN
        b2n(sum) + 2 * b2n(carry)
            = b2n(x!1) + b2n(y!1)
```

Computer-assisted theorem proving: PVS verification example

Get rid of let-expressions:

```
Rule? (beta)
HA_corr :

  |-------
{1}   b2n(HA(x!1, y!1)`2) + 2 * b2n(HA(x!1, y!1)`1) =
       b2n(x!1) + b2n(y!1)
```

Expand definition of `b2n`:

```
Rule? (expand "b2n")
HA_corr :

  |-------
{1}   IF HA(x!1, y!1)`2 THEN 1 ELSE 0 ENDIF +
       2 * IF HA(x!1, y!1)`1 THEN 1 ELSE 0 ENDIF
       = IF x!1 THEN 1 ELSE 0 ENDIF
           + IF y!1 THEN 1 ELSE 0 ENDIF
```

ASR: linguaggi di specifica, logica (48)

Computer-assisted theorem proving: PVS verification example
After many lift-if's, expand HA:

```
Rule? (expand "HA")
HA_corr :

  |-------
{1}   IF (x!1 XOR y!1)
      THEN IF x!1 THEN IF y!1 THEN FALSE ELSE TRUE ENDIF
                  ELSE IF y!1 THEN TRUE ELSE FALSE ENDIF
                  ENDIF
      ELSE IF (x!1 AND y!1) THEN TRUE
           ELSE IF x!1 THEN FALSE
                ELSE IF y!1 THEN FALSE ELSE TRUE ENDIF
                ENDIF
           ENDIF
      ENDIF
```

Computer-assisted theorem proving: PVS verification example
Boring Boolean algebra:

```
Rule? (prop)
this yields  4 subgoals:
HA_corr.1 :

{-1}  y!1
{-2}  x!1
{-3}  (x!1 XOR y!1)
  |-------
```

The goal branches into four subgoals, `HA_corr.1` through `HA_corr.4`.

ASR: linguaggi di specifica, logica (50)

Grind:

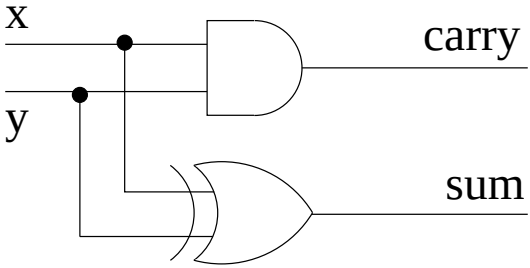
```
Rule? (grind)
Trying repeated skolemization,
instantiation, and if-lifting,

This completes the proof of HA_corr.1.
```

And so on until:

```
Rule? (grind)
Trying repeated skolemization,
instantiation, and if-lifting,

This completes the proof of HA_corr.4.

Q.E.D.
```

Computer-assisted theorem proving: PVS verification example

Another implementation:




```
HA2(x,y) : [bool, bool] =
    ((x AND y),              % carry
     (x XOR y))              % sum
```

Computer-assisted theorem proving: PVS verification example
The correctness theorem has the same form as before, only the half adder function changes:

```
HA2_corr : THEOREM        % correctness
  LET (carry, sum) = HA2(x, y) IN
    bool2nat(sum) + 2*bool2nat(carry)
      = bool2nat(x) + bool2nat(y)
```

We can also prove that the two implementations are equivalent:

```
HA_HA2_equiv: THEOREM     % equivalence
    HA(x, y) = HA2(x, y)
```

Temporal logics

- *Temporal logics* express properties related to the evolution of a system in time or, equivalently, to the states it can reach.
- E.g., we may say that "*The coolant temperature will never exceed a given limit*", or "*if signals A and B are activated, signal C will eventually be activated*".
- Let us assume that the system goes through a sequence of states $s_0, s_1, \ldots s_i \ldots$. (*linear* time).
- To each state we may associate *state variables* representing physical quantities or logical conditions.
- Properties of the system *in a given state* are expressed with formulas of ordinary logic (*propositional logic* or *predicate logic*).
- Temporal logics use operators that relate the truth values of formulas to periods or instants of time, e.g.:
  - $\Box\mathcal{F}$ means that $\mathcal{F}$ will be true from now on.
  - $\Diamond\mathcal{F}$ means that $\mathcal{F}$ will eventually become true.

Logiche temporali: sintassi della *Linear Temporal Logic* LTL
Gli operatori principali sono:

- $\Box$ *henceforth* (o **G**, *globally*), d'ora in poi;
- $\Diamond$ *eventually* (o **F**, *in the future*), prima o poi;

I due operatori sono legati dalle relazioni

$$\Box\mathcal{F} \Leftrightarrow \neg\Diamond\neg\mathcal{F}$$
$$\Diamond\mathcal{F} \Leftrightarrow \neg\Box\neg\mathcal{F}$$

Da questi operatori se ne possono definire altri:

- **U** *until*, finché ($p\mathbf{U}q$ se e solo se esiste k tale che q è vero in s_k e p è vero per tutti gli s_i con $i \leq k$);
- $\circ$ *next*, prossimo stato;
- $\blacksquare$ *always in the past*, sempre in passato;
- $\blacklozenge$ *sometime in the past*, qualche volta in passato.

ASR: linguaggi di specifica, logica (55)

Temporal logics: esempio

proprietà di **sicurezza**:

$$\Box(\mathrm{A.send(m)} \Rightarrow \mathrm{state} = \textbf{connected})$$
$$\Box(\mathrm{B.send\_ack(m)} \Rightarrow \blacklozenge\, \mathrm{B.receive(m)})$$
$$\Box(\mathrm{B.receive(m)} \Rightarrow \blacklozenge\, \mathrm{A.send(m)})$$

- se viene spedito un messaggio *m*, la connessione è attiva;
- se viene mandato un ack per un messaggio, il messaggio è già stato ricevuto;
- se viene ricevuto un messaggio, il messaggio è già stato spedito;

proprietà di **vitalità**:

$$\Box(\mathrm{A.send(m)} \Rightarrow \Diamond \mathrm{B.receive(m)})$$
$$\Box(\Box\Diamond \mathrm{A.send(m)} \Rightarrow \Diamond \mathrm{B.receive(m)})$$
$$\Box(\mathrm{B.receive(m)} \Rightarrow \Diamond \mathrm{B.send\_ack(m)})$$

- se viene spedito un messaggio *m*, prima o poi viene ricevuto;
- se un messaggio *m* viene spedito piú volte, prima o poi viene ricevuto;
- se viene ricevuto un messaggio, prima o poi si restituisce un ack.

Logiche temporali: sintassi della *Computation Tree Logic* CTL

Per modellare evoluzioni alternative (*rami*, *cammini*, *paths*, *tracce* . . .) di un sistema, la CTL introduce gli operatori

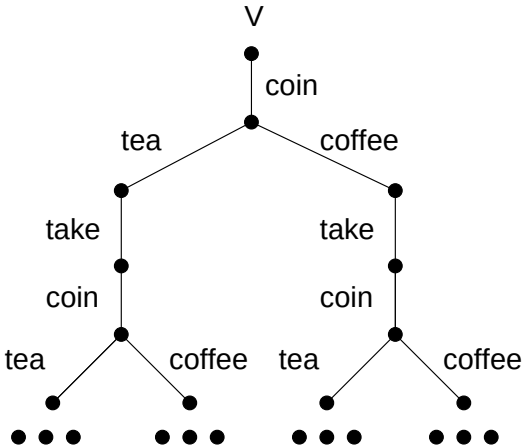
- **A**: lungo tutti i cammini;
- **E**: lungo almeno un cammino.

Questi operatori si combinano con quelli della LTL:

- **AG**(p): per ogni cammino, p è sempre vero;
- **AF**(p): per ogni cammino, p prima o poi è vero;
- **EG**(p): esiste almeno un cammino in cui p è sempre vero;
- **EF**(p): esiste almeno un cammino in cui p prima o poi è vero.
- . . .

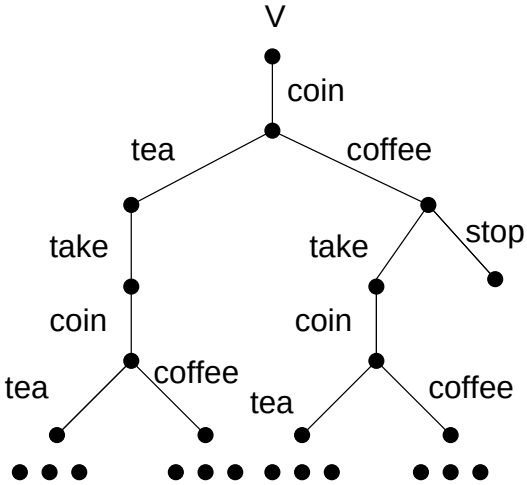
ASR: linguaggi di specifica, logica (57)

Temporal logics semantics: transition systems (1)



A vending machine *V* accepts a coin, then it delivers tea or coffee, then, after the drink has been taken by the customer, returns to the initial state.

Temporal logics semantics: transition systems (2)



A broken vending machine V, that may go out of service if coffee is selected.

Temporal logics: model checking

Model Checking is a common technique used in the automatic verification of temporal logic properties.

Model checking consists in exploring the state space and evaluating temporal logic formulas at each node.

If a formula is not satisfied, the model checker tool can usually produce a *counterexample*, e.g., a path through the state space leading to a state that violates the formula.

ASR: linguaggi di specifica, logica (60)

Temporal logics: Symbolic Analysis Laboratory (SAL)

Il Symbolic Analysis Laboratory è un ambiente di model checking che usa

- un linguaggio per definire sistemi di transizioni:
- un linguaggio per definire formule in logica modale:
  - Linear Temporal Logic (LTL), per grafi di stato lineari;
  - Computational Tree Logic CLTL), per grafi di stato ramificati;

Temporal logics: Symbolic Analysis Laboratory (SAL)

- Il sistema di transizione di un sistema semplice viene definito da un *modulo* in cui si definiscono variabili di ingresso, di uscita e locali,
- lo stato di un modulo è l'insieme dei valori delle variabili,
- per le variabili locali e di uscita sono definiti il *valore attuale*, denotato dal nome della variabile, p.es. X, ed il *valore successivo*, denotato dal nome della variabile con apice, p.es. X',
- le variabili vengono inizializzate nella sezione `INITIALIZATION` del modulo;
- le transizioni sono definite da assegnamenti al valore successivo delle variabli, nella sezione `TRANSITIONS`;
- si ottengono sistemi complessi combinamdo piú moduli per mezzo di operatori di sincronizzazione.

ASR: linguaggi di specifica, logica (62)

Temporal logics: Symbolic Analysis Laboratory (SAL)

```
short: CONTEXT = BEGIN
  State: TYPE = {ready, busy};
  main: MODULE = BEGIN
      INPUT request: boolean            % TRUE se c'e' una richiesta
      OUTPUT state: State
      INITIALIZATION state = ready
      TRANSITION
        state'                          % valore successivo
        IN                              % scelta nondeterministica
        IF (state = ready) AND request
        THEN {busy}
        ELSE {ready, busy}            % insieme di scelte possibili
        ENDIF
    END;
th1: THEOREM main |- G(request => F(state = busy));     % LTL
th2: THEOREM main |- AG(request => AF(state = busy));   % CTL
END
```

A few other formal modeling languages

- *Z* (/zɛd/). Based on predicate logic and Zermelo-Fränkel set theory.
- *Vienna Development Method* (VDM). Well-known predicate logic formalism.
- *Calculus of Communicating Systems* (CCS). A process algebra.
- *Communicating Sequential Processes* (CSP). Another process algebra.
- ...

ASR: Linguaggi di specifica orientati agli oggetti

ASR: Linguaggi di specifica, UML (introduzione) (1)

O-O methods and languages (e.g., UML) are based on *simulation*.

- An *object* represents a real-world entity, concrete (e.g., a motor) or abstract (e.g., a system of equations).
- An object is defined by its *identity*, by the values of the entity *attributes* (e.g., a motor's nameplate data, current speed and torque...) and by the *operations* the entity can execute on request by other entities (e.g., changing the speed...).
- *Links* between objects represent logical relationships between the corresponding entities. E.g. a motor *opens* a valve, a vector *satisfies* a set of equations.
- A *class* is a template description for a set of objects having the same attributes (possibly with different values) and operations.
- An *association* between two classes is a template for the links between the respective objects.

ASR: Linguaggi di specifica, UML (introduzione) (2)

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:

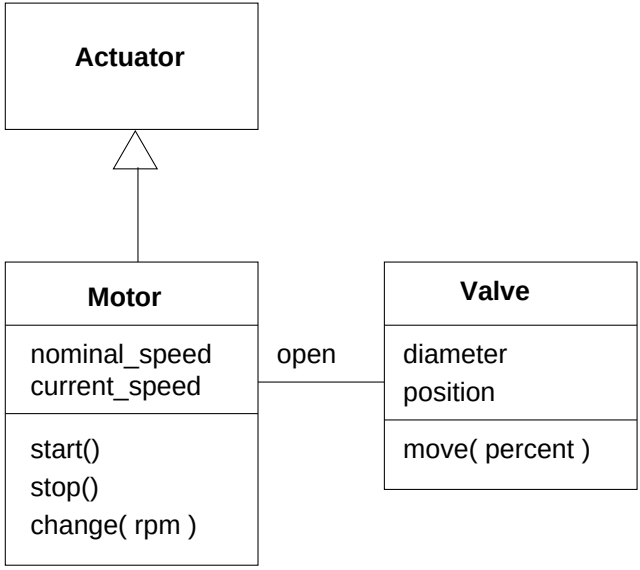
- vista **funzionale**;
- vista **strutturale**;
- vista **dinamica**;
- vista **fisica**,

Per ciascuna vista si usano uno o piú linguaggi grafici.

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

ASR: Linguaggi di specifica, UML (introduzione) (3)

Example: a very simple class diagram (structural view)



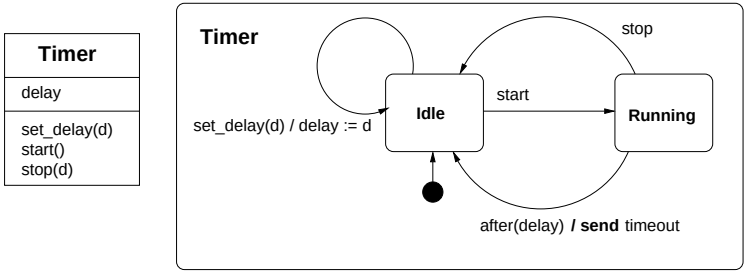
ASR: Linguaggi di specifica, UML (introduzione) (4)

Behavioral modeling (dynamic view)

- *State Machines* describe how an object or (sub)system responds to events. The UML uses a complex state machine language derived from the *Statecharts* formalism.
- *Interactions* describe how objects or (sub)systems interact by exchanging messages.
- *Activities* describe the flow of control and data involved in carrying out a task.

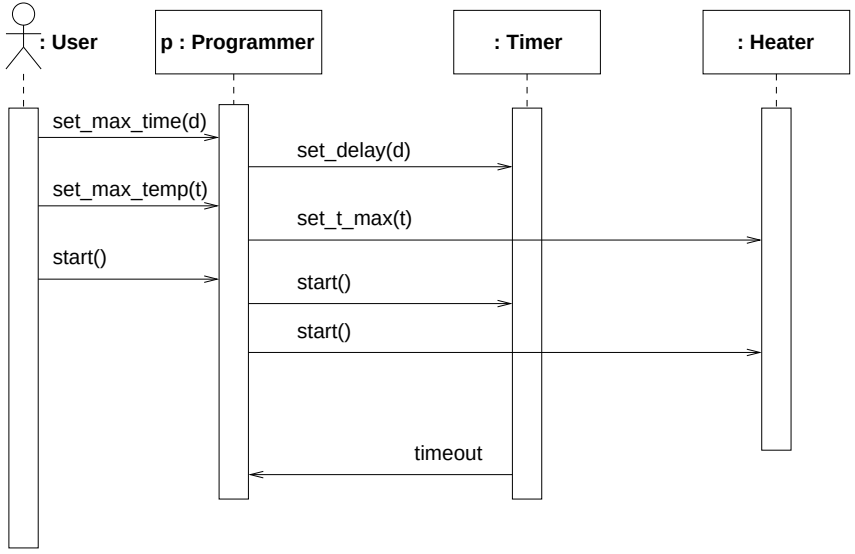
ASR: Linguaggi di specifica, UML (introduzione) (5)

Behavioral modeling: state machines



ASR: Linguaggi di specifica, UML (introduzione) (6)

Behavioral modeling: sequence diagrams



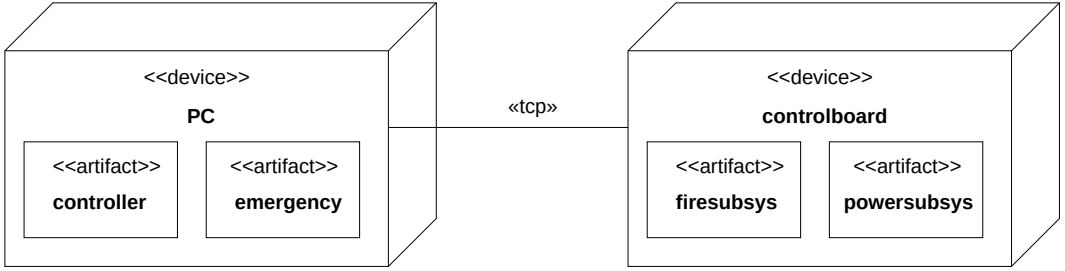
Physical modeling (physical view): deployment diagrams

Deployment diagrams describe the physical structure of a system:

- *Nodes* represent hardware devices performing computations (typically, whole computer, or lower-level parts if necessary).
- *Artifacts* represent files (containing programs or data).
- *Associations* represent communication paths between nodes.

ASR: Linguaggi di specifica, UML (introduzione) (8)

Modello fisico: diagramma di deployment



ASR: Linguaggi di specifica, UML (introduzione) (9)

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 varii 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

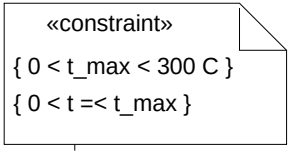
I meccanismi di estensione permettono di adattare il linguaggio ad esigenze particolari aggiungendo informazioni a elementi di modello o introducendo nuovi elementi partendo da elementi preesistemti.

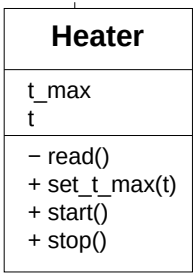
- **vincoli**: applicabili a qualsiasi elemento, in linguaggio
  - naturale;
  - logico/matematico;
  - **OCL** (Object Constraint Language), FOL adattata a UML;
- **stereotipi**: estensioni di elementi di modello;
- **valori etichettati**i: proprietà di stereotipi, della forma {*nome* = *valore*}.

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

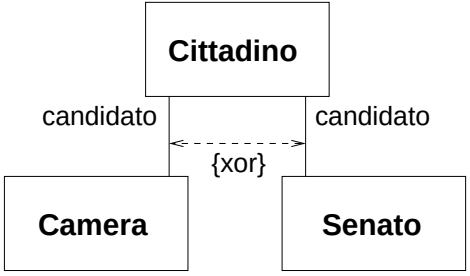
ASR: Linguaggi di specifica, UML (introduzione) (12)

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:

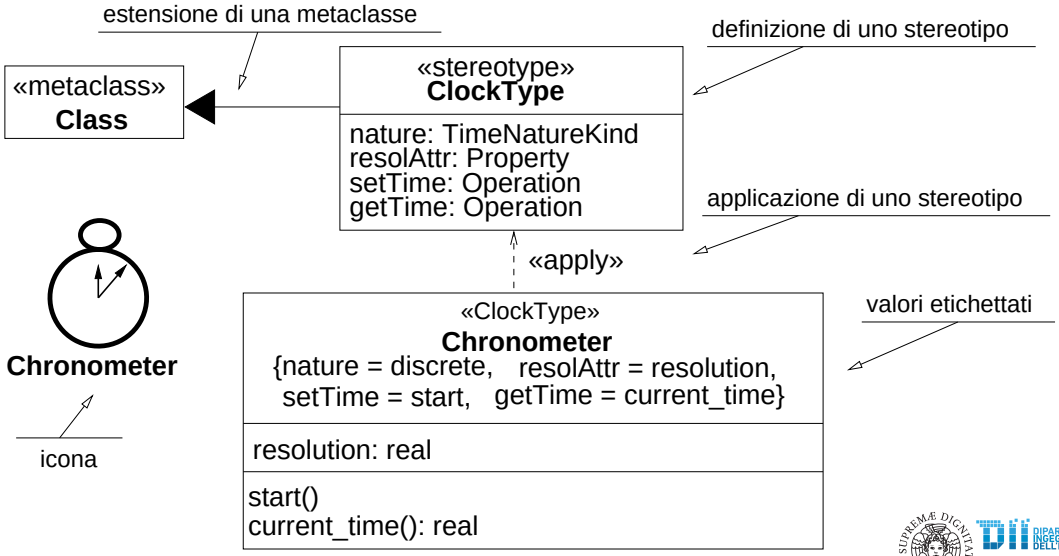
- (time) nature: discreto o denso;
- resolAttr: risoluzione temporale;
- setTime, getTime: operazioni per inizializzare e leggere la misura del tempo.

Uno sviluppatore *applica* lo stereotipo creandone un'istanza (**Chronometer**) definita dai seguenti valori etichettati:

- `{nature = discrete}`;
- `{resolAttr = resolution}`;
- `{setTime = start}`;
- `{getTime = current_time}`;

ASR: Linguaggi di specifica, UML (introduzione) (15)

Meccanismi di estensione: stereotipi e valori etichettati



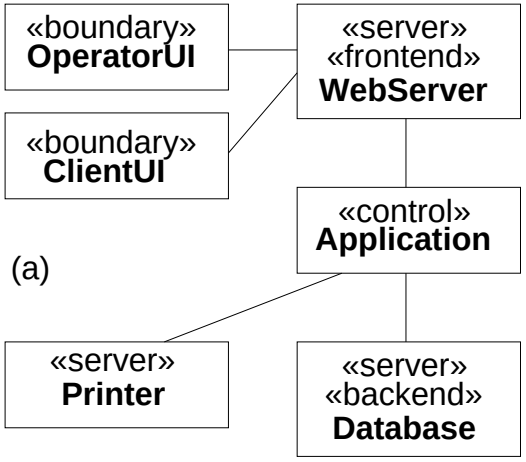
Meccanismi di estensione: stereotipi e valori etichettati

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)

«versioning»
{version = 3.1.4}
{date = 2020−04−01}
{author = Random J. Coder}
Accounting

(b)

«implementation»
{language = C++}
Solver

(c)

Meccanismi di estensione: profili

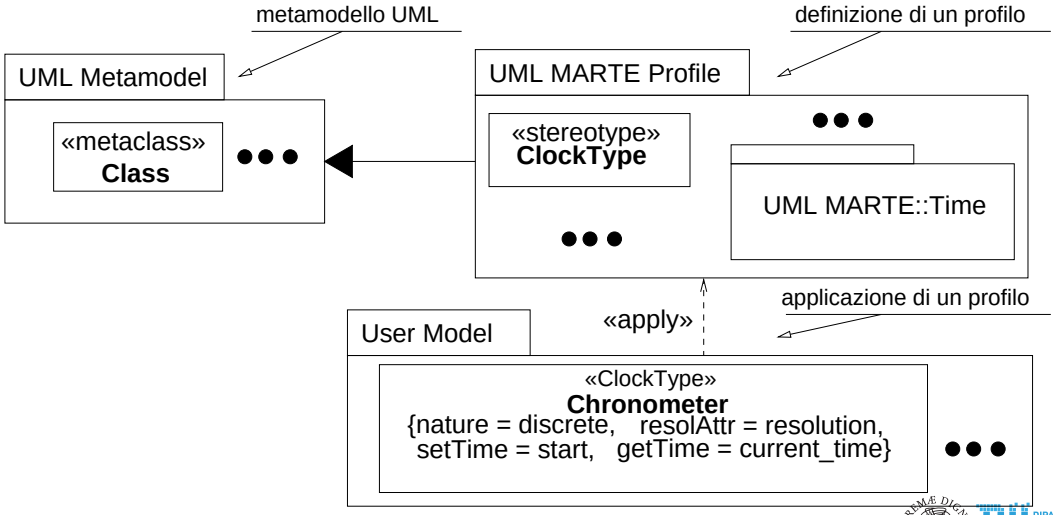
Un profilo è un insieme organico di stereotipi, che definisce nuovi elementi di modello utili in particolari campi di applicazione.

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/

Meccanismi di estensione: profili



ASR: Linguaggi di specifica, UML (diagrammi di classi) (1)

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.

ASR: Linguaggi di specifica, UML (diagrammi di classi) (2)

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.

ASR: Linguaggi di specifica, UML (diagrammi di classi) (3)

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)

ASR: Linguaggi di specifica, UML (diagrammi di classi) (4)

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* 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>
```

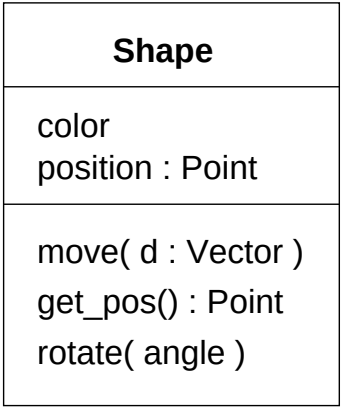
ASR: Linguaggi di specifica, UML (diagrammi di classi) (5)

Oggetti

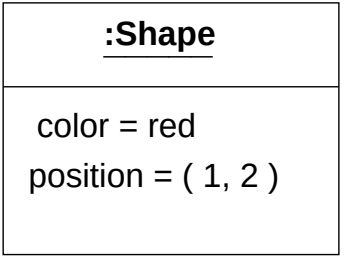
Si scrivono i nomi dell'oggetto e della classe d'appartenenza, sottolineati e separati dal carattere ':', ed opzionalmente gli attributi con i rispettivi valori.

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.



classe



oggetto

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.

Associazioni: molteplicità e ruoli
La fig. (a) del lucido 202 mostra due associazioni.

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**).

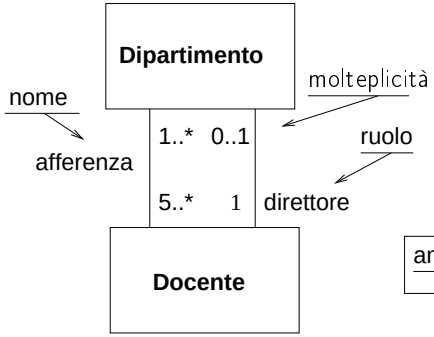
N.B.: Il diagramma non specifica se il direttore deve afferire al dipartimento!

Associazioni: molteplicità e ruoli
La fig. (b) del lucido 202 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.

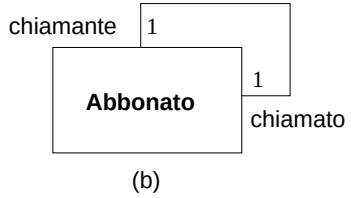
La fig. (c) mostra un'istanza (link) dell'associazione.

ASR: Linguaggi di specifica, UML (diagrammi di classi) (9)

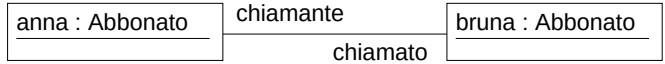
Associazioni: molteplicità e ruoli



(a)



(b)



(c)

Special associations and relationships

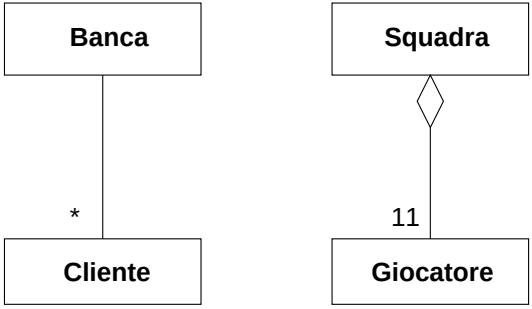
- *Aggregation*: The (weak) association of a compound entity with its components, when the components may exist outside the compound entity (e.g. a team and its players, a library and its books).
- *Composition*: The (strong) association of a compound entity with its components, when the components may not exist outside the compound entity (e.g. a motor and its parts).
- *Generalization*: A *relationship* (not an association) specifying that a class is a subset of another one (e.g., motors are a subclass of actuators).

ASR: Linguaggi di specifica, UML (diagrammi di classi) (11)

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.

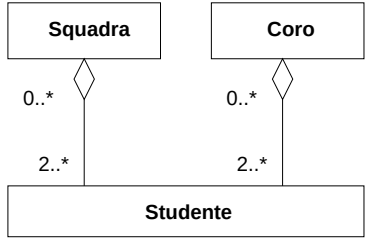
Questo concetto non ha una definizione precisa, quindi l'uso di questa annotazione è facoltativo. La figura seguente può dare un'idea della differenza fra un'associazione annotata come aggregazione (con una losanga) ed una non annotata.

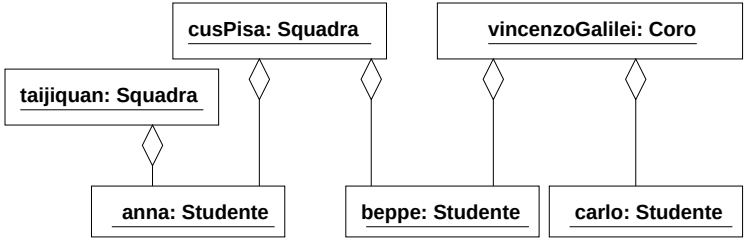


ASR: Linguaggi di specifica, UML (diagrammi di classi) (12)

Aggregazione

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

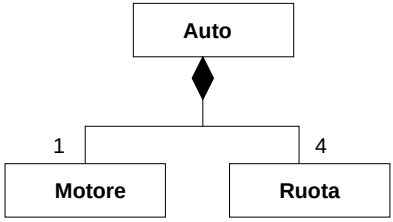


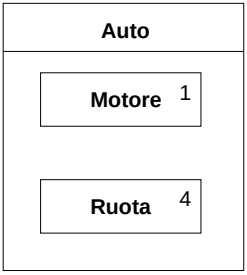


Composizione

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

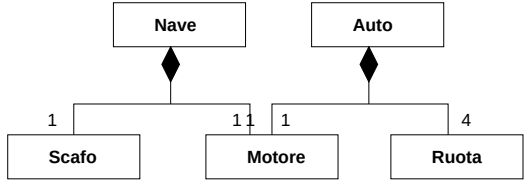
In certi casi, la relazione di composizione può implicare che i componenti non possano esistere al di fuori del composto.





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*.

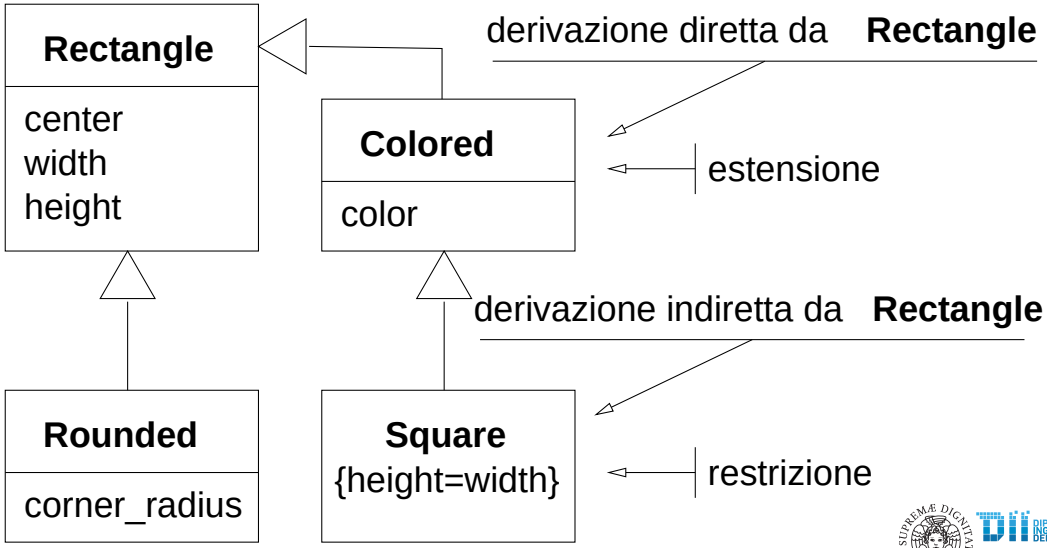
Una classe derivata generalmente è un'*estensione* della classe base, poichè *aggiunge* proprietà alla stessa.

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.

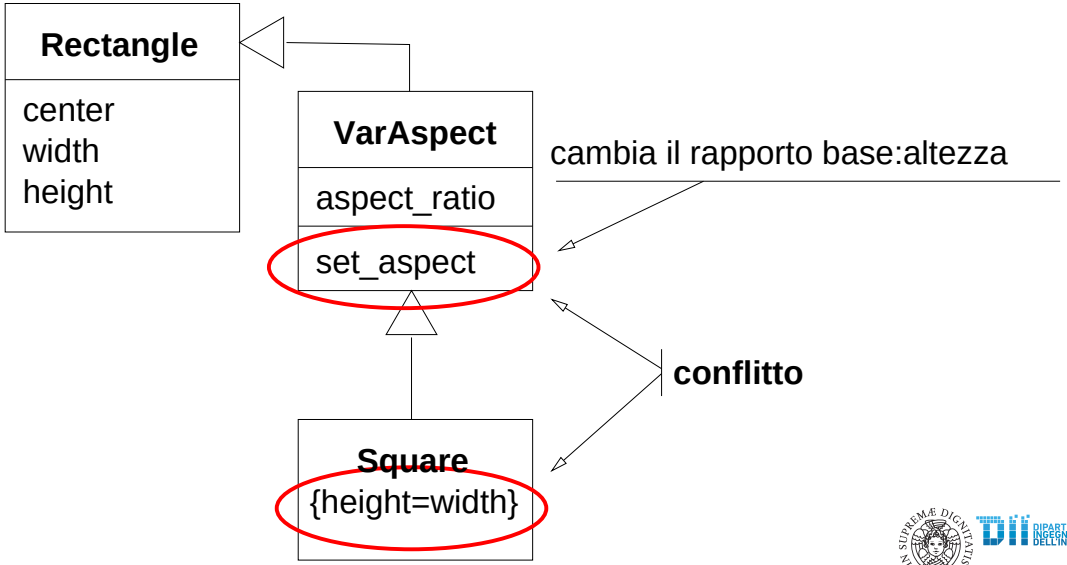
ASR: Linguaggi di specifica, UML (diagrammi di classi) (17)

Generalizzazione: derivazione



ASR: Linguaggi di specifica, UML (diagrammi di classi) (18)

Generalizzazione: princ. di sostituzione

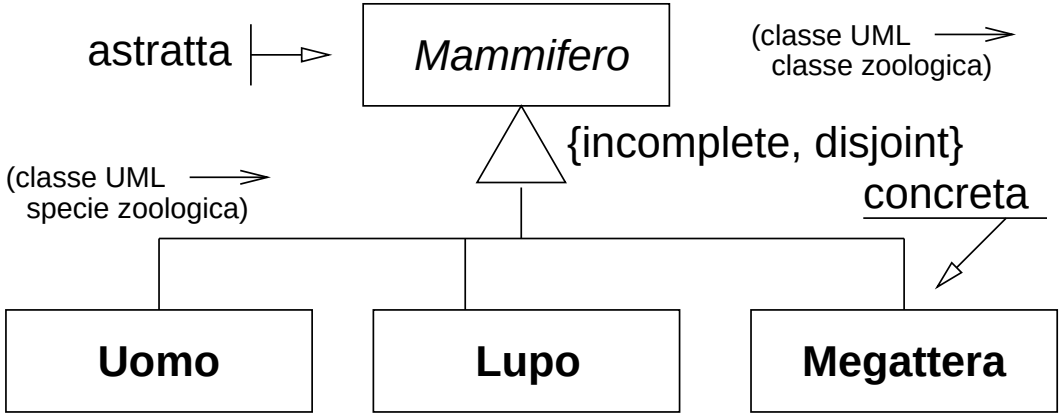


Generalizzazione: classi astratte e concrete

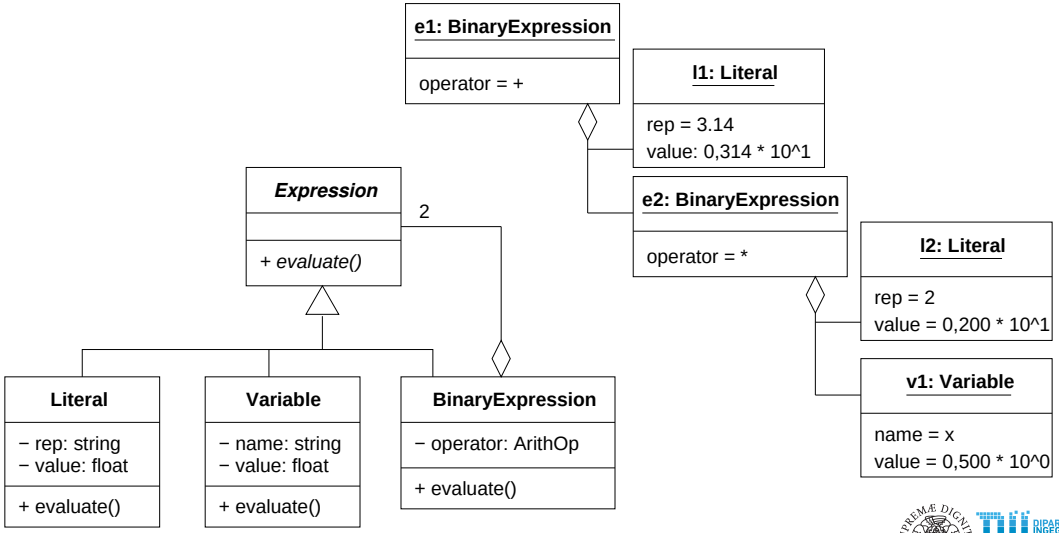
Una classe che ha istanze **dirette** si dice *concreta*.

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

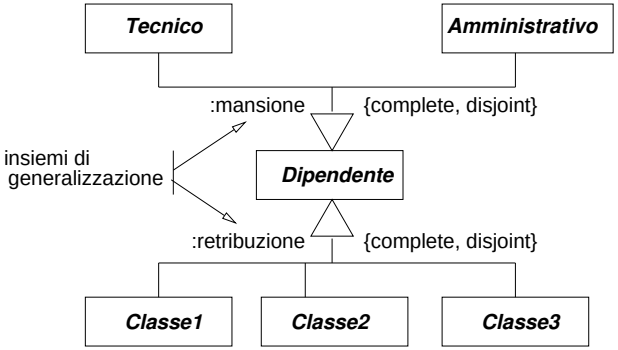
Generalizzazione: classi astratte e concrete



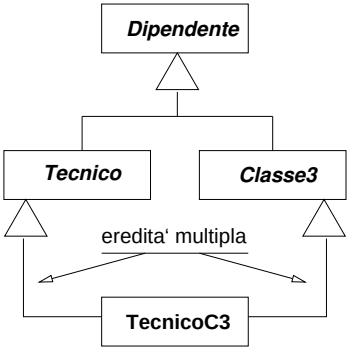
Generalizzazione: aggregazione ricorsiva



Generalizzazione: insiemi di generalizzazioni

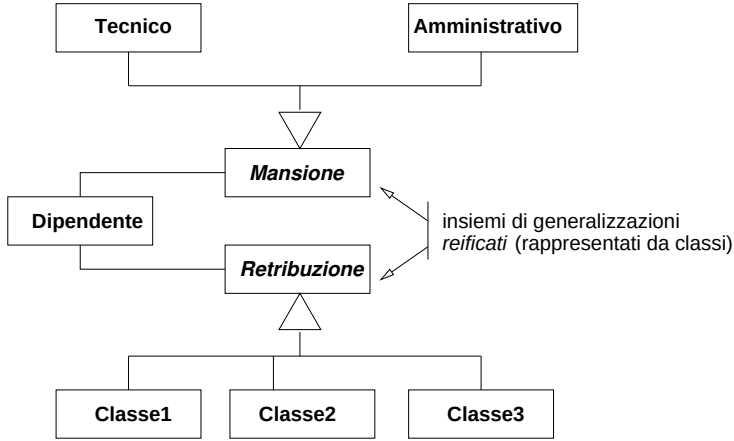


(a)

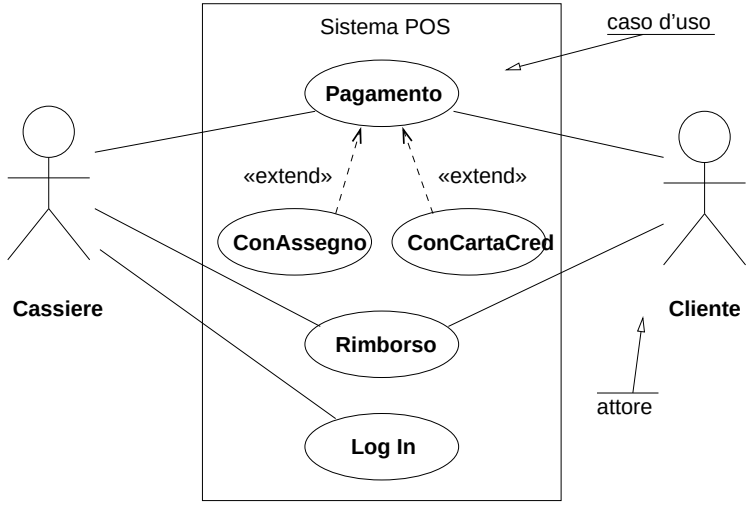


(b)

Generalizzazione: insiemi di generalizzazioni



ASR: Linguaggi di specifica, UML (diagrammi dei casi d'uso)



ASR: Linguaggi di specifica, UML (Behavioral Modeling)

- *State Machines* describe how an object or (sub)system responds to events. The UML uses a complex state machine language derived from the *Statecharts* formalism.
- *Interactions* describe how objects or (sub)systems interact by exchanging messages.
- *Activities* describe the flow of control and data involved in carrying out a task.

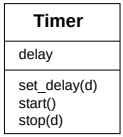
ASR: Linguaggi di specifica, UML (diagrammi di stato) (1)

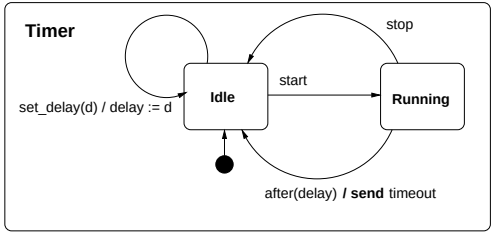
- Estensione degli automi a stati finiti;
- basati sugli **Statecharts** (David Harel);
- associati a una classe, o ad una *collaborazione* (interazione di un insieme di oggetti), o a un metodo.

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.





ASR: Linguaggi di specifica, UML (diagrammi di stato) (2)

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.

ASR: Linguaggi di specifica, UML (diagrammi di stato) (3)

Eventi

- Ricezione di *chiamate di operazione*;
- ricezione di *segnali*;
- *cambiamenti* di condizioni logiche, p.es., **when** ($p > \mathrm{P}$);
- 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

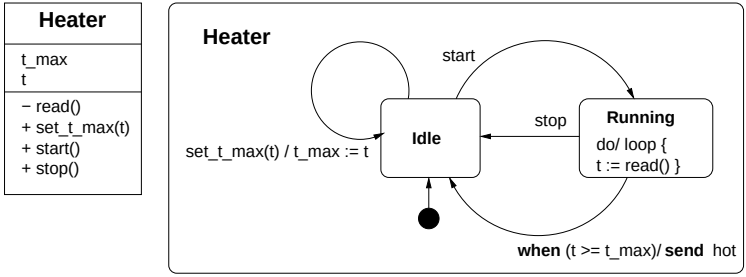
I segnali sono delle entità che gli oggetti si possono scambiare per comunicare fra di loro, e possono essere strutturati in una gerarchia di generalizzazione: per esempio, un ipotetico segnale **Input** può essere descritto come generalizzazione dei segnali **Mouse** e **Keyboard**, che a loro volta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

ASR: Linguaggi di specifica, UML (diagrammi di stato) (5)

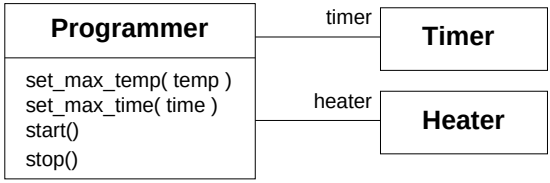
Eventi di cambiamento



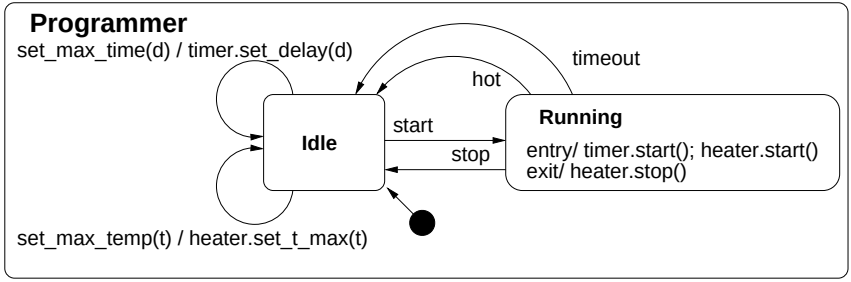
when(. . .): fire transition when a condition becomes true (*change event*)
do/: perform an *activity* while in a given state.

ASR: Linguaggi di specifica, UML (diagrammi di stato) (6)

Azioni di entry ed exit

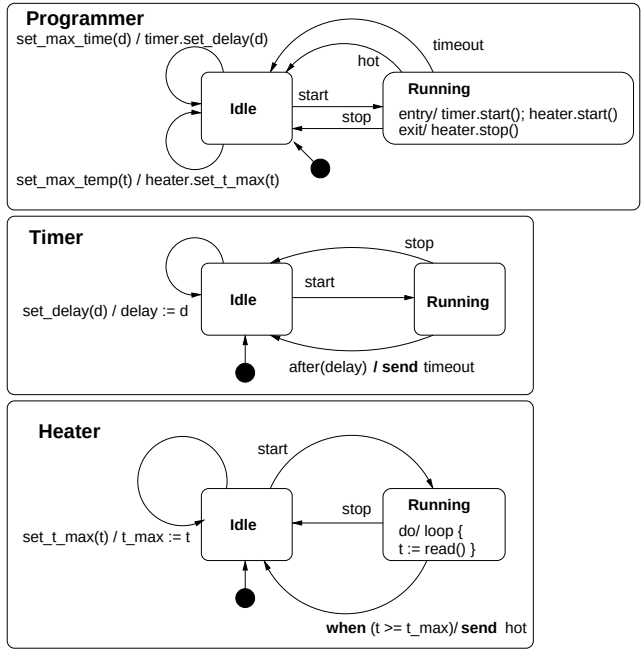


the top−level events are calls to the **Timer** interface and reception of the timeout and hot signals

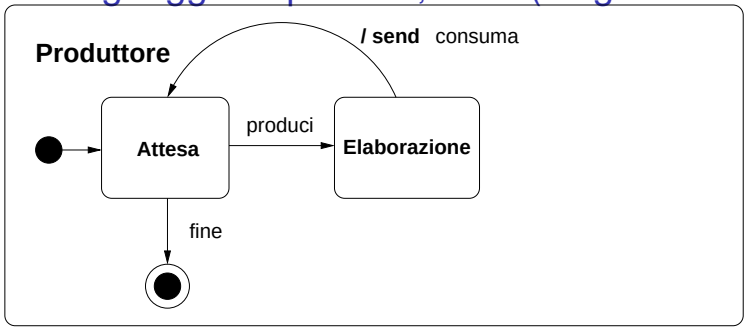


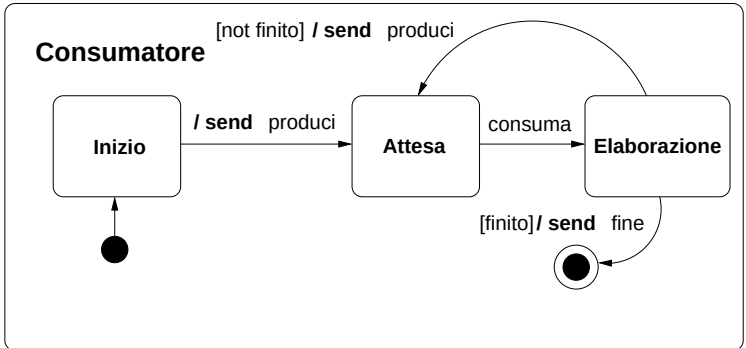
heater, timer: *rolenames* to identify participants in an association
entry/: action(s) performed when entering a state
exit/: action(s) performed when leaving a state

ASR: Linguaggi di specifica, UML (diagrammi di stato) (7)



ASR: Linguaggi di specifica, UML (diagrammi di stato) (8)





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.

Gestione degli eventi

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

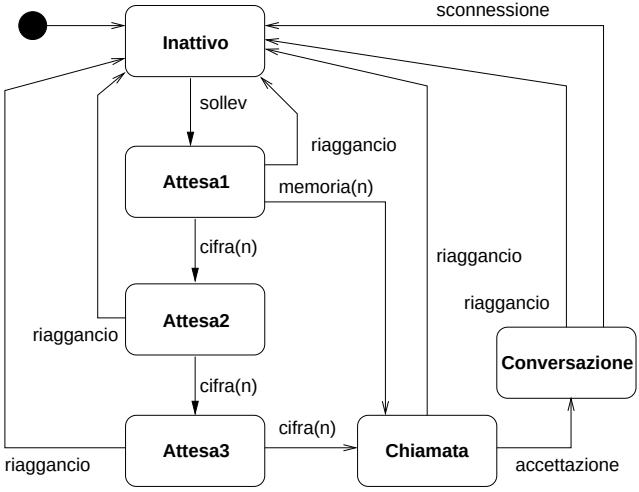
In una macchina a stati *gerarchica* il comportamento del sistema in un dato stato (*superstato*) può essere specificato da un insieme di *sottostati*.

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.

ASR: Linguaggi di specifica, UML (diagrammi di stato) (13)

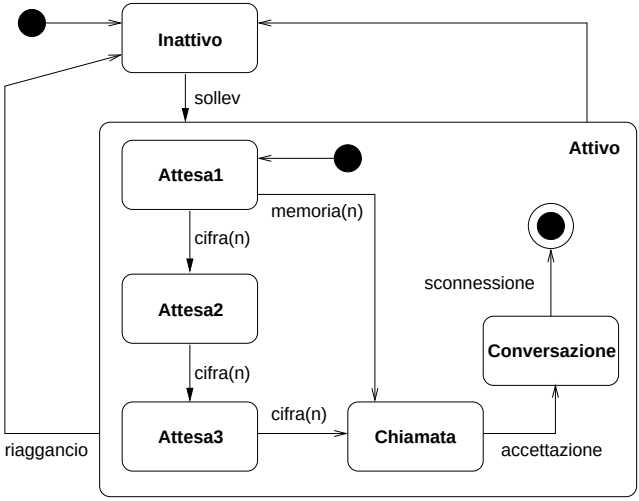
macchine a stati gerarchiche



macchina a stati “piatta” (non gerarchica)

ASR: Linguaggi di specifica, UML (diagrammi di stato) (14)

macchine a stati gerarchiche



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

macchine a stati gerarchiche

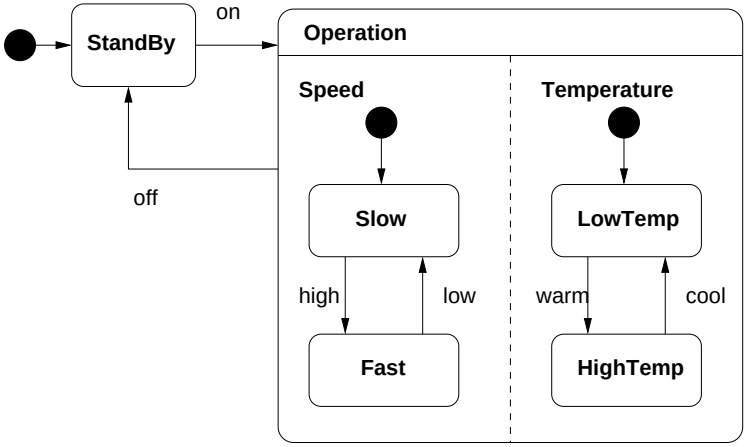
La transizione in ingresso al superstato **Attivo** porta la macchina nel sottostato iniziale (**Attesa1**) di quest'ultimo.

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**.

ASR: Linguaggi di specifica, UML (diagrammi di stato) (16)

Regioni concorrenti



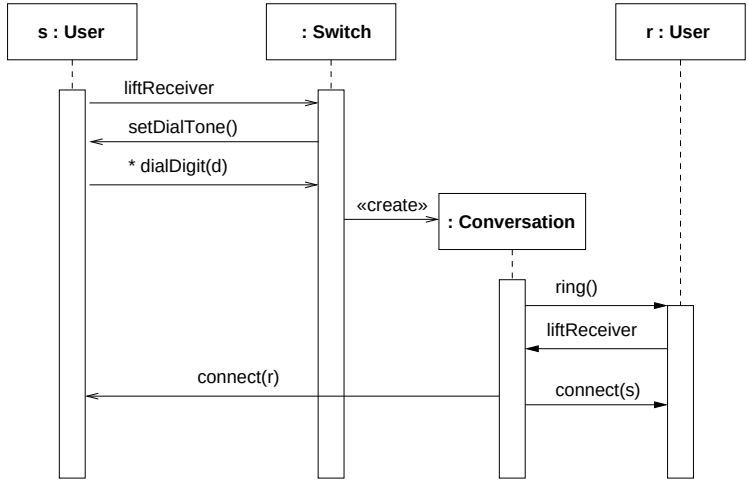
i sottosistemi per il controllo della velocità e della temperatura si evolvono in modo concorrente

ASR: Linguaggi di specifica, UML (diagrammi di sequenza) (1)

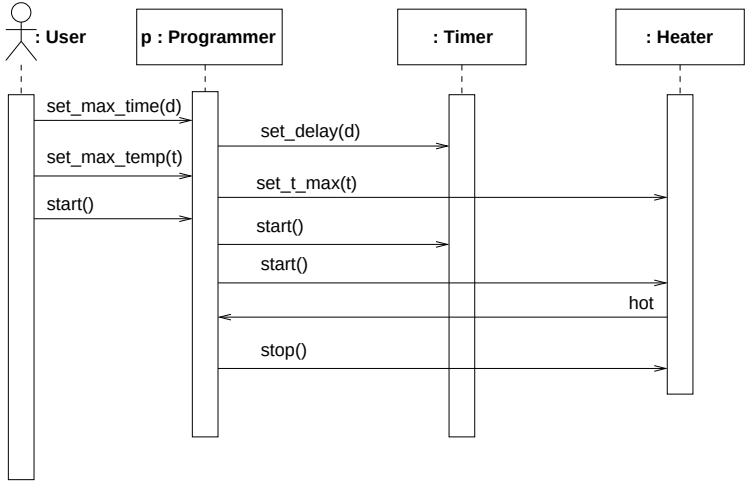
Un diagramma di sequenza descrive l'interazione fra piú oggetti mettendo in evidenza il flusso di messaggi scambiati e la loro successione temporale. I diagrammi di sequenza sono quindi adatti a rappresentare degli *scenari* possibili nell'evoluzione di un insieme di oggetti.

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.

ASR: Linguaggi di specifica, UML (diagrammi di sequenza) (2)

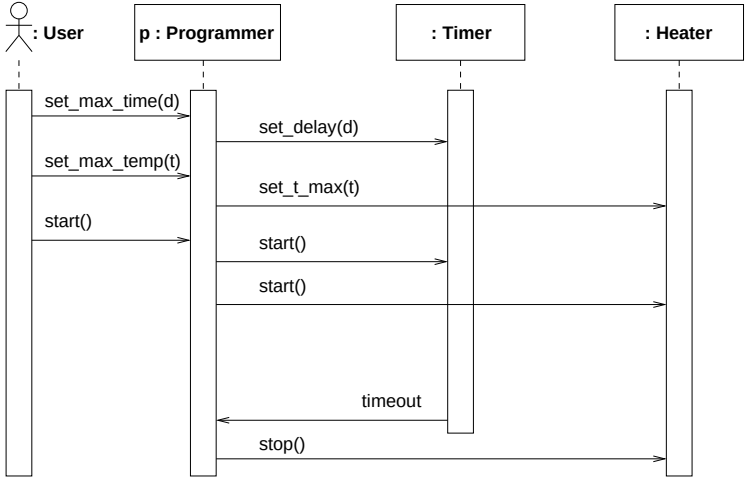


ASR: Linguaggi di specifica, UML (diagrammi di sequenza) (3)



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

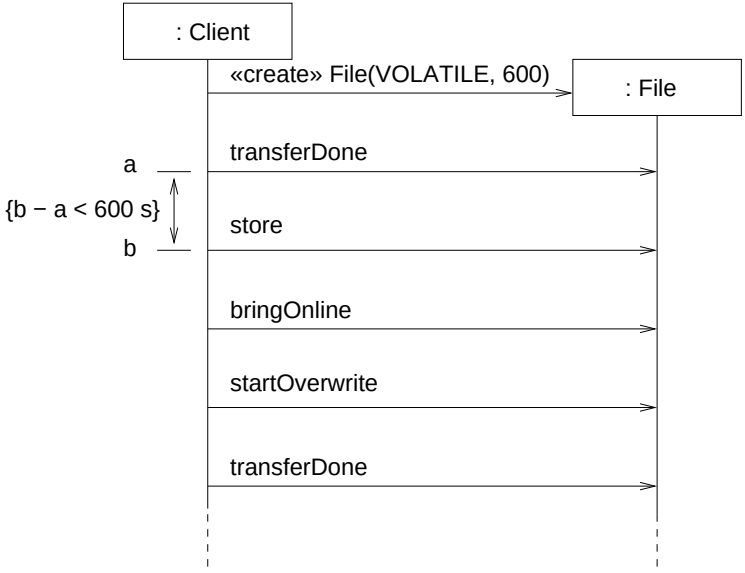
ASR: Linguaggi di specifica, UML (diagrammi di sequenza) (4)



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

ASR: Linguaggi di specifica, UML (diagrammi di sequenza) (5)

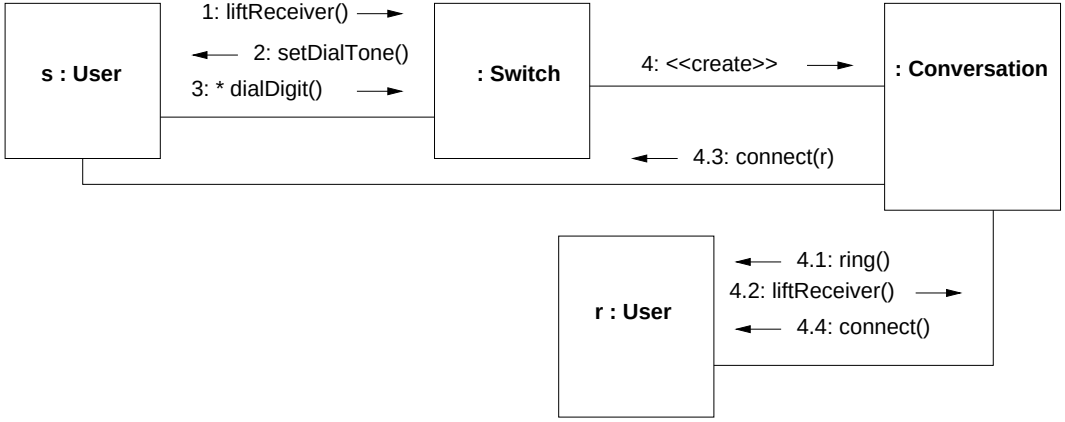
Vincoli temporali



ASR: Linguaggi di specifica, UML (diagrammi di comunicazione) (1)

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.

ASR: Linguaggi di specifica, UML (diagrammi di comunicazione) (2)



ASR: Linguaggi di specifica, UML (diagrammi di attività) (1)

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*).

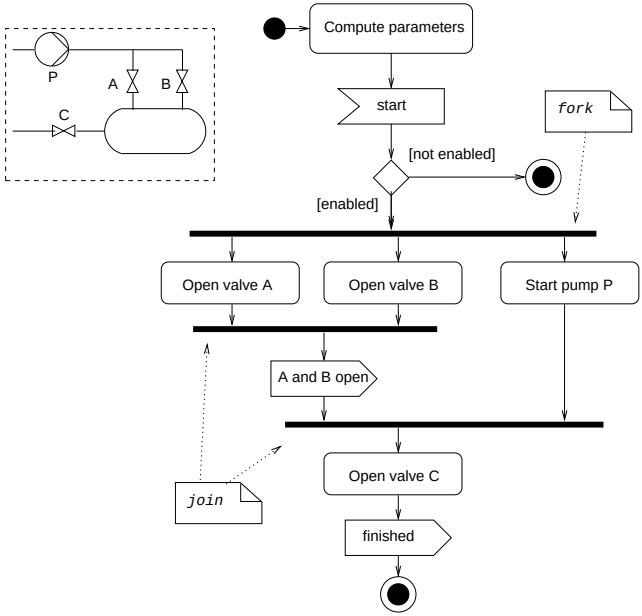
ASR: Linguaggi di specifica, UML (diagrammi di attività) (2)

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.

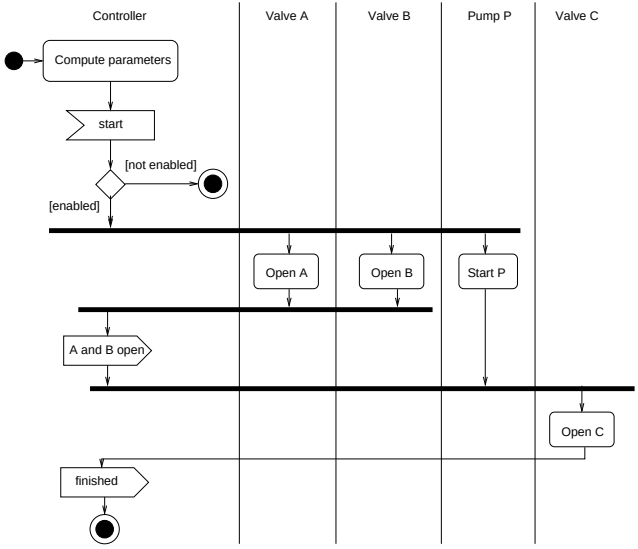
I diagrammi di attività possono descrivere attività svolte da entità differenti, raggruppandole graficamente. Ciascuno dei gruppi cosí ottenuti è una *partizione*, detta anche *corsia* (*swimlane*).

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.

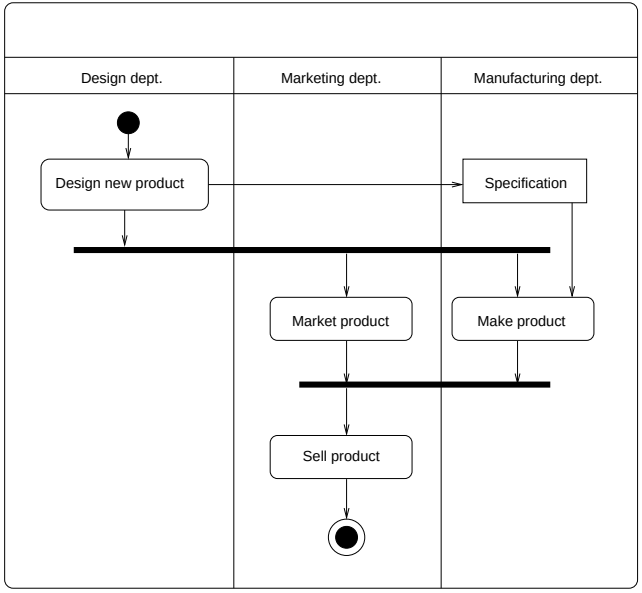
ASR: Linguaggi di specifica, UML (diagrammi di attività) (3)



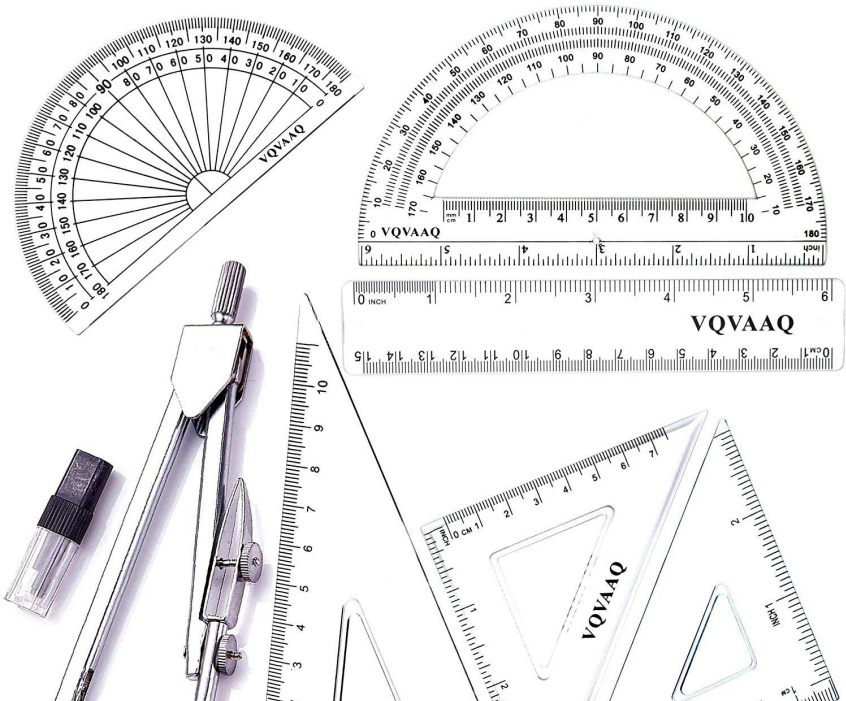
ASR: Linguaggi di specifica, UML (diagrammi di attività) (4)



ASR: Linguaggi di specifica, UML (diagrammi di attività) (5)



PROGETTO



Indice

Il progetto: obiettivi

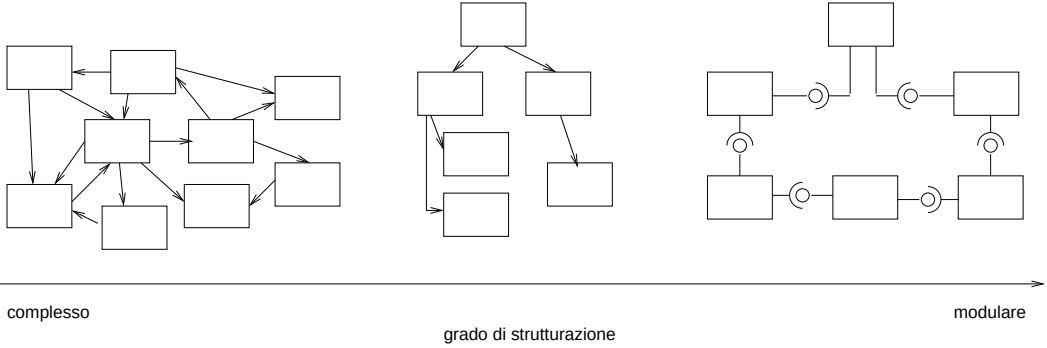
- **architettura logica del sw**
  - **moduli**:
    - **sottosistemi**;
    - **componenti**;
    - **moduli unitari**.
  - **relazioni**:
    - **associazioni**;
    - **generalizzazione**;
    - **dipendenze**;
    - **collaborazioni**.
- **architettura fisica**
  - **del sw**:
    - **file eseguibili**;
    - **librerie**;
    - **dati**;
    - ....
  - **dello hw**:
    - **nodi computazionali**;
    - **ambienti di esecuzione**.

Il progetto: proprietà

-
  - **correttezza funzionale** del prodotto;
  - **rispetto dei req, non funzionali** del prodotto (tradotti, se possibile, in req, 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.

Il progetto: 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**.



Il progetto: moduli (1)

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.

Il progetto: moduli (2)

- interfaccia
  - offerta (*provided*)
  - 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.

Il progetto: moduli (3)

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.

Il progetto: moduli (4)

Ripartizione delle responsabilità

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.

Il progetto: moduli (5)

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.

Il progetto: moduli (6)

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**.

Il progetto: moduli (7)

Il criterio base nel progetto dei moduli è quindi

- **ottenere il massimo disaccoppiamento fra i moduli.**

David L. Parnas. **On the Criteria To Be Used in Decomposing Systems into Modules**, Communications of the ACM, vol.15, n. 12, 1972.
http://www.ing.unipi.it/ a009435/issw/extra/criteria.pdf

Il progetto: moduli (8)

Incapsulamento

Incapsulare significa, approssimativamente, raggruppare alcuni elementi isolandoli dal resto del sistema e definire l'interfaccia di tale gruppo, nascondendo 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*.

Linguaggi di progetto: UML (1)

Incapsulamento: 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.

Linguaggi di progetto: UML (2)

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.

Linguaggi di progetto: UML (3)

Incapsulamento: elementi ≪interface≫

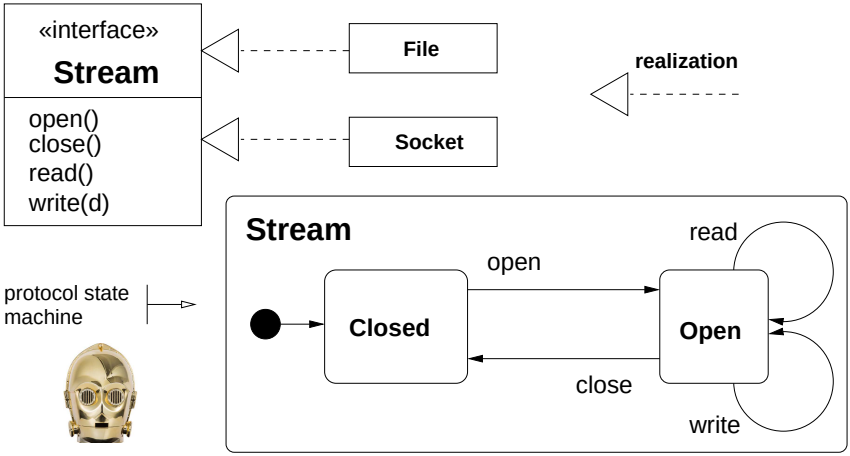
Un'interfaccia si può rappresentare **separatamente** dal modulo che la implementa, usando l'elemento ≪interface≫.

Un'interfaccia può avere piú **implementazioni alternative**.

Linguaggi di progetto: UML (4)

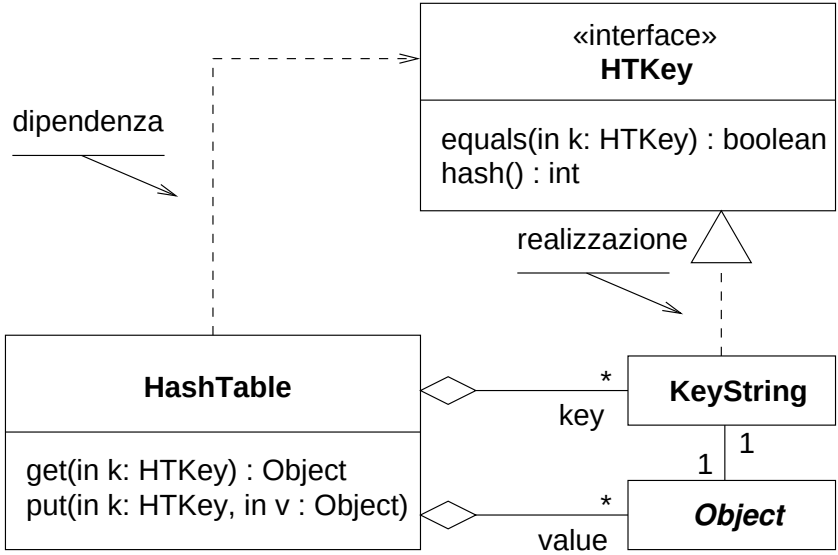
Incapsulamento: elementi ≪interface≫

L'elemento ≪interface≫ è un elenco di operazioni con visibilità pubblica, a cui si possono aggiungere dei vincoli e un protocollo.



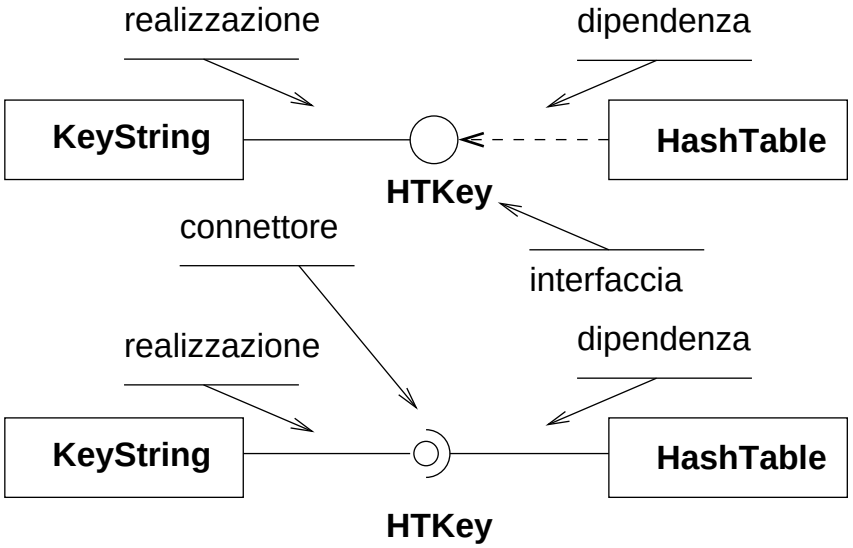
Linguaggi di progetto: UML (5)

Incapsulamento: elementi ≪interface≫



Linguaggi di progetto: UML (6)

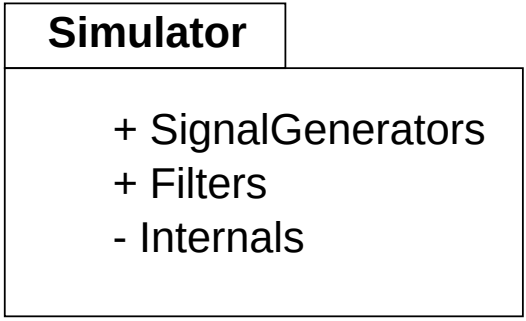
Incapsulamento: elementi ≪interface≫



Linguaggi di progetto: UML (7)

Incapsulamento: package

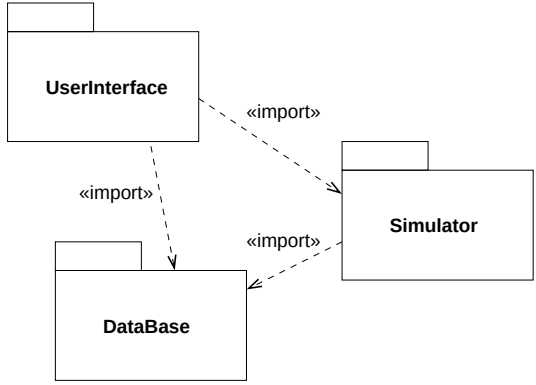
- Un package serve a organizzare **elementi di modello**;
- può non avere una corrispondenza diretta con la struttura dell'architettura software;
- quando viene usato per modellare un (sotto)sistema software, si chiama "*interfaccia*" l'insieme dei suoi elementi visibili:



Linguaggi di progetto: UML (8)

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++).

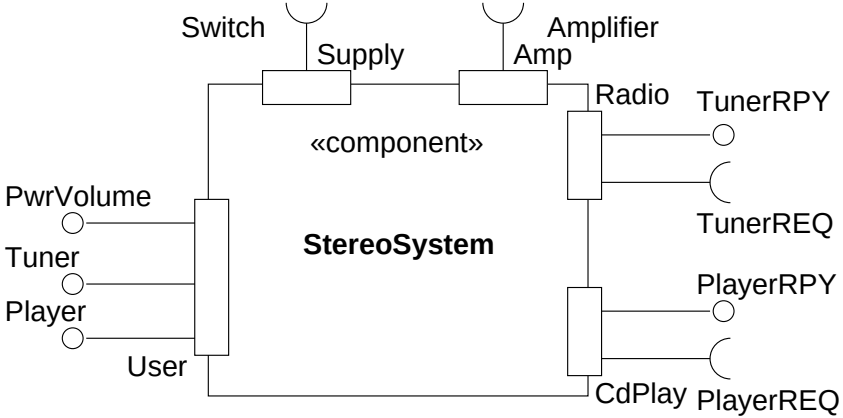


Linguaggi di progetto: UML (9)

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*).

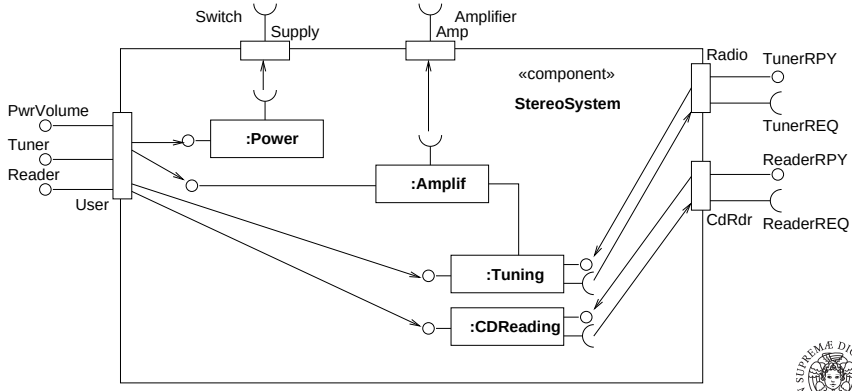


Linguaggi di progetto: UML (10)

Incapsulamento: componenti

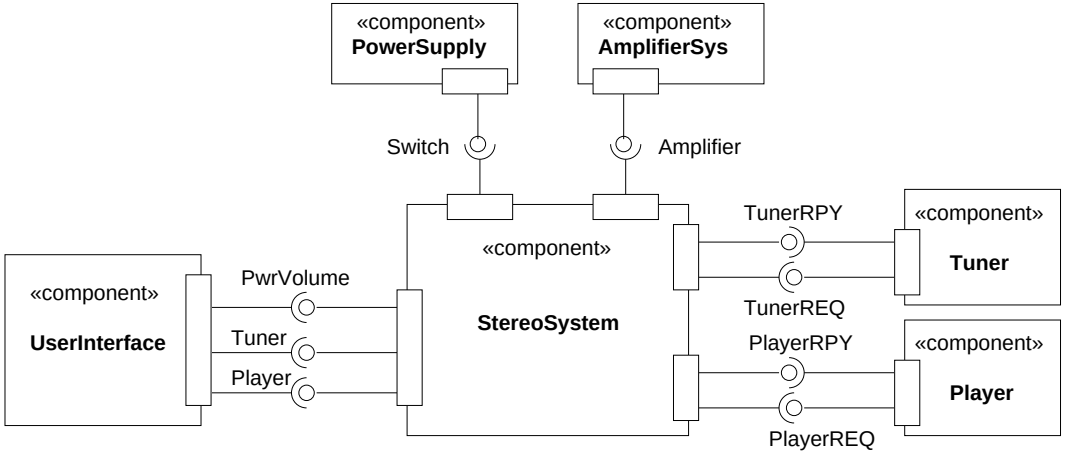
Un componente può essere realizzato *direttamente* da un'istanza di una classe, o *indirettamente* da istanze di piú classi o componenti.

La relazione di *delega* mostra la provenienza o la destinazione delle comunicazioni (chiamate di operazioni e trasmissioni di eventi) passanti attraverso i port.



Linguaggi di progetto: UML (11)

Incapsulamento: componenti



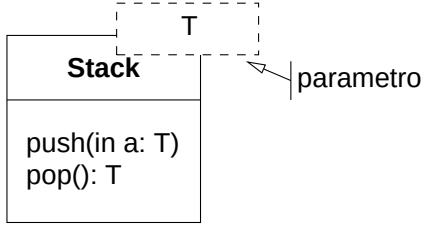
Linguaggi di progetto: UML (12)

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*.

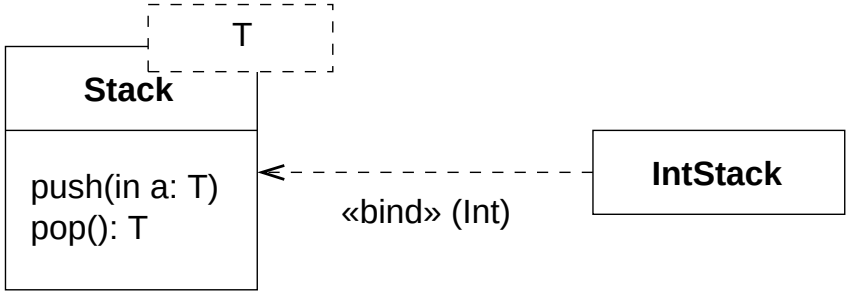
Gli elementi di modello che possono essere parametrizzati sono *classi*, *package*, *componenti*, *operazioni*, *collaborazioni* (gruppi di istanze interagenti) etc. (Quindi il concetto di genericità non si applica soltanto ai moduli).



Linguaggi di progetto: UML (13)

Moduli generici

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

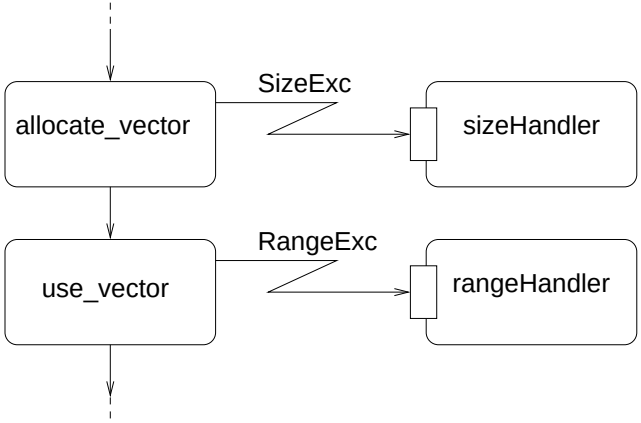


Stack<T ⟶ int>

Linguaggi di progetto: UML (14)

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.



Il progetto orientato agli oggetti

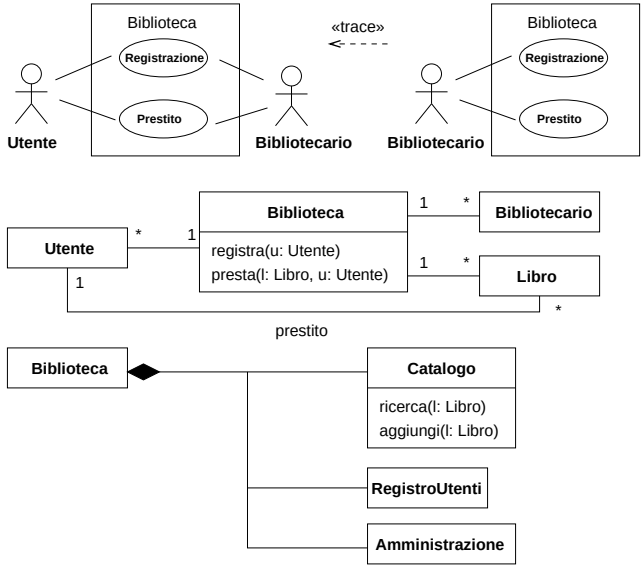
Il modello di analisi è il punto di partenza per definire l'architettura software, la cui struttura, negli stadi iniziali della progettazione, ricalca quella del modello di analisi.

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.

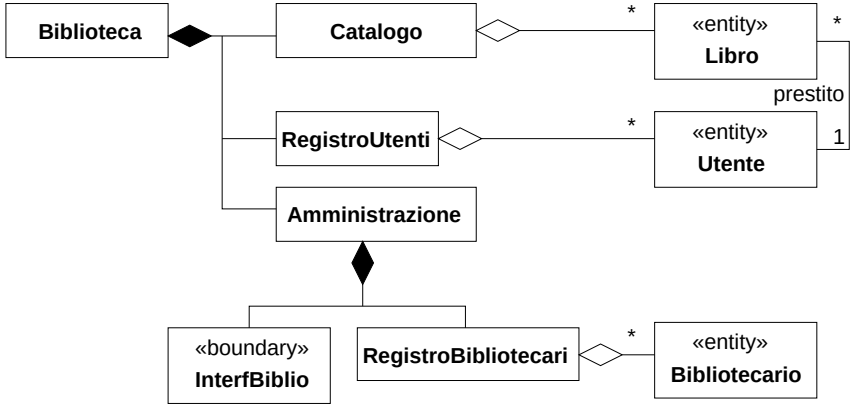
Il progetto OO: esempio (1)

modello di analisi



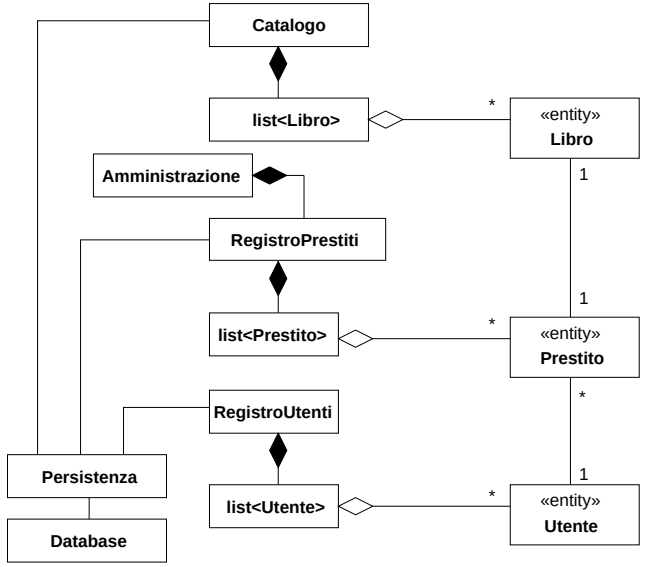
Il progetto OO: esempio (2)

modello parziale di progetto



Il progetto OO: esempio (3)

modello parziale di progetto



Il progetto OO: eredità e polimorfismo (1)

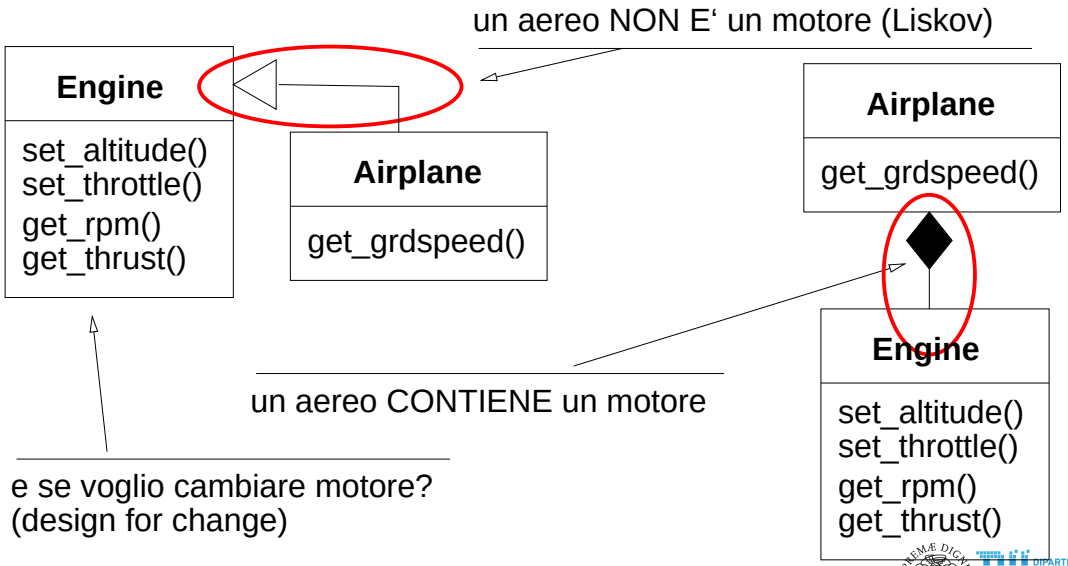
Nei linguaggi di programmazione, l'eredità è il meccanismo che inserisce gli attributi e operazioni di una classe base nelle classi derivate. Questo meccanismo si può sfruttare per tre scopi:

- 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();
};
```

Il progetto OO: eredità e polimorfismo (2)



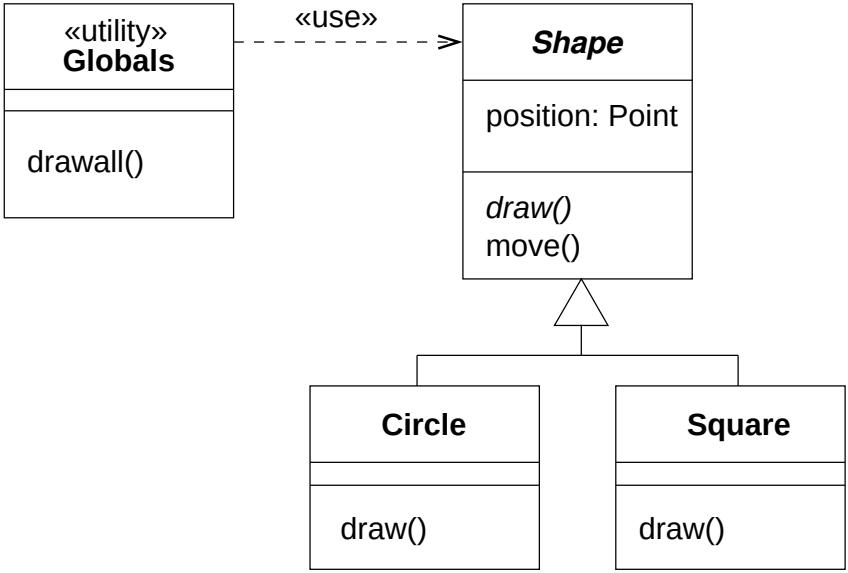
Il progetto OO: eredità e polimorfismo (3)

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.

Il progetto OO: eredità e polimorfismo (4)



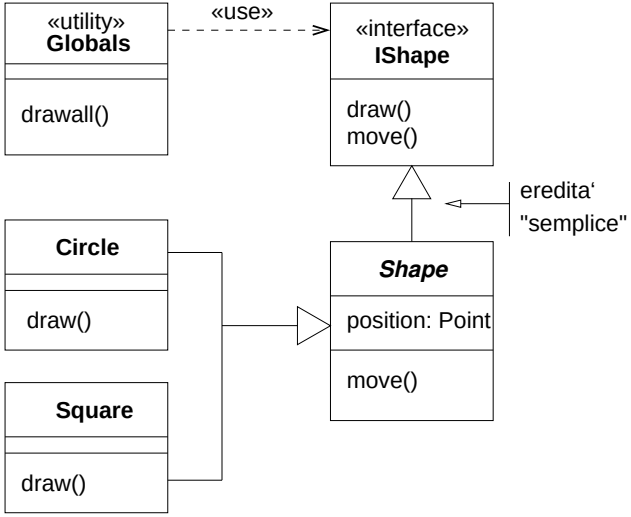
Il progetto OO: eredità e polimorfismo (5)

```
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. */ };
};
```

Il progetto OO: eredità e polimorfismo (6)

```
/* 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);
}
```

Il progetto OO: eredità e polimorfismo (7)



Il progetto OO: eredità e polimorfismo (8)

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; };
};
```

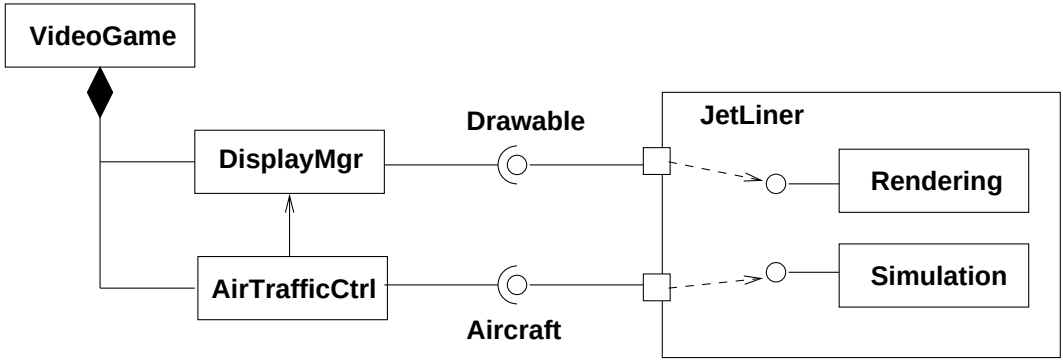
Il progetto OO: eredità e polimorfismo (9)

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

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

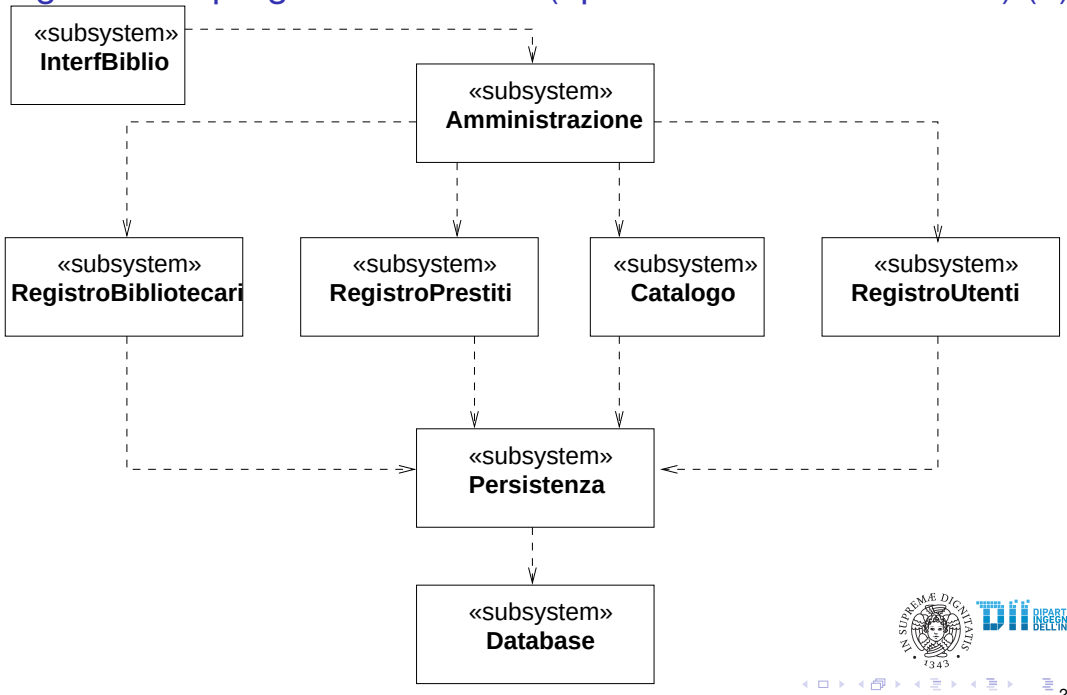
```
void drawall(IShape** shps)
 {
   // metodo immutato ANCHE SE AGGIUNGO NUOVE CLASSI DERIVATE
 }

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

Il progetto OO: eredità e polimorfismo (10)

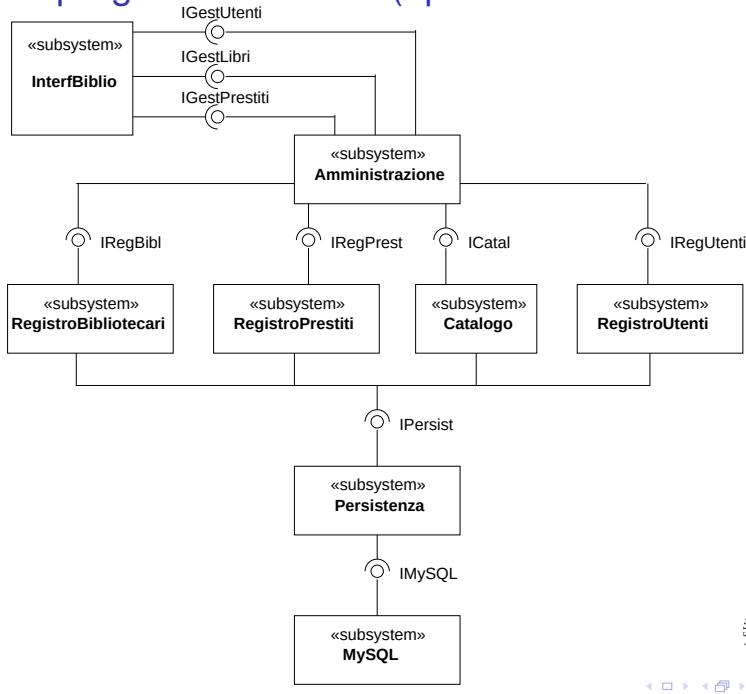


ogni componente di tipo **JetLiner** offre le stesse interfacce, ma le può realizzare per diversi tipi di aereo.

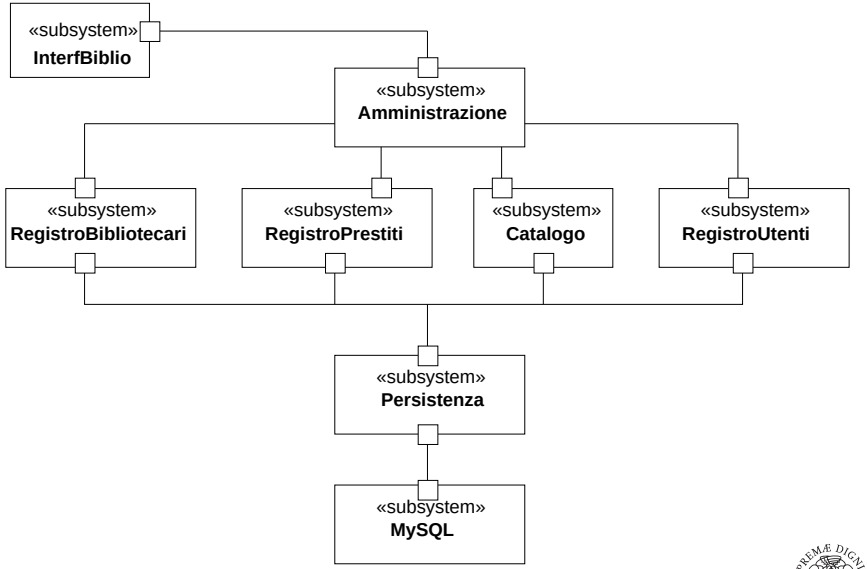
Il progetto OO: progetto di sistema (ripartizione in sottosistemi) (1)



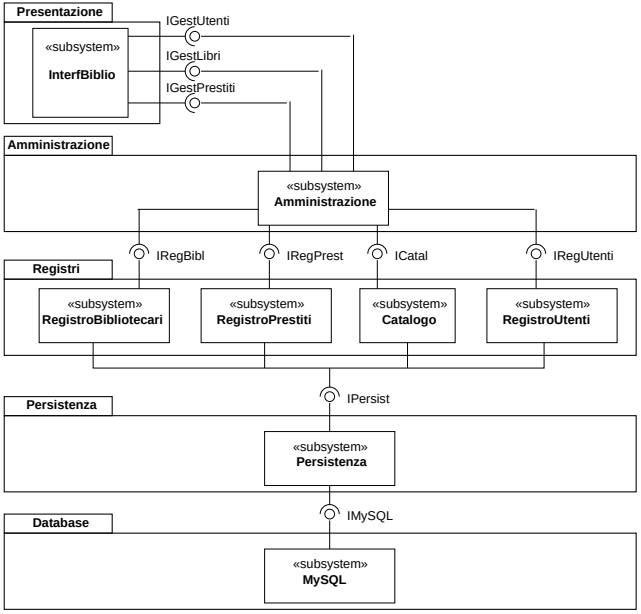
Il progetto OO: progetto di sistema (ripartizione in sottosistemi) (2)



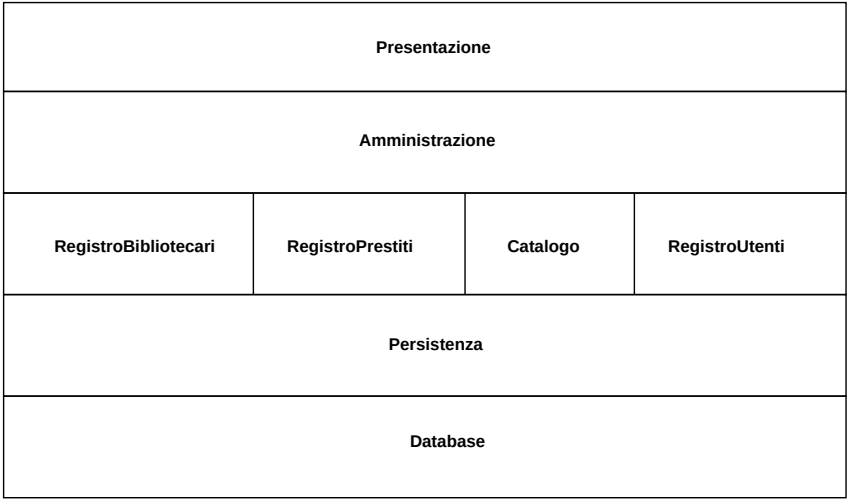
Il progetto OO: progetto di sistema (ripartizione in sottosistemi) (3)



Il progetto OO: progetto di sistema (strati e partizioni) (1)

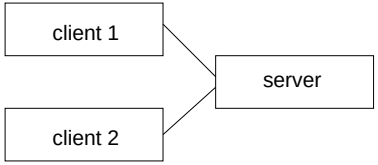


Il progetto OO: progetto di sistema (strati e partizioni) (2)

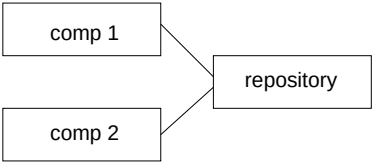


Il progetto OO: progetto di sistema (architetture standard)

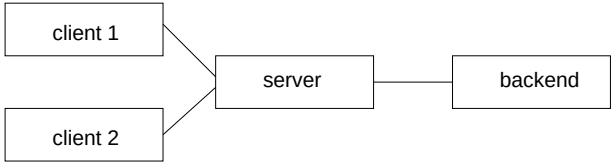
pipeline



client−server a 1 livello



repository



client−server a 2 livell1 (two−tier)

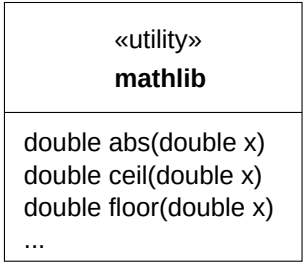
Il progetto OO: progetto di sistema (librerie) (1)

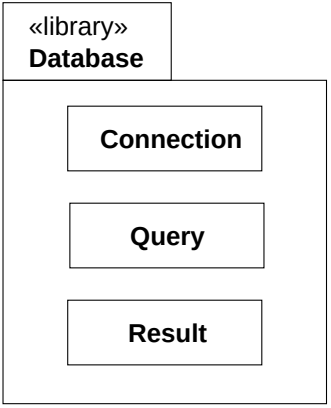
Una libreria logica è una raccolta di componenti che offrono servizî ad un livello di astrazione piuttosto basso.

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.

Il progetto OO: progetto di sistema (librerie) (2)





Il progetto OO: progetto di sistema (framework) (1)

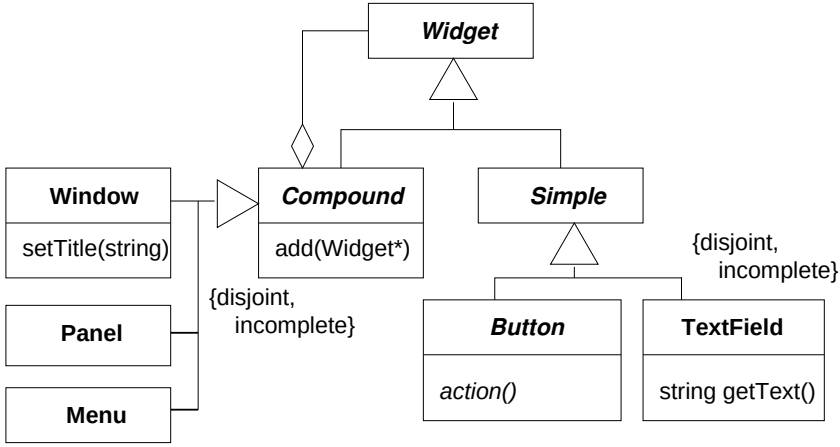
Un framework contiene invece dei componenti ad alto livello di astrazione che offrono uno schema di soluzione preconfezionato per un determinato tipo di problema.

I framework si usano "*dall'alto verso il basso*", riempiendo delle strutture complesse predefinite (delle "intelaiature") con dei componenti semplici specializzati.

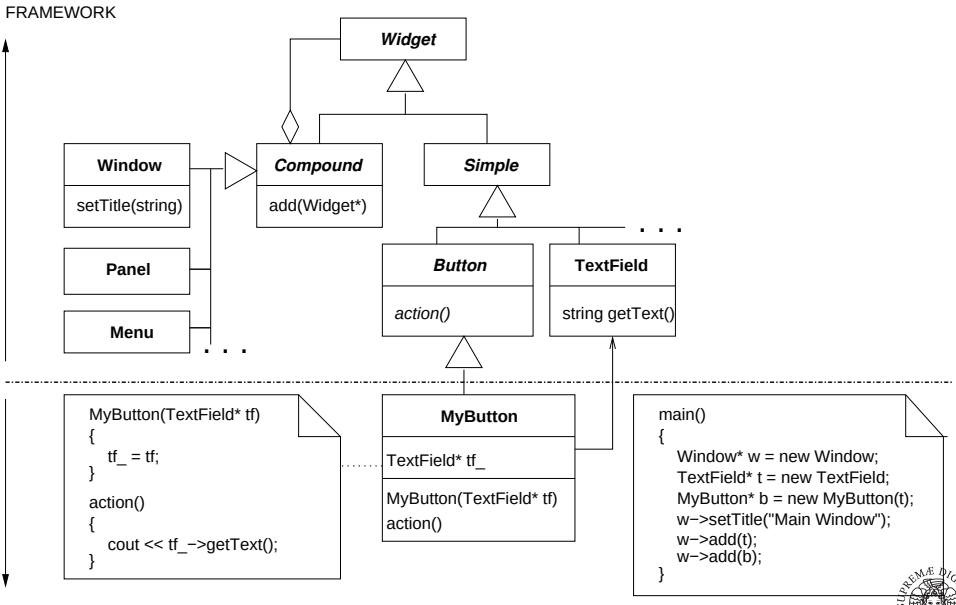
Per esempio, un framework per risolvere generici problemi di equzioni differenziali può essere applicato ad un problema particolare, aggiungendovi moduli specifici sviluppati *ad hoc*. Similmente si può specializzare un framework per problemi geometrici.

Il progetto OO: progetto di sistema (framework) (2)

Esempio: framework per costruire interfacce grafiche

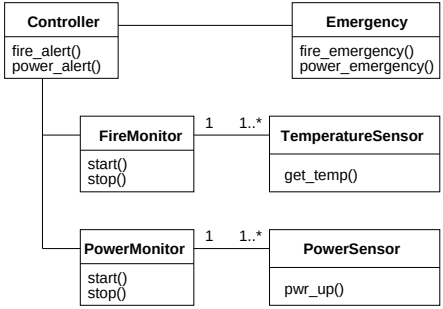


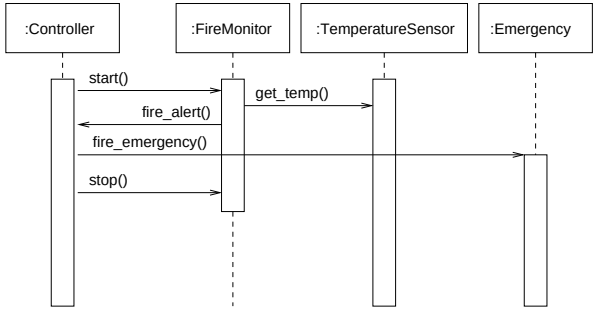
Il progetto OO: progetto di sistema (framework) (3)



Il progetto OO: progetto di sistema (sistemi concorrenti) (1)

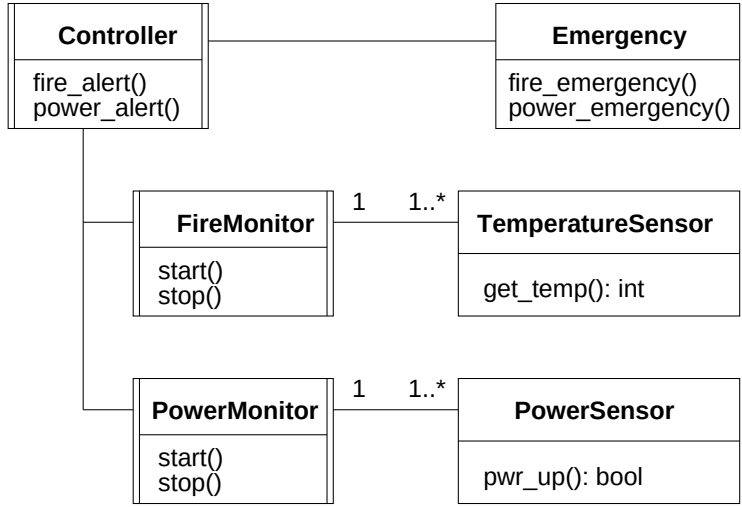
Consideriamo un modello iniziale (di analisi o di progetto) che non tenga conto della concorrenza.





Il progetto OO: progetto di sistema (sistemi concorrenti) (2)

Mettiamo in evidenza le classi *attive* (**Controller**, **FireMonitor**, **PowerMonitor**).

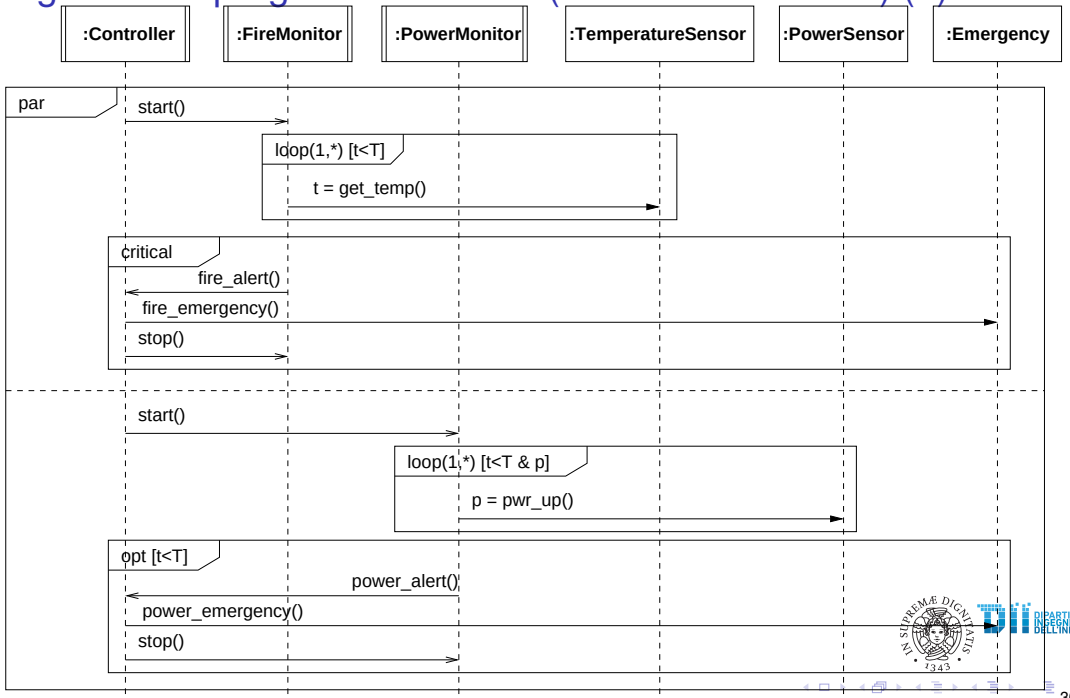


Il progetto OO: progetto di sistema (sistemi concorrenti) (3)

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.

Il progetto OO: progetto di sistema (sistemi concorrenti) (4)



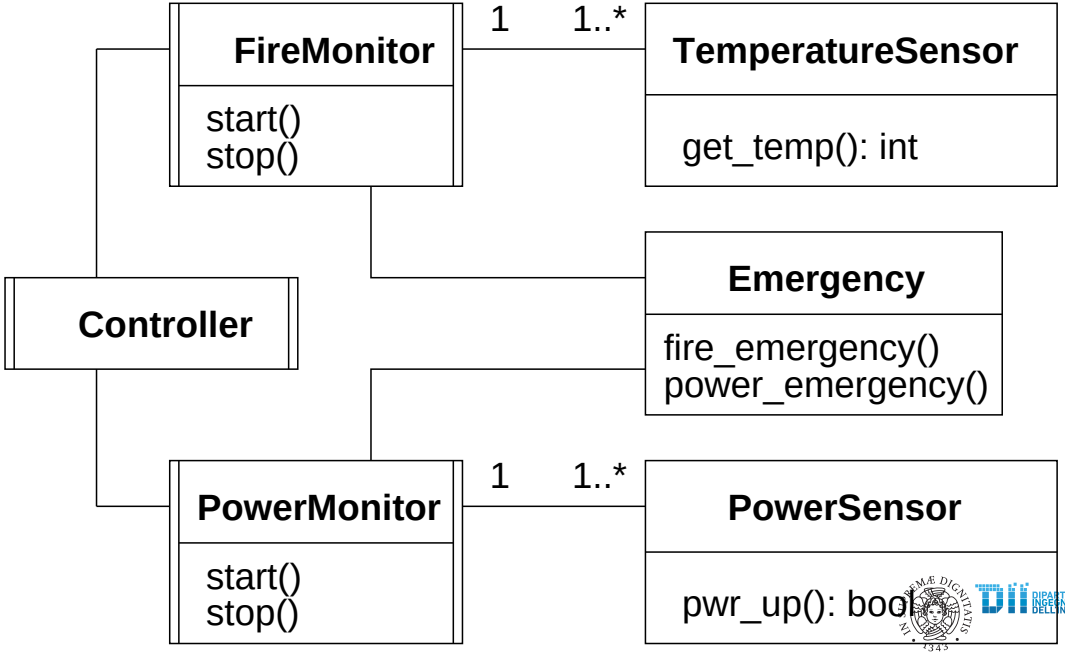
Il progetto OO: progetto di sistema (sistemi concorrenti) (5)

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*.

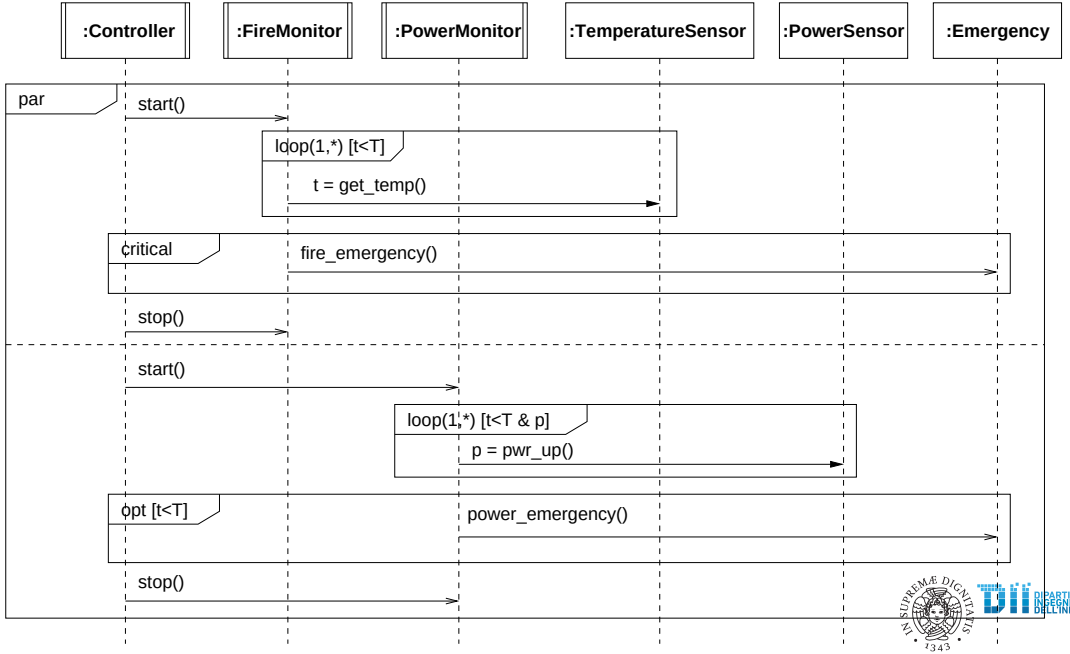
Nel progetto del sistema bisogna tener conto della condivisione di risorse per evitare problemi come il *blocco* (o *deadlock*) o l'*inconsistenza* delle informazioni causata da accessi concorrenti a dati condivisi.

In questo caso particolare, bisogna rispettare il vincolo costituito dalla priorità dell'attività fire_emergency() rispetto all'altra.

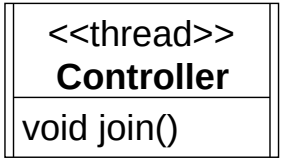
Il progetto OO: progetto di sistema (sistemi concorrenti) (6)

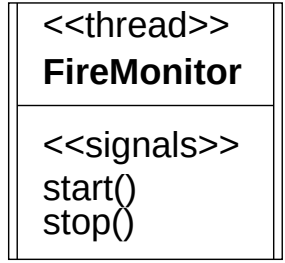


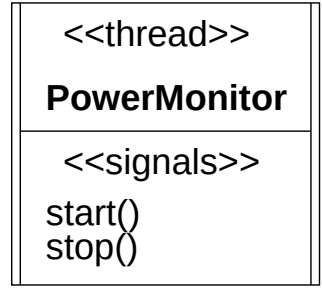
Il progetto OO: progetto di sistema (sistemi concorrenti) (7)

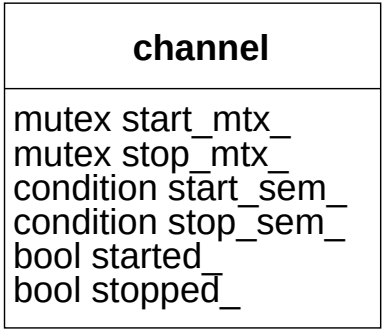


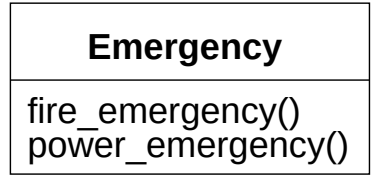
Il progetto OO: progetto di sistema (sistemi concorrenti) (8)





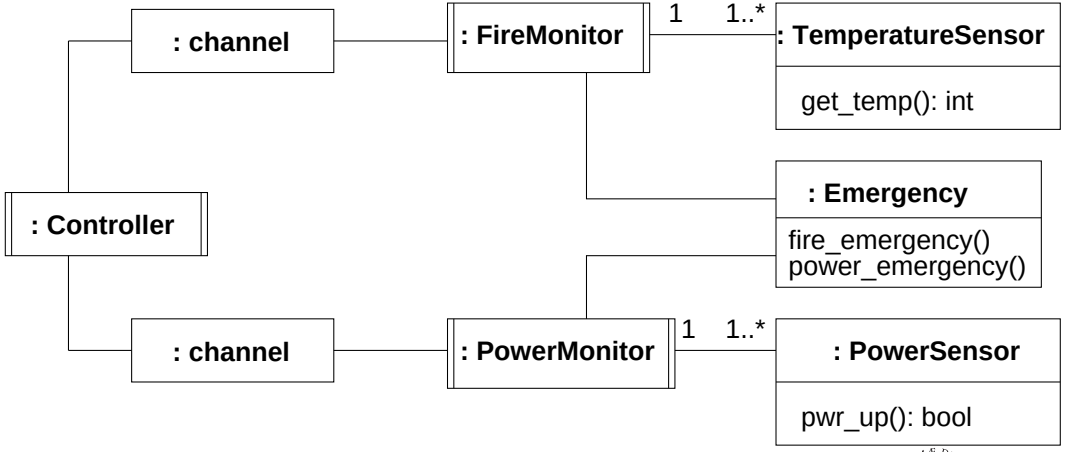




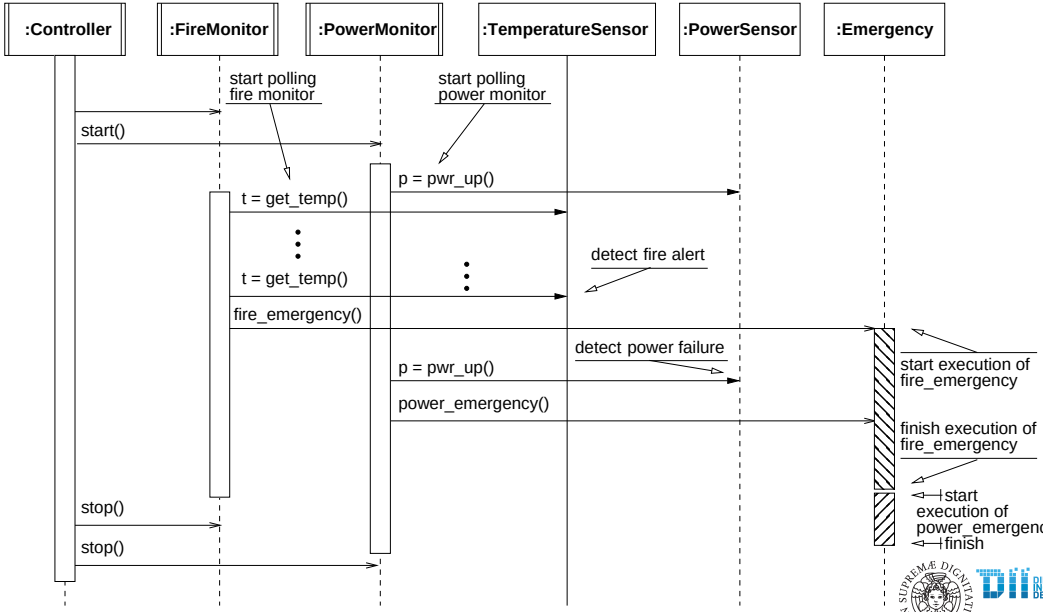


Il progetto OO: progetto di sistema (sistemi concorrenti) (9)

Codice C++ su
http://www.ing.unipi.it/~a009435/issw/es_sist_concorrente.zip.



Il progetto OO: progetto di sistema (sistemi concorrenti) (10)



Il progetto OO: i design pattern (1)

I *design pattern* sono **schemi generali di soluzioni** per problemi ricorrenti.

Un pattern consiste nella descrizione sintetica di un problema e della relativa soluzione.

Specifica gli elementi strutturali (classi, componenti, interfacce...), le relazioni reciproche e il loro modo di interagire.

La descrizione viene integrata con una discussione dei vantaggi e svantaggi della soluzione, delle condizioni di applicabilità e delle possibili tecniche di implementazione.

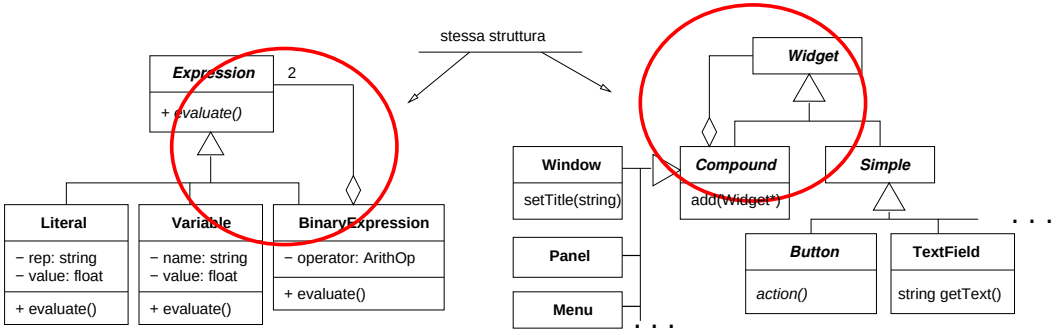
Normalmente vengono presentati anche dei casi di studio.

I design pattern si usano principalmente nella fase intermedia fra progetto di sistema e progetto dettagliato, ma anche nel progetto di sistema (architetture standard).

Il pattern Composite

scopo: offrire un'interfaccia comune a classi le cui istanze possono far parte di strutture gerarchiche.

Il progetto OO: i design pattern (3)

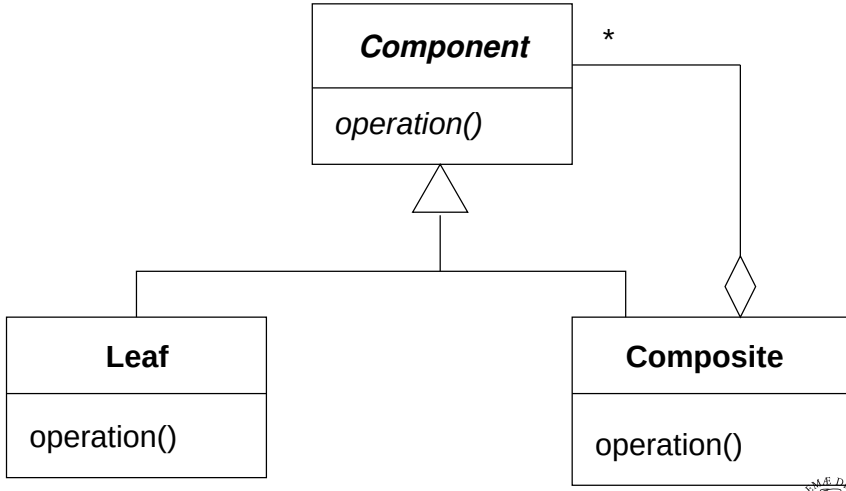


espressioni aritmetiche

interfaccia grafica

Il progetto OO: i design pattern (4)

Pattern “Composite”



Il progetto OO: i design pattern (5)

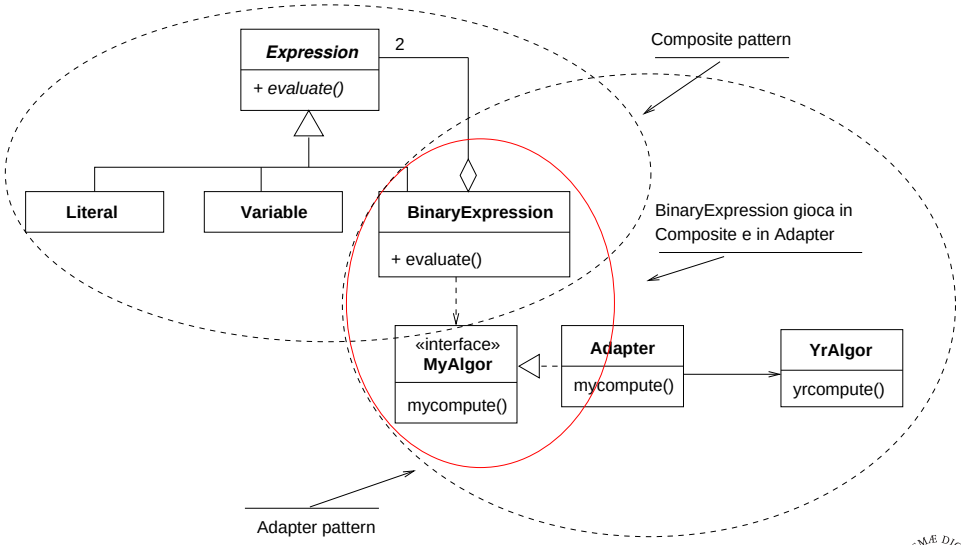
Un design pattern **non è un componente software**, ma solo uno schema di soluzione per un **particolare aspetto** del funzionamento di un sistema.

Gli elementi strutturali di un pattern rappresentano dei *ruoli* che saranno interpretati dalle entità effettivamente realizzate.

Ciascuna di queste entità può interpretare un ruolo diverso in diversi pattern, poiché in un singolo sistema (o sottosistema) si devono risolvere diversi problemi con diversi pattern.

Un pattern non è una ricetta rigida da applicare meccanicamente, ma uno schema che deve essere adattato alle diverse situazioni, anche con un po' d'inventiva.

Il progetto OO: i design pattern (6)



Il pattern Adapter

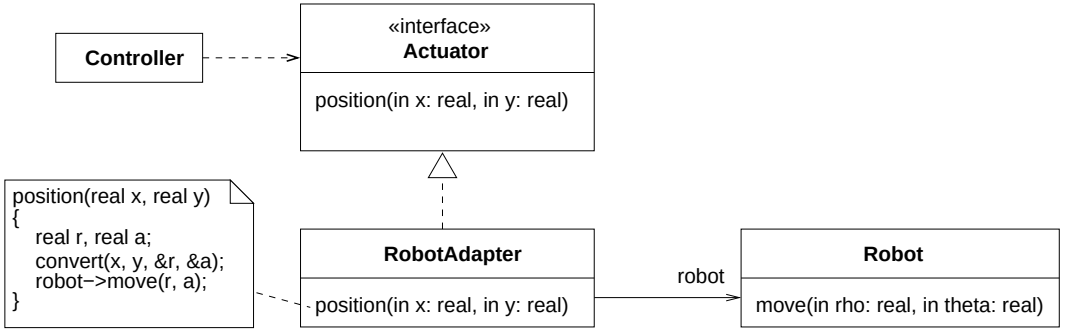
scopo: adattare un'interfaccia offerta ad un'interfaccia richiesta.

esempio: la classe **Controller** richiede l'interfaccia **Actuator**. La semantica dell'operazione astratta position() è "posizionare un utensile sul punto di coordinate cartesiane (x, y)". L'interfaccia offerta dalla classe **Robot** ha l'operazione move() che posiziona l'utensile sul punto di coordinate polari (ρ, θ).

soluzione: inserire la classe **RobotAdapter** che realizza **Actuator** trasformando le coordinate cartesiane in polari, e chiamando l'operazione move().

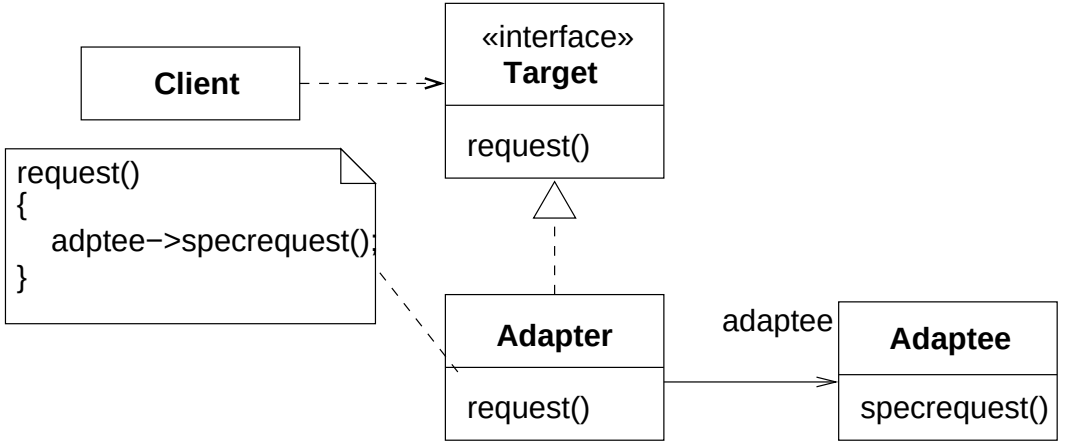
Il progetto OO: i design pattern (8)

Il pattern Adapter: esempio



Il progetto OO: i design pattern (9)

Il pattern Adapter



Il progetto OO: i design pattern (10)

Il pattern Proxy virtuale

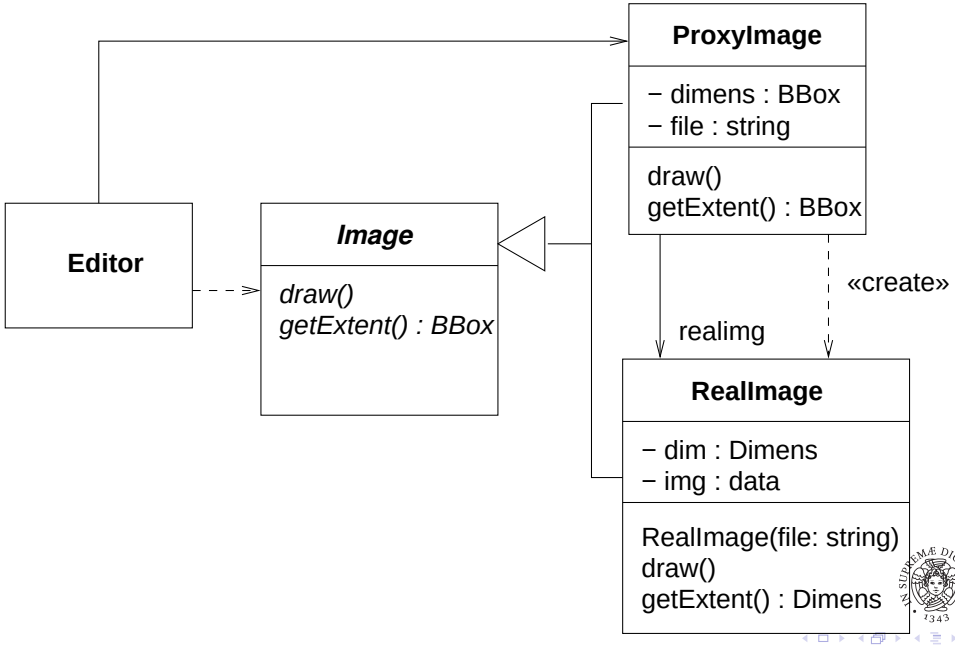
scopo: differire operazioni costose.

esempio: Un programma di videoscrittura deve inserire nel testo delle immagini, implementate da istanze di una classe **Image** che offre le operazioni draw(), che carica in memoria l'immagine e la disegna, e getExtent(), che restituisce le dimensioni dell'immagine. Per impaginare il testo basta che siano note le dimensioni delle immagini, quindi conviene differire il caricamento dell'immagine fintanto che non è necessario visualizzarla.

soluzione: un'istanza della classe **ProxyImage** fa da segnaposto per **RealImage**, che contiene una struttura dati per rappresentare l'immagine. Le chiamate all'operazione getExtent() vengono eseguite direttamente da **ProxyImage**, mentre le chiamate a draw() vengono delegate a **RealImage**, che viene istanziata solo alla prima invocazione di draw().

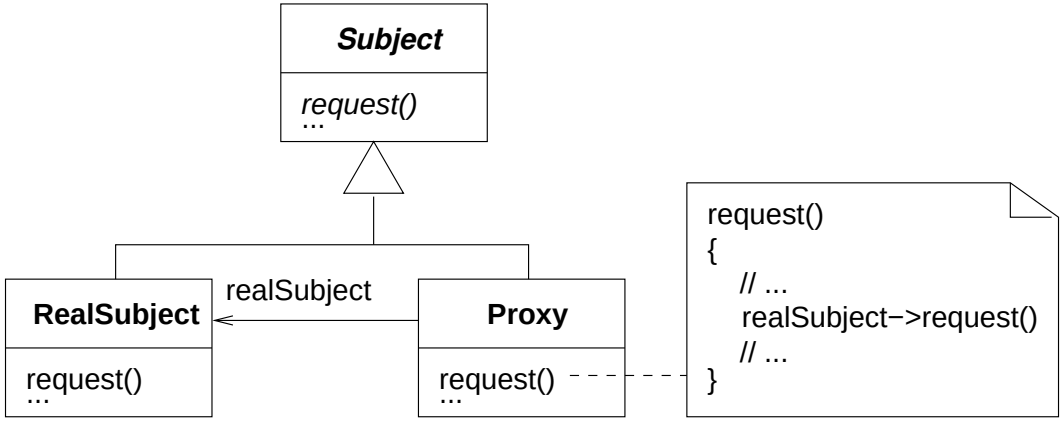
Il progetto OO: i design pattern (11)

Il pattern Proxy virtuale: esempio



Il progetto OO: i design pattern (12)

Il pattern Proxy virtuale



Il pattern Proxy remoto

scopo: chiamare da remoto le operazioni di un oggetto.

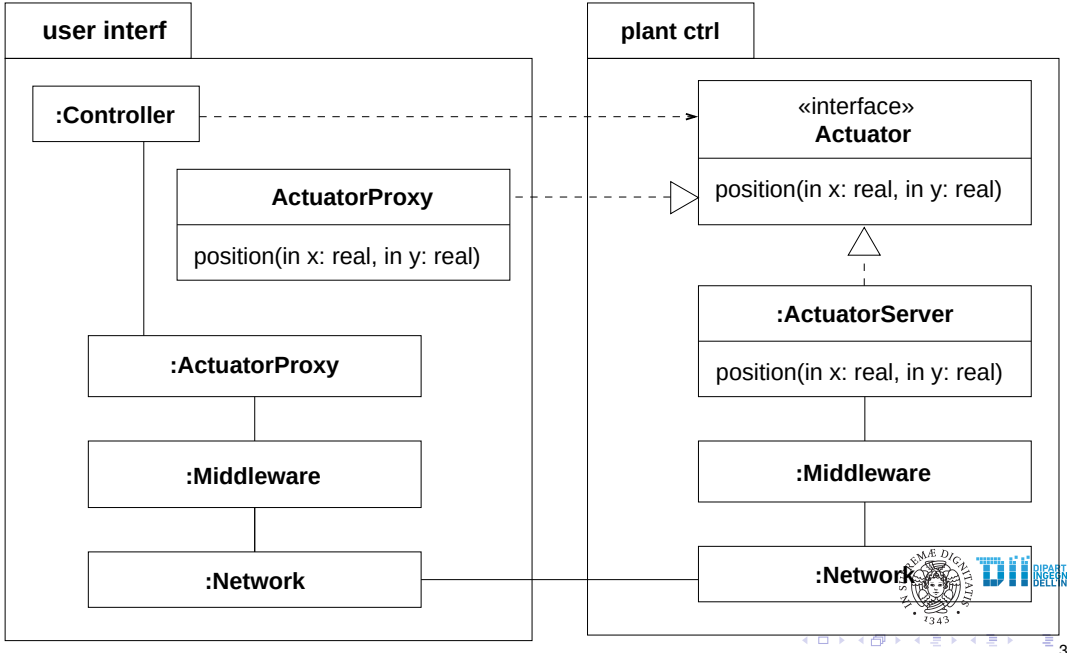
esempio: un impianto deve essere comandato remotamente.

soluzione: l'interfaccia utente (locale) contiene un'istanza della classe **ActuatorProxy** che realizza l'interfaccia di programmazione dell'impianto da controllare. Le operazioni di **ActuatorProxy** inviano messaggi al sistema di controllo remoto, usando un'infrastruttura di comunicazione. Dal lato dell'impianto, i messaggi vengono convertiti in chiamate a un'istanza di **ActuatorServer**.

(V. anche http://www.ing.unipi.it/~a009435/issw/extra/corba0708.pdf).

Il progetto OO: i design pattern (14)

Il pattern Proxy remoto: esempio



Il pattern Bridge

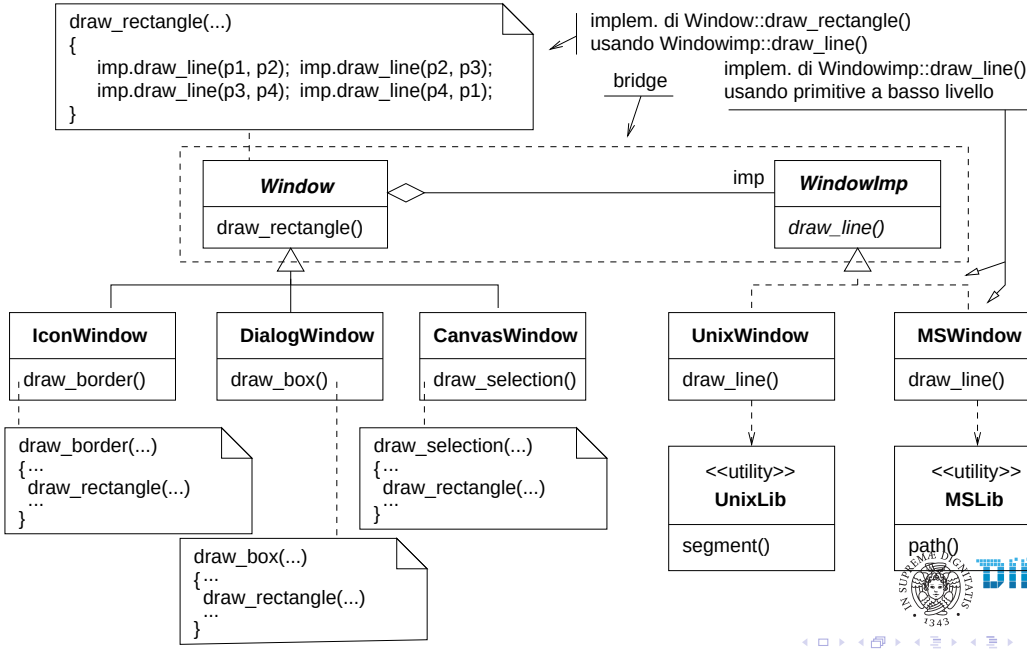
scopo: disaccoppiare una famiglia di classi dalla loro implementazione.

esempio: un'interfaccia grafica (GUI) comprende una famiglia di classi derivate dalla clase **Window**; vogliamo poter implementare l'interfaccia grafica usando diverse librerie grafiche, modificando la GUI indipendentemente dalla libreria.

soluzione: la classe **Window**, che offre operazioni ad alto livello (p.es., draw_rectangle()) dipende da una classe astratta (o interfaccia) **WindowImp** a piú basso livello (p.es., draw_line()). Le operazioni di **Window** e delle classi derivate sono implementate con operazioni (astratte) di **WindowImp**. Queste sono implementate con le operazioni delle librerie grafiche.

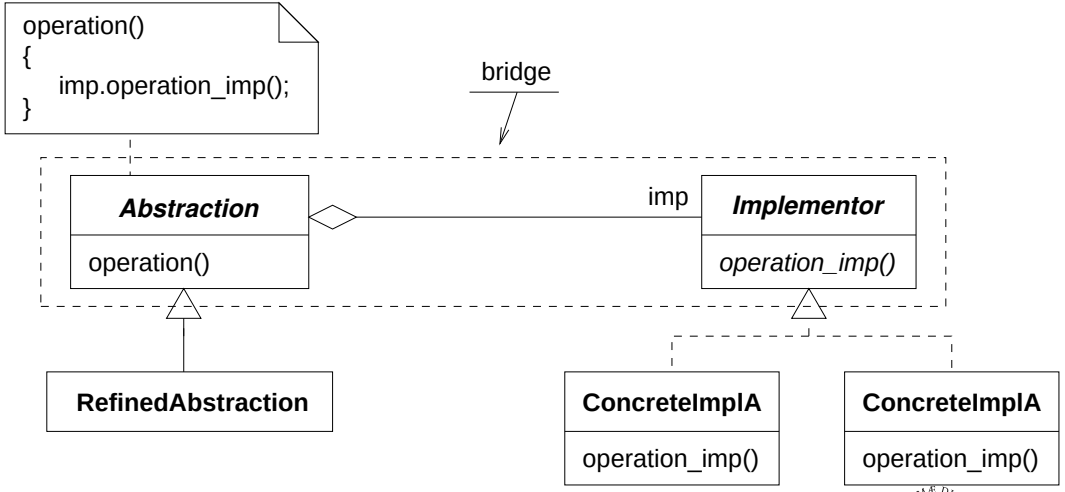
Il progetto OO: i design pattern (16)

Il pattern Bridge: esempio



Il progetto OO: i design pattern (17)

Il pattern Bridge



Il pattern Abstract factory

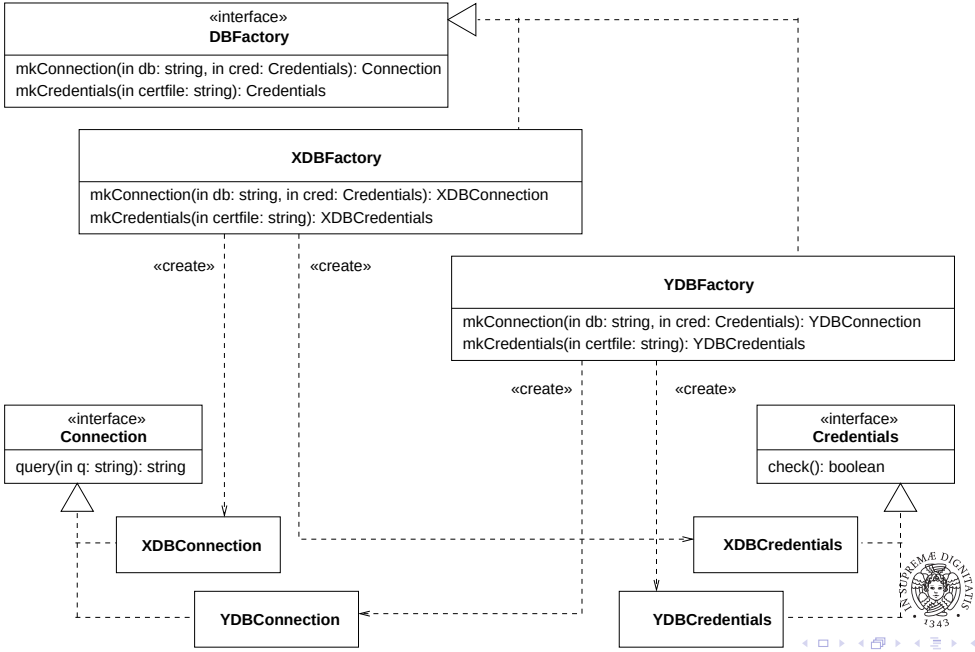
scopo: creare famiglie di oggetti interrelati in modo che i clienti non dipendano dalla loro implementazione.

esempio: un'applicazione accede a un database usando le interfacce **Connection** e **Credentials**. Queste sono implementate da due database diversi, XDB e YDB, scelte dall'applicazione all'inizio dell'esecuzione.

soluzione: si definisce un'interfaccia **DBFactory** per creare istanze di **Connection** e **Credentials**, realizzata dalle classi **XDBFactory** e **YDBFactory**, una per ciascuna delle librerie alternative.

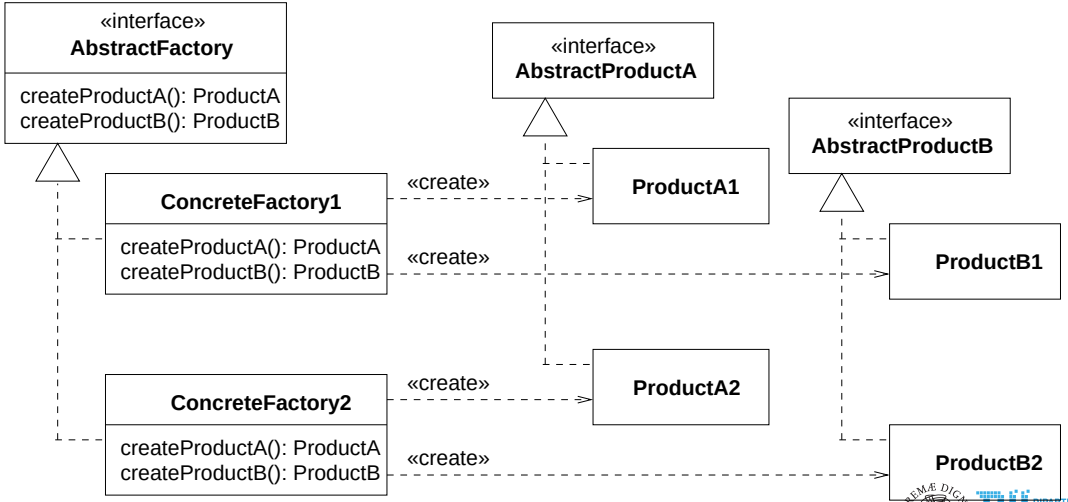
Il progetto OO: i design pattern (19)

Il pattern Abstract factory: esempio



Il progetto OO: i design pattern (20)

Il pattern Abstract factory



Il pattern Iterator

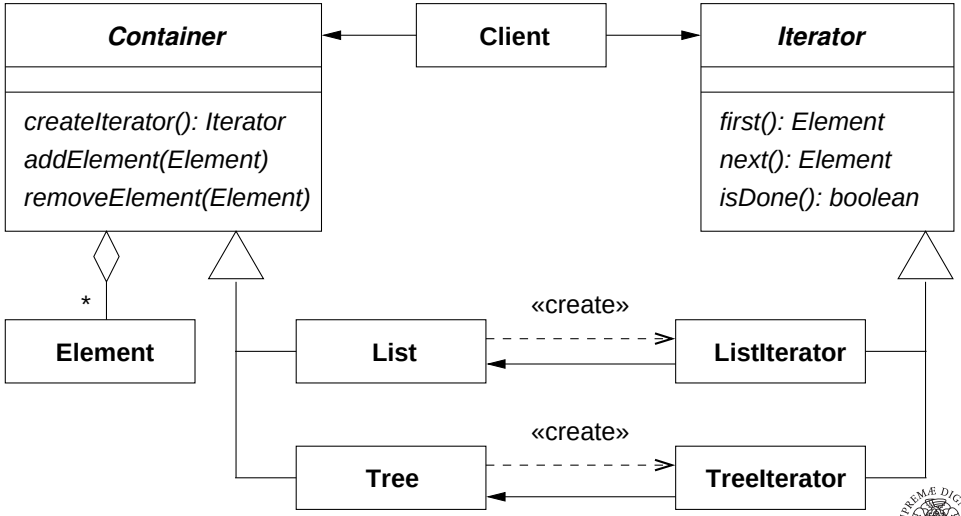
scopo: accedere in sequenza agli elementi di una struttura dati senza dipendere dall'implementazione.

esempio: Una raccolta di oggetti **Element** può essere implementata usando strutture dati diverse, per esempio liste (**List**) o alberi (**Tree**). Un'applicazione deve poter accedere agli elementi di questa raccolta indipendentemente da come viene implementata.

soluzione: si definisce una classe astratta (o un'interfaccia) **Iterator** che offre le operazioni necessarie per accedere in sequenza agli elementi della raccolta, implementate da classi specifiche per ciascuna implementazione (**ListIterator** e **TreeIterator**). La classe astratta (o interfaccia) **Container** offre le operazioni per costruire la raccolta e per creare un iteratore, e viene implementata da **List** o **Tree**.

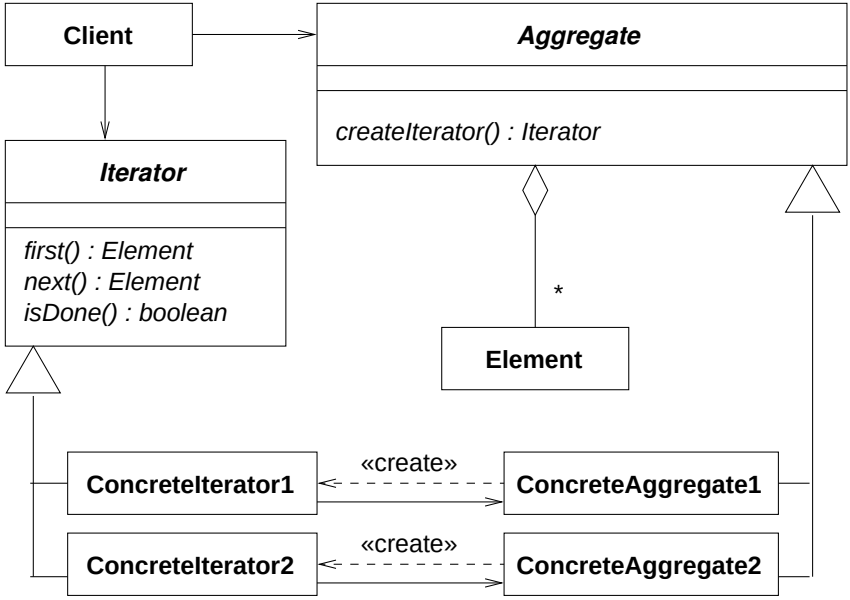
Il progetto OO: i design pattern (22)

Il pattern Iterator: esempio



Il progetto OO: i design pattern (23)

Il pattern Iterator



Il pattern Strategy

scopo: definire una famiglia di algoritmi intercambiabili.

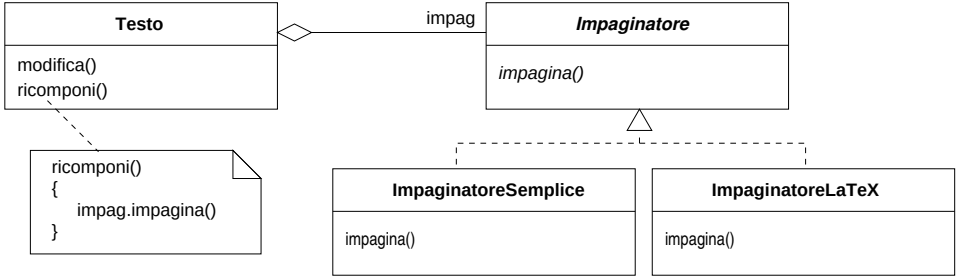
esempio: un testo deve essere (re)impaginato dopo le modifiche usando algoritmi alternativi.

soluzione: si definisce un'interfaccia comune per i diversi algoritmi, implementata da classi diverse.

Oss.: un algoritmo di impaginazione deve riempire la pagina in modo uniforme, tenendo conto delle spaziature, delle dimensioni dei caratteri, delle illustrazioni etc.

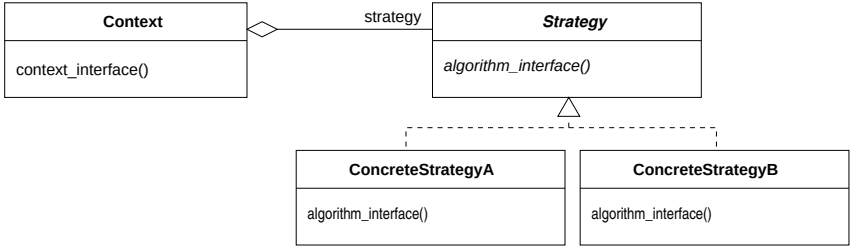
Il progetto OO: i design pattern (25)

Il pattern Strategy: esempio



Il progetto OO: i design pattern (26)

Il pattern Strategy



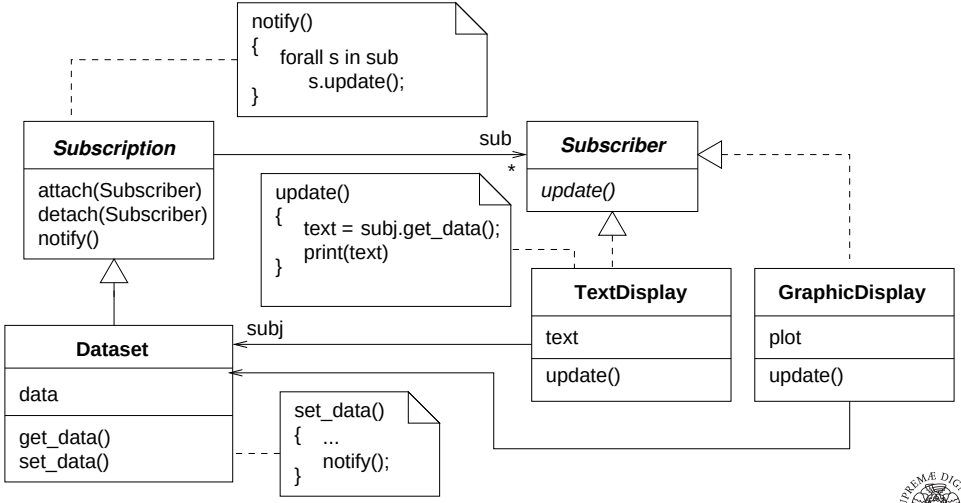
Il pattern Observer (Publish-subscribe)

scopo: definire un sistema di dipendenze in modo che un oggetto possa notificare altri oggetti dei suoi cambiamenti di stato.

esempio: un insieme di dati deve essere visualizzato in diversi modi.

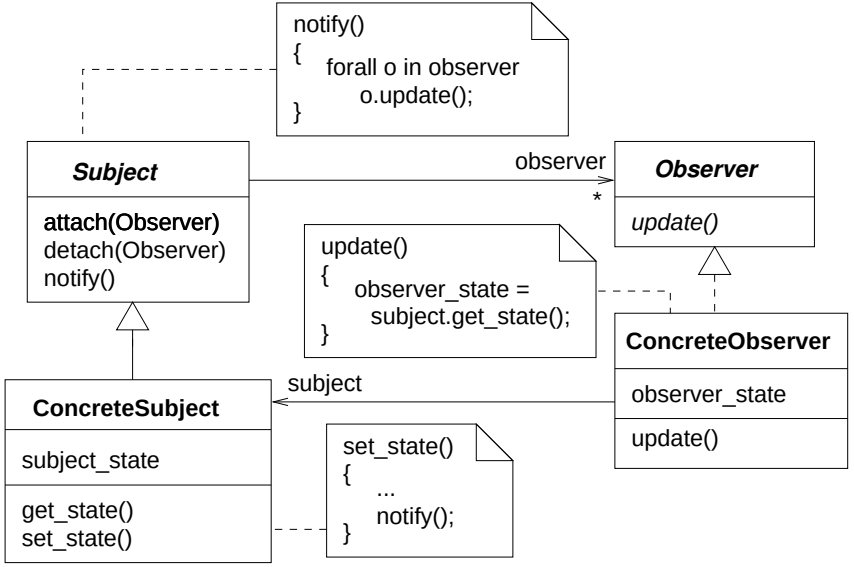
soluzione: si definisce un'interfaccia comune per aggiornare i sistemi di visualizzazione, ed un interfaccia per informare l'insieme dei dati che dei sistemi di visualizzazione devono essere notificati dei cambiamenti di stato.

Il pattern Observer: esempio



Il progetto OO: i design pattern (29)

Il pattern Observer



Il progetto OO: i design pattern (30)

Il pattern Singleton

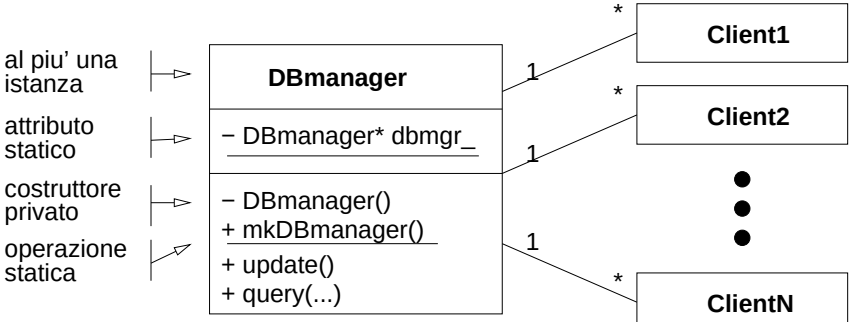
scopo: garantire che nel sistema esista **una sola istanza** di una certa classe.

esempio: un un certo numero di applicazioni deve accedere ad un gestore di un database.

soluzione: si assegna la visibilità privata (o protetta) al costruttore della classe che implementa il gestore. Le applicazioni clienti ottengono un puntatore all'unica istanza per mezzo di un'operazione statica. Questa alloca un'istanza del gestore in memoria dinamica solo la prima volta che viene invocata.

Il progetto OO: i design pattern (31)

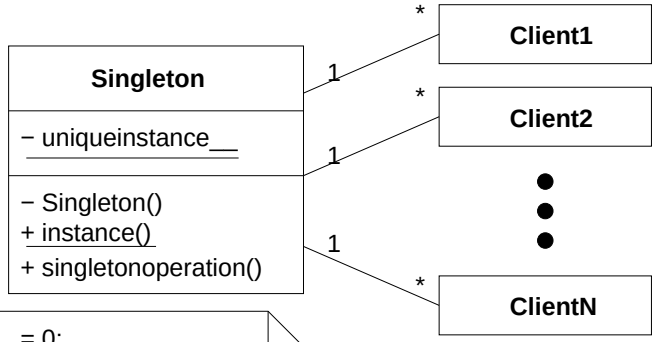
Il pattern Singleton: esempio




```
DBmanager* DBmanager::dbmgr_ = 0;

DBmanager* DBmanager mkDBmanager()
{
   if (dbmgr_ == 0)
      dbmgr_ = new DBmanager;
   return dbmg_;
}
```

Il progetto OO: i design pattern (32)

Il pattern Singleton




```
uniqueinstance_ = 0;

instance()
{
   if (uniqueinstance_ == 0)
      uniqueinstance_ = new Singleton;
   return uniqueinstance_;
}
```

Il progetto OO: progetto in dettaglio (1)

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.

Il progetto OO: progetto in dettaglio (2)

Proprietà delle classi

- Completezza
- sufficienza
- primitività
- coesività
- disaccuppiamento.

Evitare ottimizzazioni inutili (usare strumenti di profilazione).

Il progetto OO: progetto in dettaglio (3)

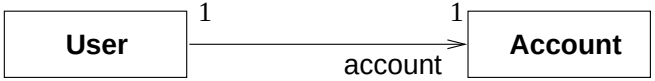
Progetto delle associazioni

- 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?

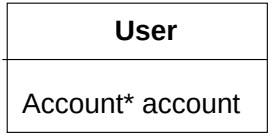
Il progetto OO: progetto in dettaglio (4)

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



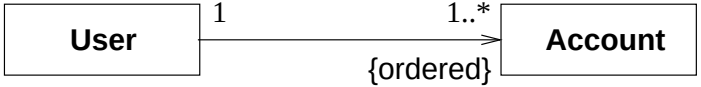
Considerare la scelta fra associazione e attributo.

Il progetto OO: progetto in dettaglio (5)

Progetto delle associazioni: 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.

Considerare l'uso di iteratori.



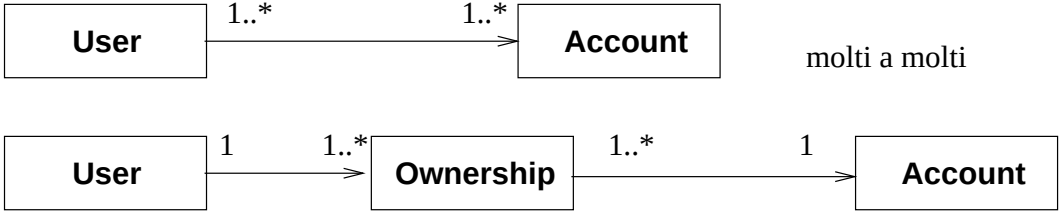
uno a molti

Il progetto OO: progetto in dettaglio (6)

Progetto delle associazioni: 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.

Nell'esempio, ogni istanza di **Ownership** contiene un puntatore ad **Account**.

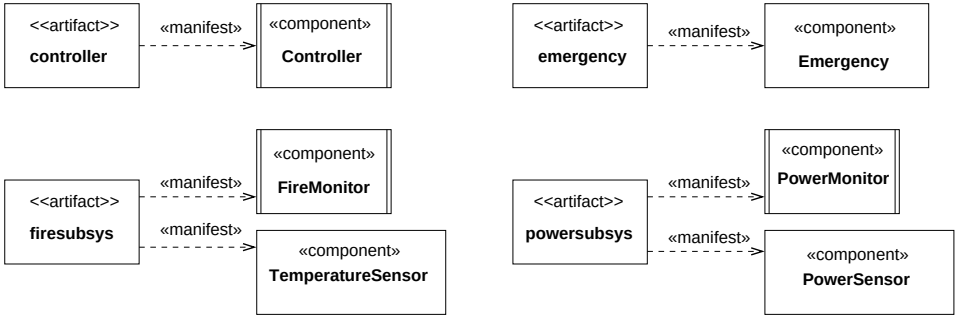


Il progetto OO: l'architettura fisica (1)

- architettura logica: **moduli**:
  - componenti
  - classi
  - ...
- architettura fisica:
  - software (modella le dipendenze per la costruzione (*build*) e l'esecuzione del sw): **artefatti**:
    - file eseguibili
    - file collegabili (.o, .obj)
    - librerie (.a, .so, .lib, .dll, .jar ...)
    - codice sorgente, dati, pagine web ...
  - hardware: **nodi**:
    - calcolatori
    - processori
    - memorie
    - cluster
    - dispositivi (*device*) ...

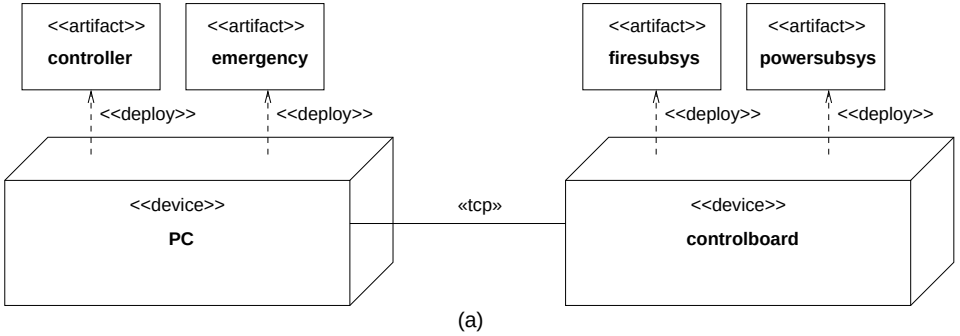
Il progetto OO: l'architettura fisica (2)

Relazione fra moduli e artefatti (manifest)



Il progetto OO: l'architettura fisica (3)

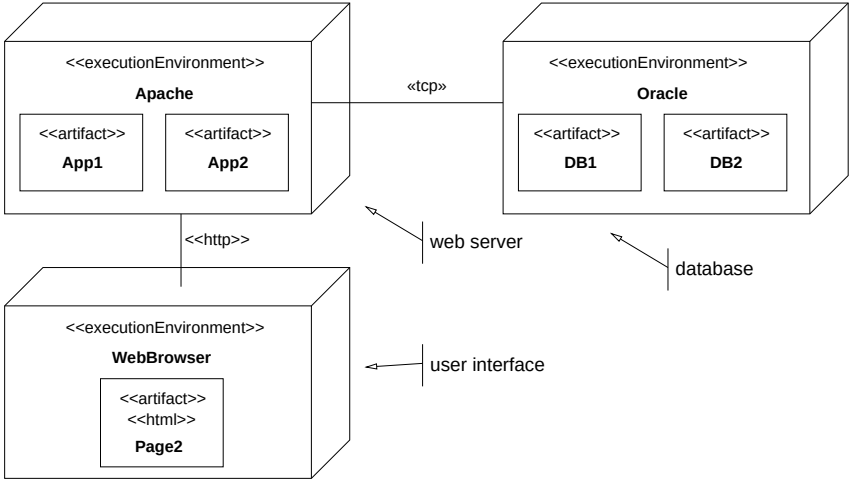
Relazione fra nodi e artefatti (deploy)



Il progetto OO: l'architettura fisica (4)

Ambienti di esecuzione

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



Gestione dei dati

Gestione dei dati (1)

- formati di rappresentazione;
  - binario
  - ascii
  - XML e derivati
  - JSON . . .
- specifica dei formati di rappresentazione;
  - ASN.1 (Abstract Syntax Notation 1)
  - grammatiche regolari (riconoscono token)
  - grammatiche non contestuali (riconoscono statement) . . .
- metodi di memorizzazione
  - file
  - database
    - relazionali (SQL)
    - gerarchici (*Lightweight Directory Access Protocol* LDAP, *Hierarchical Data Format* HDF5)
    - object-oriented . . .
- sistemi distribuiti
  - transazioni
  - replicazione
  - consistenza . . .

Gestione dei dati (2)

ASN.1

- La **Abstract Syntax Notation** serve a descrivere i dati trasmessi nei protocolli si comunicazione.
- I dati sono organizzati in vario modo (strutture, sequenze . . .) a partire da tipi predefiniti (*INTEGER, BOOLEAN . . .*)
- La specifica dei dati è **indipendente** dalla loro rappresentazione concreta.
- I dati specificati in ASN.1 possono essere rappresentati in varii modi, descritti da **regole di codifica** (*encoding rules*).

```
WeatherReport ::= SEQUENCE
{
    stationNumber    INTEGER (1..99999),
    timeOfReport     UTCTime,
    pressure         INTEGER (850..1100),
    temperature      INTEGER (-100..60),
}
```

Gestione dei dati (3)

Grammatiche regolari

- Dato un alfabeto finito di simboli, un'espressione regolare (ER) definisce un insieme di sequenze (o *stringhe*) di simboli.
- Un'ER è formata da simboli e operatori, fra cui la **ripetizione**, la **scelta** e la **concatenazione**.
- Programmi come il *Lex* producono il codice che riconosce se una stringa di caratteri appartiene all'insieme definito da un'ER.

```
[a-zA-Z_][a-zA-Z_0-9]*     return( IDENTIFIER );
[0-9]+                     return( INTEGER );
"="                        return( '=' );
"+"                        return( '+' );
"-"                        return( '-' );
"*"                        return( '*' );
"/"                        return( '/' );
```

Gestione dei dati (4)

Grammatiche non contestuali

- Dato un alfabeto finito di **simboli terminali** (**token**), una **produzione** è una regola (anche *ricorsiva*) che definisce un **simbolo non terminale**, cioè un insieme di sequenze di simboli terminali e non terminali.
- Nei linguaggi di programmazione, i simboli terminali sono gli *identificatori*, le *parole chiave* etc. I simboli non terminali sono i varii tipi di istruzioni.
- Programmi come lo *Yacc* producono il codice che riconosce se una sequenza di simboli appartiene all'insieme definito da una produzione.
- Le grammatiche non contestuali servono anche a definire linguaggi per la rappresentazione di dati *strutturati*.

```
espr : espr '+' espr
     | espr '-' espr
     | espr '*' espr
     | espr '/' espr
     | IDENTIFIER
     | INTEGER
     ;
```

CONVALIDA E VERIFICA

(By Simon A. Eugster - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=7900253)

Indice

Convalida e verifica

- *Verification* checks an implementation against its specification.
  - *Are we making the product right?*
- *Validation* checks a product (final or intermediate) against its intended requirements.
  - *Are we making the right product?*

With these definitions, deliverables at any stage of the process may be validated.

Convalida e verifica: analisi statica (1)

The following is based on IAEA TRS 384:

- *Walk-through*: A document (such as code, design, etc.) is presented to a group including developers and people possibly not involved in development. The document is evaluated and criticized.
- *Inspection*: A document or set of documents is read by a group of people who check the document(s) for **expected defects** of some class (e.g., typical programming mistakes, common design flaws. . .). Checklists are commonly used.
- *Automatic analysis*: Automatic tools (*including compilers*) can check a program for indicators of possible anomalies, such as unreachable statements or usage of non initialized variables.

Convalida e verifica: analisi statica (2)

- *Formalized descriptions*: Writing a specification (or design) document using a formal language. This is **a way to validate an informal requirements statement**, and a pre-requisite for formal verification.
- *Program proving*: *Assertions* (statements about relationships among variables) are associated with the beginning (*pre-conditions*) and the end (*post-conditions*) of a program segment (such as a procedure), and if the program is correct, the post-conditions can be proved by logical arguments to be a consequence of the pre-conditions.
- *Symbolic execution*: The input variables are assigned symbolic values (say, x) instead of numeric ones. A symbolic interpreter applies the program statements to the input variables and computes the output values as symbolic (i.e., algebraic or logical) expressions that can be checked for compliance with the program specification.

Convalida e verifica: analisi statica (3)

A formal model enables developers to find out about important properties of the system:

- *Safety* properties: undesired states will *not* be reached, undesired actions will *not* be executed.
  - E.g., *Will the temperature raise above a given threshold? Will the reactor trip on a false alarm?*
- *Liveness* properties: desired states *will* be reached, desired actions *will* be executed.
  - E.g., *Will the reactor reach full power? Will it trip on a real alarm?*

Convalida e verifica: analisi dinamica

(www.adeptis.ru/vinci/m_part7.html)

Program testing can be used to show the *presence* of bugs, but never to show their absence.

E. Dijkstra, quoted in Dahl et al., *Structured Programming*.

Convalida e verifica: analisi dinamica (1)

- *Prototype execution*: A *prototype* is a partial or simplified version of a software system, whose internal structure is often unrelated to that of the final product. Prototypes can be used to assess the feasibility of design choices, to let users validate the specifiers' interpretation of the requirements, and to experiment with different design choices.
- *Simulation*: The environment (e.g., the plant) where the software will operate may be simulated by a simulator program.
- *Testing*: The software is exercised by a selected set of inputs (*test data*) to discover possible malfunctions.

Convalida e verifica: analisi dinamica (2)

Selezione dei dati può essere guidata:

- dalle **specifiche** (test **funzionale**, *black-box*);
- dalla **struttura del programma** (test **strutturale**, *white-box*);

nessuno dei due criteri da solo è sufficiente, poiché ciascuno di essi permette di scoprire tipi diversi di guasti.

Correttezza dei risultati può essere decisa:

- dall'utente (convalida);
- dal confronto con le specifiche (verifica); in questo caso, le specifiche si rivelano tanto piú utili quanto piú sono formali;
- dal confronto con versioni precedenti; avviene nei test di *regressione*, in cui si verifica una nuova versione del software;

Terminazione può essere decisa in base a modelli statistici che permettano di stimare il numero di anomalie sopravvissute al test, oppure in base a **criteri di copertura**.

Convalida e verifica: analisi dinamica (3)

Faults and failures

- A *failure* (*guasto*, *malfunzionamento*) is an incorrect behaviour of the software.
- A *fault* (*difetto*, *anomalia*) (commonly known as *bug*) is a defect in the software that *may* cause an observable failure.
- Software failures are *systematic*, not random.
- Nevertheless, software failures appear to occur randomly, since a given failure may occur only when a particular “unlucky” input *activates* the fault.
- Further, a given fault may cause different failures, a given failure may be produced by different faults, a fault may hide another one. . .

Convalida e verifica: analisi dinamica (4)

Oracles

An *oracle* is someone or something that tells us if a test result is correct or not. How does the oracle know?

- Well known data (e.g., standard numerical tables).
- Hand computation.
- Comparison with results of previous or alternate versions.
- *Executable specifications*: A prototype is written in a formal executable language (e.g., LOTOS, Prolog. . .), and used as a reference.

Convalida e verifica: analisi dinamica (5)

- *Unit testing*: Each unit module is tested in the coding phase.
- *Integration testing*: In the SW integration phase, the correct interfacing of each unit module with the rest of the system is tested.
- *Regression testing*: After each change in the software, tests are made to ensure that at least the previous functionalities are preserved. Regression testing is then applied during coding, integration, and maintenance.
- *System testing*: The final system is tested.

Convalida e verifica: analisi dinamica (6)

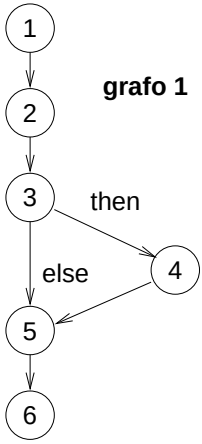
Grafo di controllo e criteri di copertura

- il *grafo di conrollo* è un'astrazione di un algoritmo
- ogni *nodo* del grafo rappresenta un'istruzione (o *comando*) del programma da testare
- ogni *arco* rappresenta il passaggio del flusso di controllo da un'istruzione all'altra
- un *percorso* è una successione di nodi e archi
- un *criterio di copertura* stabilisce quali elementi del grafo devono essere esercitati nell'attività di testing
  - copertura dei *comandi*
  - copertura delle *decisioni*
  - copertura delle *condizioni*
  - copertura delle *decisioni e condizioni*
  - copertura dei *cicli*

Convalida e verifica: analisi dinamica (7)

Copertura dei comandi

```
1   read(x);
2   read(y);
3   if (x != 0)
4       x = x + 10;
5   y = y / x;
6   write(x, y);
```



grafo 1

Any single pair (x_1, y_1) with $x_1 \neq 0$ *covers* all statements, but it does not activate a possible *divide-by-zero* failure at stmt 5.

Different *coverage criteria* spot different types of faults.

Convalida e verifica: analisi dinamica (8)

Copertura delle decisioni (*branch coverage*)

Richiede che ogni arco del grafo di controllo venga percorso almeno una volta.

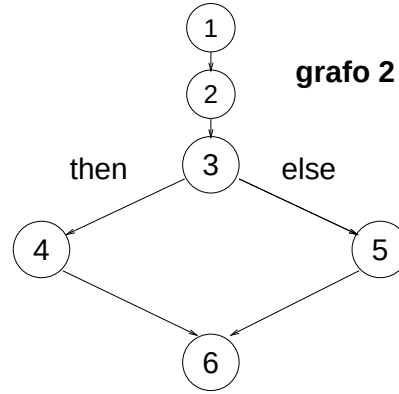
Per il programma precedente, il criterio di copertura delle decisioni (*branch coverage*) richiede che ogni test contenga sia coppie tali che la condizione `(x != 0)` sia vera, sia coppie tali che sia falsa, ovvero almeno una coppia con x diverso da zero ed una con x uguale a zero, e cosí si trova il malfunzionamento dovuto alla divisione per zero.

Convalida e verifica: analisi dinamica (9)

Copertura delle decisioni

Per altri programmi il questo criterio non basta. Nel programma seguente, è possibile una divisione per zero sia nel ramo *then* che nel ramo *else*:

```
1   read(x);
2   read(y);
3   if (x == 0 || y > 0)
4       y = y / x;
    else
5       x = y + 2 / x;
6   write(x, y);
```



grafo 2

Un test accettabile è `{(x = 5, y = 5), (x = 5, y = -5)}`. Nemmeno questo test, però, è capace di rilevare il malfunzionamento dovuto alla divisione per zero.

Convalida e verifica: analisi dinamica (10)

Copertura delle condizioni

Richiede che ciascuna *condizione elementare* di ogni comando di branch venga resa sia vera che falsa dai diversi dati appartenenti al test.

Nell'esempio considerato, questo criterio viene soddisfatto, per esempio, dal test `{(x = 0, y = -5), (x = 5, y = 5)}`.

Questo test, però, non soddisfa il criterio di copertura delle decisioni (si può verificare che per il test visto la condizione complessiva del comando `if` sia sempre vera), per cui non riesce a rilevare la divisione per zero nel ramo `else`.

Convalida e verifica: analisi dinamica (11)

Copertura delle decisioni e delle condizioni

Richiede che sia le condizioni elementari sia le condizioni complessive siano vere e false per diversi dati di test.

Nell'esempio considerato, questo criterio viene soddisfatto dal test `{(x = 0, y = -5), (x = 5, y = 5), (x = 5, y = -5)}`;

osserviamo però che in questo caso **non** si rileva la divisione per zero nel ramo `then`.

Convalida e verifica: analisi dinamica (12)

Questa tabella riassume i tre esempi relativi al grafo 2:

test set	condizioni $x = 0$	condizioni $y > 0$	ramo	criterio soddisfatto	trova guasto nel ramo
(x = 5, y = 5)	F	T	then	D	no
(x = 5, y = -5)	F	F	else		no
(x = 5, y = 5)	F	T	then	C	no
(x = 0, y = -5)	T	F	then		sí
(x = 5, y = 5)	F	T	then		no
(x = 0, y = -5)	T	F	then	D, C	sí
(x = 5, y = -5)	F	F	else		no

criterio: D = decisioni, C = condizioni, DC = decisioni e condizioni

Convalida e verifica: analisi dinamica (13)

Criteri *data flow*

A form of structural criteria, where data are selected in order to cover all paths containing significant operations on variables:

- *Variable definition*.
- *Variable usage in a computation*.
- *Variable usage in the evaluation of a logical condition*.

Convalida e verifica: analisi dinamica (14)

Test funzionale

- *Equivalence classes and boundary values*: The set of possible inputs (including invalid ones) is divided into subsets, such that the values within each class produce equivalent outputs (under some criterion). Test data are then chosen “inside” each class and at the class boundaries.
- *Decision tables*: First, we identify logical conditions that different outputs must satisfy (e.g., a variable may be positive or negative, a signal may be ON or OFF...), then we find combinations of conditions on inputs that make output conditions true or false. These relationships are summarized in a decision table (or graph). Data are selected to cover all columns of the table.
- *Formal models*: If there is a formal model, test data may be selected by criteria based on the model. For example, if the system is modeled as a state machine, possible criteria are coverage of all states, all transitions, all paths...

Test funzionale: classi di equivalenza

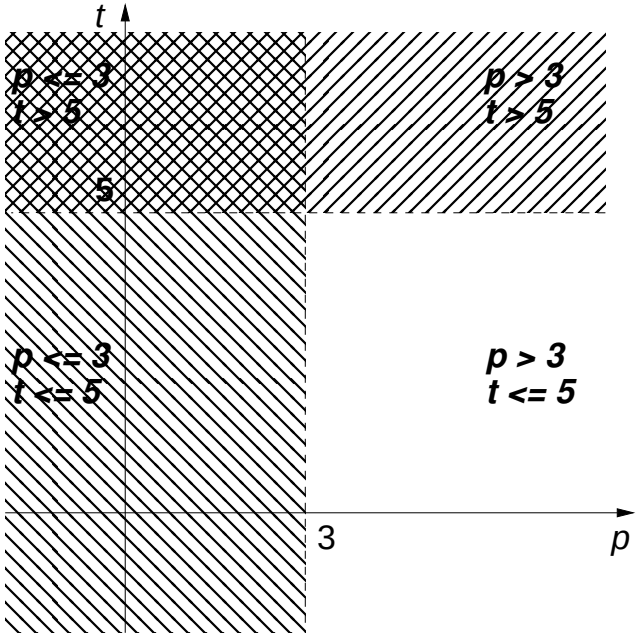
Supponiamo che un programma sia stato cosí specificato:

L'operazione A deve essere eseguita nel caso che la pressione sia minore o uguale a 3 bar, oppure la temperatura sia maggiore di 5 °C*. Altrimenti deve essere eseguita l'operazione B.*

Le classi di equivalenza sono costituite dagli insiemi $\{p \mid p \leq 3\}$ e $\{t \mid t > 5\}$, dalla loro unione. dalla loro intersezione e dal complemento della loro unione, come mostrato nel diagramma seguente.

Convalida e verifica: analisi dinamica (16)

Test funzionale: classi di equivalenza



Convalida e verifica: analisi dinamica (17)

Test funzionale: tabelle e grafi

Esempio: un catalogo accetta comandi di due caratteri, come, p.es., `D4` o `L4` per visualizzare la sez. 4. Il sistema esegue il comando se ben formato, altrimenti emette un messaggio di errore.

Cause	
1	il primo carattere è `D`
2	il primo carattere è `L`
3	il secondo carattere è una cifra
20	il primo carattere è `D` oppure `L`

Effetti	
50	visualizzazione della sezione richiesta
51	primo carattere errato
52	secondo carattere errato

Convalida e verifica: analisi dinamica (18)

Test funzionale: tabelle e grafi

I numeri che identificano le cause e gli effetti sono arbitrari. Nel lucido precedente, il numero 20 identifica una causa intermedia, cioè una combinazione logica di cause primarie.

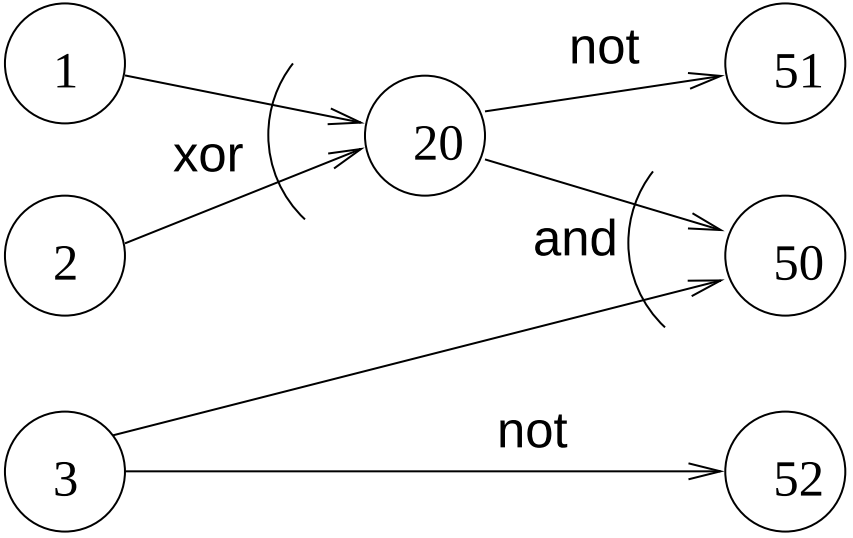
Costruiamo una **tabella cause-effetti** in cui ogni riga è associata ad una causa o un effetto, ed ogni colonna rappresenta nella parte superiore una combinazione di valori logici delle cause, e nella parte inferiore le corrispondenti azioni del sistema.

Cause	combinazioni					
1	1	1	0	0	0	0
2	0	0	0	0	1	1
3	0	1	0	1	0	1
20	1	1	0	0	1	1
Effetti						
50	0	1	0	0	0	1
51	0	0	1	1	0	0
52	1	0	1	0	1	0

Convalida e verifica: analisi dinamica (19)

Test funzionale: tabelle e grafi

Una **tabella cause-effetti** può essere rappresentata da un **grafo cause-effetti**:



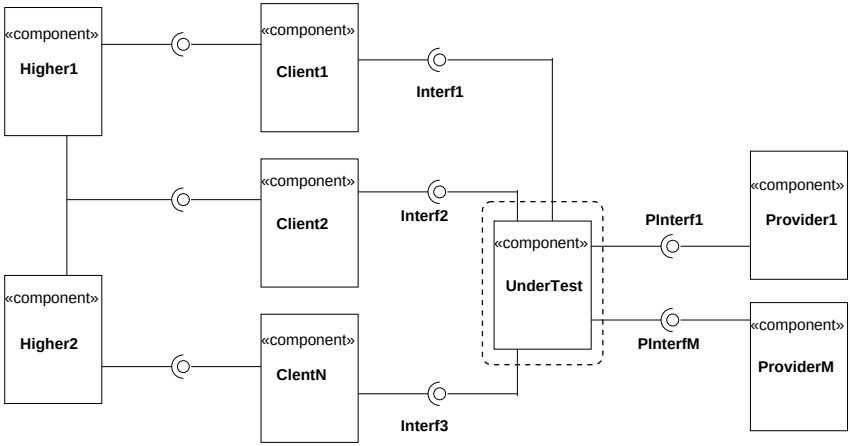
Convalida e verifica: il framework CppUnit (1)

I framework CppUnit e Mockpp servono a facilitare la scrittura di programmi di test, sia nel test di unità che nel test di integrazione.

Esistono altri framework analoghi sia per il C++ che per altri linguaggi.

Il lucido successivo mostra l'architettura di un generico sistema in cui mettiamo in evidenza un componente da collaudare, supponendo che prima di **UnderTest** sia stato implementato e collaudato solo il componente Provider1

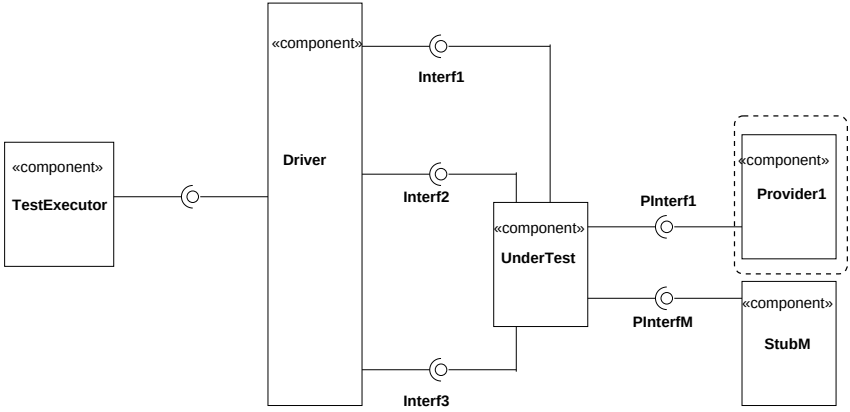
Convalida e verifica: il framework CppUnit (2)



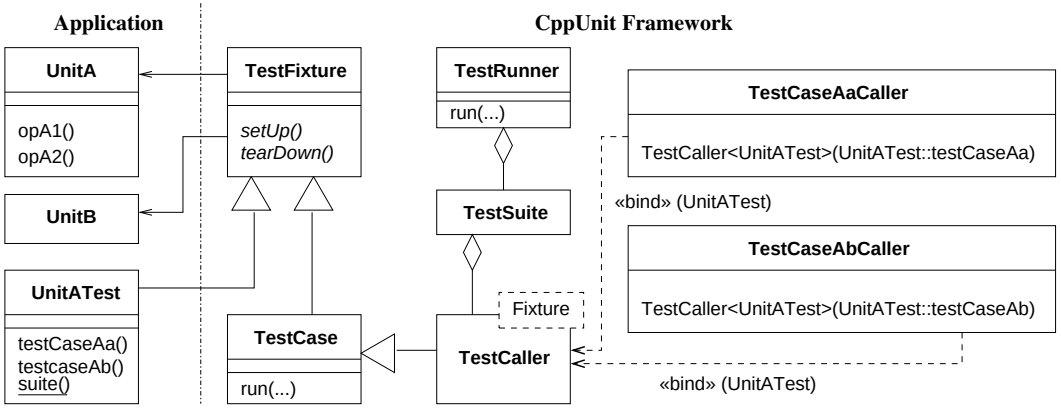
Convalida e verifica: il framework CppUnit (3)

Bisogna realizzare l'infrastruttura di test (*test harness*) che sostituisca, ai fini del test, le parti non ancora implementate.

- CppUnit: per realizzare i **driver** (clienti del modulo da testare);
- Mockpp: per realizzare gli **stub** (fornitori del modulo da testare);



Convalida e verifica: il framework CppUnit (4)



Convalida e verifica: il framework CppUnit (5)

- **UnitA** è la classe da collaudare,
- **UnitB** è una classe che collabora con la precedente, e
- **UnitATest** è il driver.
- **TestFixture** crea ed inizializza le istanze della classe sotto test e le altre classi, e poi le distrugge alla fine del test.
- **TestCaseAaCaller** esegue il caso di test *a*, cioè il metodo testCaseAa() della classe
- **TestCaseAbCaller** analogo al precedente, per il caso si test *b*.
- **TestRunner** è la classe di piú alto livello nel programma di test, quella che coordina le altre.

Convalida e verifica: il framework CppUnit (6)

The tester writes (in the driver class) a method for each test case (`testCaseAa()`, `testCaseAb()`,...). Such methods must:

1. ensure that the test preconditions hold;
2. call one or more operations on the instance(s) of the class under test;
3. verify whether the expected postconditions hold.

The last point relies on predefined assertion macros (`CPPUNIT_ASSERT()`).

Convalida e verifica: il framework CppUnit (7)

```
class Complex {              // --> UnitA under test
  double real;
  double imag;
public:
  Complex(double r, double i = 0) : real(r), imag(i) {};
  friend bool operator==(const Complex& a, const Complex& b);
  friend Complex operator+(const Complex& a, const Complex& b);
};

bool operator==(const Complex &a, const Complex &b)
{
  return a.real == b.real && a.imag == b.imag;
}

Complex operator+(const Complex &a, const Complex &b)
{
  return Complex(a.real, a.imag + b.imag);
}
```

Convalida e verifica: il framework CppUnit (8)

La classe driver è `ComplexTest`:

```
class ComplexTest                       // -> UnitATest
    : public CppUnit::TestFixture  {
private:
    Complex* m_10_1;
    Complex* m_1_1;
    Complex* m_11_2;
public:
    static CppUnit::Test* suite();
    void setUp();
    void tearDown() ;
    void testEquality();                // -> testCaseAa
    void testAddition();                // -> testCaseAb
};
```

Convalida e verifica: il framework CppUnit (9)

```
CppUnit::Test*
ComplexTest::
suite()
{
  CppUnit::TestSuite* suiteOfTests =
      new CppUnit::TestSuite("ComplexTest");
  suiteOfTests->addTest( new          // aggiungi un test
    CppUnit::TestCaller<ComplexTest>( // del driver ComplexTest
      "testEquality",
      &ComplexTest::testEquality)); // per il caso testEquality
  suiteOfTests->addTest(new CppUnit::TestCaller<ComplexTest>(
                                  "testAddition",
                                  &ComplexTest::testAddition));
  return suiteOfTests;
}
```

Convalida e verifica: il framework CppUnit (10)

```
void
ComplexTest::
setUp()
{
    m_10_1 = new Complex(10, 1);     // istanzia dati di test
    m_1_1 = new Complex(1, 1);
    m_11_2 = new Complex(11, 2);
}

void
ComplexTest::
tearDown()
{
    delete m_10_1;                   // dealloca dati di test
    delete m_1_1;
    delete m_11_2;
}
```

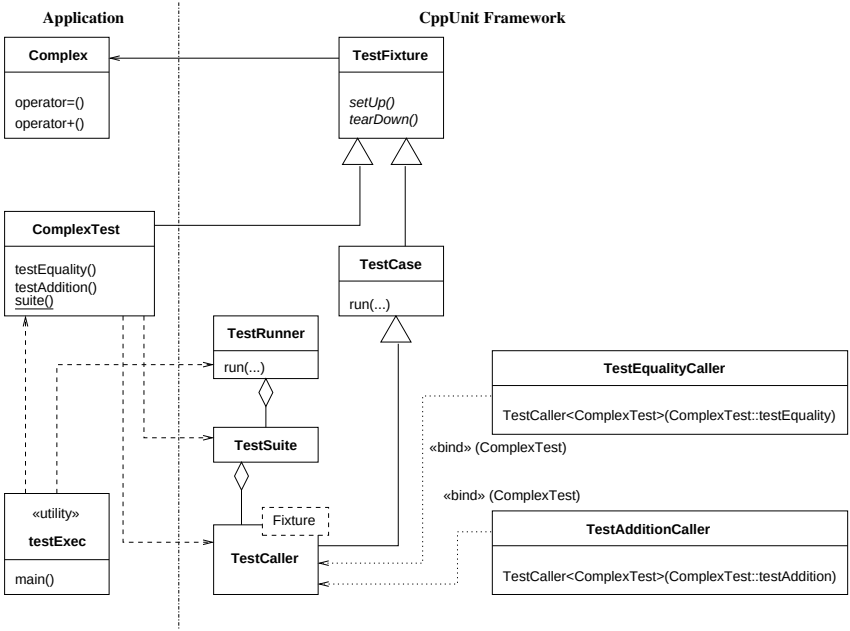
Convalida e verifica: il framework CppUnit (11)

```
void
ComplexTest::
testEquality()
{
    // (10, 1) = (10, 1)
    CPPUNIT_ASSERT(*m_10_1 == *m_10_1);
    // (10, 1) != (11, 2)
    CPPUNIT_ASSERT(!(*m_10_1 == *m_11_2));
}

void
ComplexTest::
testAddition()
{
    // (10, 1) + (1, 1) = (11, 2)
    CPPUNIT_ASSERT(*m_10_1 + *m_1_1 == *m_11_2);
}
```

Convalida e verifica: il framework CppUnit (12)

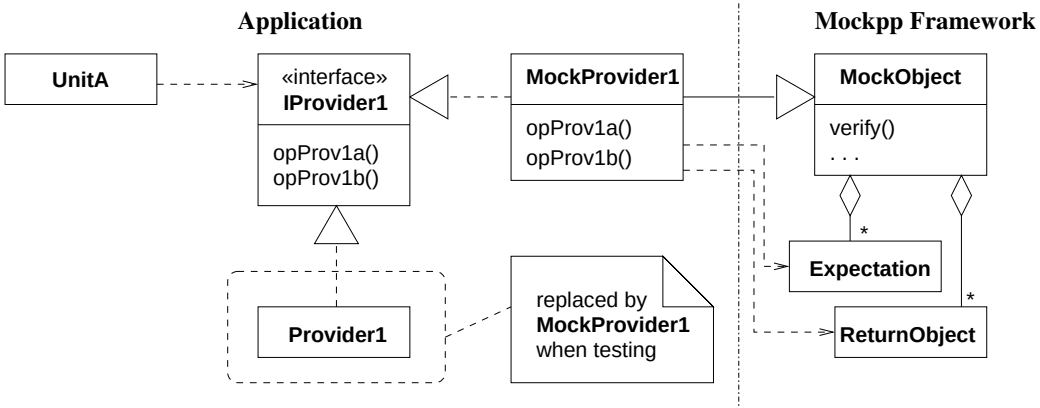
```
int
main()
{
    CppUnit::TextTestRunner runner;
    runner.addTest(ComplexTest::suite());
    runner.run();
    return 0;
}
```

Convalida e verifica: il framework CppUnit (13)



Convalida e verifica: il framework Mockpp (1)

Il framework Mockpp si basa su una versione del pattern *Adapter*.



Convalida e verifica: il framework Mockpp (2)

- **UnitA**: the class under test.
- **IProvider1**: interface of component Provider1.
- **Provider1**: a provider of the class under test.
- **MockProvider1**: a *mock object* replacing Provider1.
- **MockObject**: the base for mock objects.
- **Expectation**: base class for containers of values exepected to be passed to Provider1 methods by UnitA methods, and possibly constraints on the same values.

Convalida e verifica: il framework Mockpp (3)

- A mock object can be seen as a *smart stub*, as it can verify the correct invocation of the (provider) operations it simulates.
- *Expectations* are the simulation and verification engine of the mock object.
- They are containers (lists, sets,...) that are filled with values (or constraints) when the mock object is initialized.
- The mock object methods simulate the real ones by passing their actual arguments (or other values computed at run-time) to the expectation objects.
- The expectation objects may perform several types of checks, such as comparing actual with expected values, checking their time ordering, verifying arithmetics constraints...
- An exception is raised if a check fails.

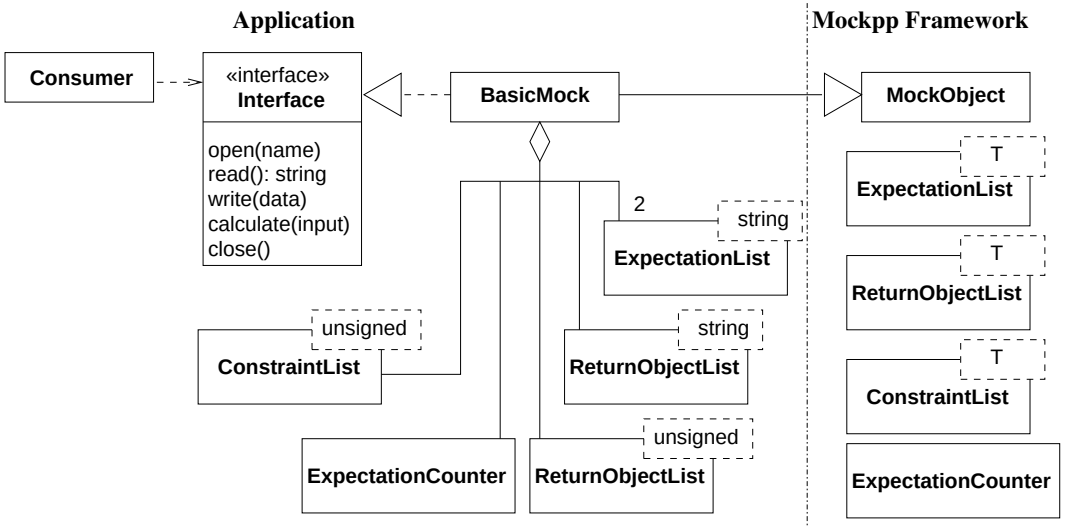
Convalida e verifica: il framework Mockpp (4)

Un'istanza di `Consumer` apre un file di configurazione, legge tre righe e chiude il file, poi compie delle elaborazioni.

Queste operazioni vengono eseguite da una classe (ancora da realizzare) che realizzi l'interfaccia `Interface`.

Vogliamo verificare che `Consumer` usi correttamente le operazioni di una classe che realizzi `Interface`.

Convalida e verifica: il framework Mockpp (5)



Convalida e verifica: il framework Mockpp (6)

```
class Interface
{ public:
    virtual void open(const std::string &name) = 0;
    virtual std::string read() = 0;
    virtual std write(const std::string &s) = 0;
    virtual unsigned calculate(unsigned input) = 0;
    virtual void close() = 0;
};

class Consumer {
    Interface* configfile;
    std::string config1, config2, config3;
public:
    Consumer(Interface* intf) : configfile(intf) {};
    void load();
    void process();
    void save();
};
```

Convalida e verifica: il framework Mockpp (7)

```
void Consumer::load()
{   configfile->open("file1.lst");
    config1 = configfile->read();
    config2 = configfile->read();
    config3 = configfile->read();
    configfile->close();
}

void Consumer::process()
{
  unsigned x = configfile->calculate(1);
  x += configfile->calculate(2);
  x =+ configfile->calculate(3);

  config1 += "/processed";
  config2 += "/processed";
  config3 += "/processed";
}
```

Convalida e verifica: il framework Mockpp (8)

```
void Consumer::save()
{
  configfile->open("file1.lst");
  configfile->write(config1);
  configfile->write(config2);
  configfile->write(config3);
  configfile->close();
}
```

Convalida e verifica: il framework Mockpp (9)

```
class BasicMock : public Interface, public mockpp::MockObject {
public:
  mockpp::ExpectationList<std::string>  open_name;
  mockpp::ExpectationCounter            close_counter;
  mockpp::ExpectationList<std::string>  write_data;
  mockpp::ReturnObjectList<std::string> read_data;
  mockpp::ReturnObjectList<unsigned>    calculate_output;
  mockpp::ConstraintList<unsigned >     calculate_input;

  BasicMock() { /* inizializzazioni */ };
  virtual void open(const std::string &name);
  virtual std::string read();
  virtual void write(const std::string &s);
  virtual unsigned calculate(unsigned input);
  virtual void close();
};
```

Convalida e verifica: il framework Mockpp (10)

```
void BasicMock::open(const std::string& name)
{ open_name.addActual(name); }

std::string BasicMock::read()
{ return read_data.nextReturnObject(); }

void BasicMock::write(const std::string &s)
{ write_data.addActual(s); }

unsigned BasicMock::calculate(unsigned input)
{ calculate_input.addActual(input);
  return calculate_output.nextReturnObject(); }

void BasicMock::close()
{ close_counter.inc(); }
```

Convalida e verifica: il framework Mockpp (11)

```
int main(int argc, char **argv)
{ try {
    BasicMock mock;
    mock.open_name.addExpected("file1.lst");       // expctn
    mock.open_name.addExpected("file1.lst");       // expctn
    mock.close_counter.setExpected(2);             //   "
    mock.read_data.addObjectToReturn("record-1"); // return
    mock.read_data.addObjectToReturn("record-2"); //   "
    mock.read_data.addObjectToReturn("record-3"); //   "
    mock.write_data.addExpected("record-1/processed");
    mock.write_data.addExpected("record-2/processed");
    mock.write_data.addExpected("record-3/processed");
    mock.calculate_output.addObjectToReturn(10);
    mock.calculate_output.addObjectToReturn(20);
    mock.calculate_output.addObjectToReturn(30);
```

Convalida e verifica: il framework Mockpp (12)

```
    Consumer consumer(&mock);                          // esecuzione
    consumer.load();                                   //    "
    consumer.process();                                //    "
    consumer.save();                                   //    "
    mock.verify();                                     // controllo
    std::cout << "Test eseguito con successo" << std::endl;
  } catch(std::exception &ex) {
    std::cout << "Errori.\n" << ex.what() << std::endl
    return 1;
  }
  return 0;
}
```

Convalida e verifica: CppUnit e Mockpp (1)

Consideriamo un caso in cui il modulo sotto test richieda sia un driver che uno stub, per esempio:

```
class Calculator
{
   int base_
   IFile* f_;
public:
    Calculator (int in_base, IFile* f)
        : base_(in_base), f_(f) {};

    int add(int num);  // add num to base_
    int sub(int num);  // subtract num from base_
    int filesub();     // subtract value read from f_
};
```

Convalida e verifica: CppUnit e Mockpp (2)

```
int
Calculator::
add(int num)
{
      return base_ + num;
}
int
Calculator::
sub(int num)
{
      return base_ - num;
}
int
Calculator::
filesub()
{
      return base_ - f_->read();
}
```

Convalida e verifica: CppUnit e Mockpp (3)

Lato stub

```
class IFile {
public:
     virtual int read() = 0;
};

class FileStub : public IFile, public mockpp::MockObject {
public:
    mockpp::ReturnObjectList<int> read_data;
    mockpp::ExpectationList<int>  input_data;
    mockpp::ExpectationCounter    filesub_counter;
    FileStub()
        : mockpp::MockObject(MOCKPP_PCHAR("FileStub"), 0)
        , input_data(MOCKPP_PCHAR("FileStub/input_data"), this)
        , read_data(MOCKPP_PCHAR("FileStub/read_data"), this)
        , filesub_counter(MOCKPP_PCHAR("FileStub/close_counter"),
    { }
    virtual int read();
}
```

Convalida e verifica: CppUnit e Mockpp (4)

Lato stub

```
int
FileStub::
read()
{
    return read_data.nextReturnObject();
}
```

Convalida e verifica: CppUnit e Mockpp (5)

Lato driver

```
class Calculator_test
        : public CppUnit::TestFixture
{
     int b_;
     FileStub* f_;
     Calculator* c_;
public:
     static CppUnit::Test* suite();
     void setUp();
     void setfile(IFile* f);
     void tearDown();

     void test_add();
     void test_sub();
     void test_filesub();
};
```

Convalida e verifica: CppUnit e Mockpp (6)

Lato driver

```
void Calculator_test::
setUp()
{    b_ = 100;
     f_ = new FileStub();
     c_ = new Calculator(b_, f_);
     f_->read_data.addObjectToReturn(10);
     f_->read_data.addObjectToReturn(20);
     f_->read_data.addObjectToReturn(30);
     f_->input_data.addExpected(90);
     f_->input_data.addExpected(80);
     f_->input_data.addExpected(70);
}
```

Convalida e verifica: CppUnit e Mockpp (7)

Lato driver

```
void Calculator_test::
test_add()
{ CPPUNIT_ASSERT(123 == c_->add(23));
}

void Calculator_test::
test_sub()
{ CPPUNIT_ASSERT(78 == c_->sub(22));
}

void Calculator_test::
test_filesub()
{ CPPUNIT_ASSERT(90 == c_->filesub());
  CPPUNIT_ASSERT(80 == c_->filesub());
  CPPUNIT_ASSERT(70 == c_->filesub());
}
```

Convalida e verifica: CppUnit e Mockpp (8)

Lato driver

```
int main(int argc, char ** /*argv*/)
{
  CppUnit::TextTestRunner runner;
  runner.addTest(Calculator_test::suite());

  try {
        runner.run();
  } catch(std::exception &ex) {
        std::cout << std::endl
            << "Error occured.\n" << ex.what() << std::endl
            << std::endl;
        return 1;
  }
  return 0;
}
```

Convalida e verifica: test di integrazione

Si testa l'interfacciamento fra moduli il cui funzionamento è già stato testato individualmente. È possibile eseguire il test di integrazione dopo che tutti i moduli sono stati testati individualmente, assemblandoli e provando il sistema completo (*big-bang test*).

In genere si preferisce una strategia incrementale del test di integrazione, che permette di integrare i moduli man mano che vengono completati e che superano il test di unità. In questo modo gli errori di interfacciamento vengono scoperti prima ed in modo piú localizzato, e si risparmia sul numero di driver e di stub che devono essere sviluppati.

Convalida e verifica: i test di sistema

- test di stress
- test di robustezza
- test di regressione
- test di accettazione
- ...

Convalida e verifica: i test in grande (esempio) (1)

Test dello Storage Resource Manager (SRM)
http://www.ing.unipi.it/~a009435/issw/extra/ewdc09.pdf

A *Storage Element* (SE) is a Grid Service that provides:

- A mass storage system.
- A GridFTP data-transfer service between the SE and the Grid.
- Local POSIX-like input/output calls for application access to SE data.
- Authentication, authorization and audit/accounting facilities.

The SRM is a common interface for different Storage Elements.
The SRM specification defines many service requests:

- *Space management* functions to reserve, allocate, release, and manage **storage spaces**, their types and lifetimes.
- *Data transfer* functions to store and retrieve files into and from SRM spaces either from the client's space or from other remote storage systems.
- Other classes: *Directory*, *Permission*, and *Discovery* functions.

Convalida e verifica: i test in grande (esempio) (2)

The goals of testing are:

- Validating the **SRM interface and protocol specification** for adherence to the explicit and implicit **user requirements**, and against inconsistency, incompleteness, or inefficiency;
- validating the **SRM implementations** for compliance with the specification;
- checking the SRM implementations for performance and reliability.

Difficulties arise from:

- Large and complex set of service requests,
- informal specification,
- number of different implementations, and
- number of sites.

Convalida e verifica: i test in grande (esempio) (3)

A typical SRM request: srmReserveSpace

Input parameters:

TRetentionPolicyInfo	**retentionPolicyInfo** (REPLICA, OUTPUT, or CUSTODIAL)
unsigned long	**desiredSizeOfGuaranteedSpace**
string	authorizationID
unsigned long	desiredSizeOfTotalSpace
int	desiredLifetimeOfReservedSpace
TTransferParameters	transferParameters

. . . *more optional parameters, nine in total*

Output parameters:

TReturnStatus	**returnStatus**
string	requestToken

. . . *more optional parameters*

Convalida e verifica: i test in grande (esempio) (4)

tabelle e grafi cause-effetti

operating conditions, e.g.:
Causes:

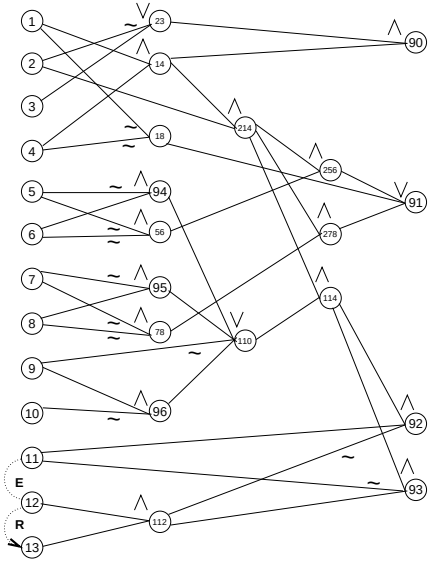
1	retentionPolicyInfo is not NULL
2	**retentionPolicyInfo is supported by server**
	...
11	requestToken is returned **[11 and 12 mutually exclusive]**
12	spaceToken is returned **[12 requires 13]**
13	sizeOfGuaranteedReservedSpace and lifetimeOfReservedSpace are returned
	...

Effects:

94	sizeOfGuaranteedReservedSpace = default
95	lifetimeOfReservedSpace = default
96	transferParameters is ignored

Convalida e verifica: i test in grande (esempio) (5)

tabelle e grafi cause-effetti

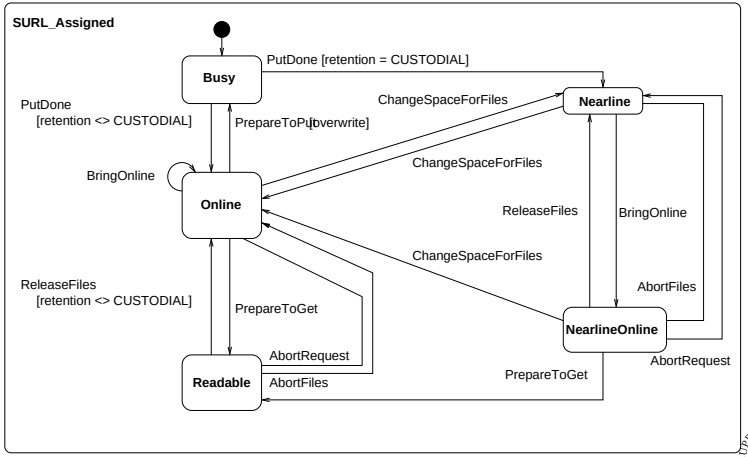


Convalida e verifica: i test in grande (esempio) (6)

Error guessing

Error guessing = pragmatic knowledge + formalization.

Example: formalization by state machines led to discover unexpected interactions.



Partial state machine for a file.

Convalida e verifica: i test in grande (esempio) (7)

test cases

Five families of test cases have been designed:

Availability	to check the availability in time of the SRM service end-points.
Basic	to verify basic functionality of the implemented SRM APIs.
Use Cases	to check boundary conditions, use cases derived by real usage, function interactions, exceptions, etc.
Exhaustion	to check “extreme” values and properties of input and output arguments such as length of filenames, URL format, etc.
Stress tests	to stress the systems, identify race conditions, study the behavior of the system when critical concurrent operations are performed, etc.

Convalida e verifica: i test in grande (esempio) (8)

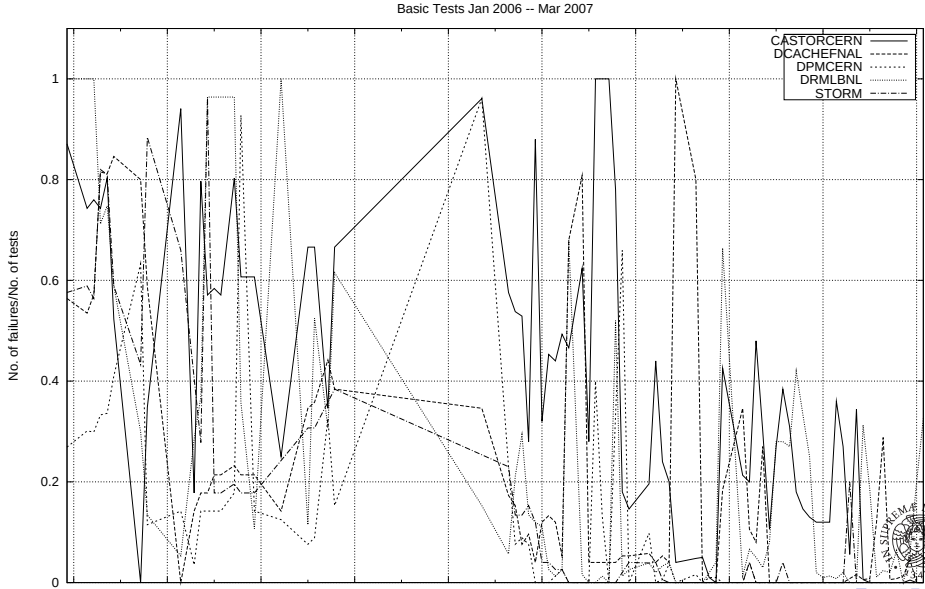
Test execution and analysis

- Execution framework based on S2 test scripts and shell scripts.
  - invoke SRM requests;
  - make checks on return codes;
  - define complex test actions.
- Automatic execution and result logging six times a day.
- Monthly plots for each test family.

Convalida e verifica: i test in grande (esempio) (9)

Test execution and analysis

Percent of failures for five storage sites (2 $\times$ CERN, FNAL, LBNL, CNAF)



Convalida e verifica: il linguaggio TTCN-3 (1)

Il linguaggio *Test and Testing Control Notation 3* (TTCN-3) è stato concepito espressamente per l'esecuzione di test, è orientato al test di sistemi reattivi, e viene usato soprattutto nel settore delle telecomunicazioni per eseguire il *test di conformità* (*conformance*), che verifica il rispetto di determinati standard, e in particolare dei protocolli di comunicazione.

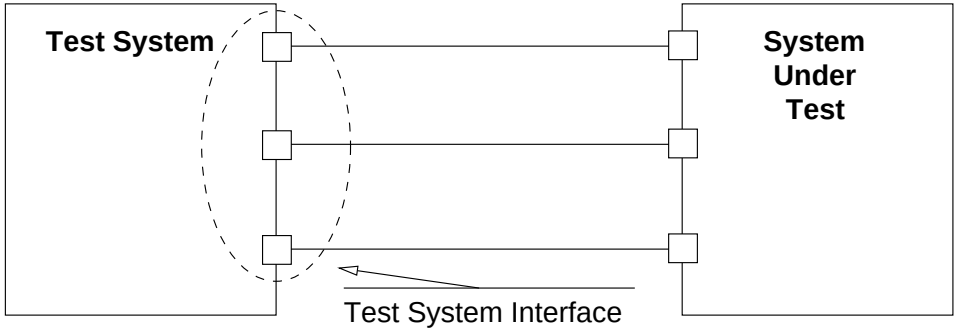
Il TTCN-3 è comunque applicabile a diversi tipi di test e di applicazioni.

Il linguaggio permette di definire un **sistema di test** formato da uno o piú **componenti** che scambiano messaggi o chiamate di procedure col sistema sotto test.

Un componente può rappresentare sia un cliente che un fornitore del sistema sotto test.

Un `testcase` è una procedura che usa i componenti per inviare e ricevere i messaggi o le chiamate di procedura per un determinato caso di test.

Convalida e verifica: il linguaggio TTCN-3 (2)

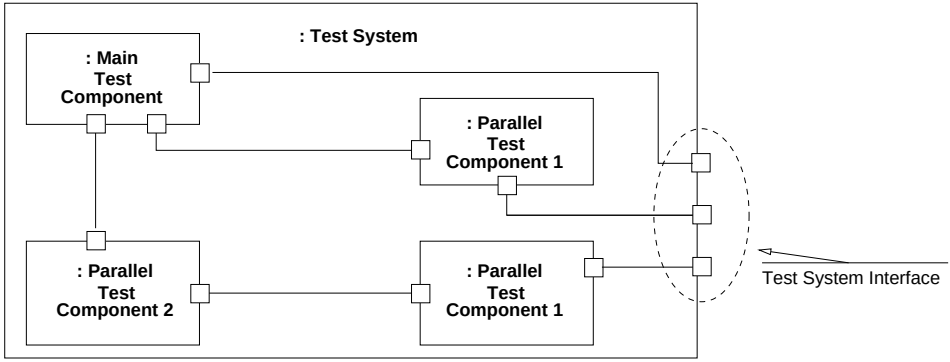


Il sistema sotto test può essere un sistema sw o un sistema fisico (p.es., una rete di comunicazione o un dispositivo).

Il sistema di test viene eseguito da un ambiente TTCN-3 che provvede a stabilire il collegamento col sistema sotto test.

L'interfaccia del sistema di test è definito da uno o piú **port**.

Convalida e verifica: il linguaggio TTCN-3 (3)



Il sistema di test contiene un **Main Testing Component** (**MTC**) e, opzionalmente, dei **Parallel Testing Component** (**PTC**).

I collegamenti fra componenti interni al sistema di test vengono creati con clausole `connect`, quelli fra componenti interni e port dell'interfaccia esterno vengono creati con clausole `map`.

Convalida e verifica: il linguaggio TTCN-3 (4)

Elementi caratteristici del linguaggio

- costrutto `testcase`;
- comunicazioni **sincrone** (**procedure**) e **asincrone** (**messaggi**);
- operazioni per comunicazioni asincrone (`send`, `receive` e molte altre);
- sitema di tipi simile ad ASN.1, adatto a sistemi di TLC;
- possibilità di specificare i formati dei messaggi usando **template** ed **espressioni regolari**;
- istruzioni per creare e gestire **timer**;
- istruzioni `alt` e `interleave` per rappresentare, rispettivamente, comportamenti alternativi o concorrenti;
- tipo enumerato `verdict` per rappresentare l'esito dei test (**none**, **pass**, **fail**, **inconc**, **error**);
- possibilità di definire **dinamicamente** (cioè a tempo di esecuzione) la configurazione del sistema di test (clausole `connect` e `map`).

N.B.: i template del TTCN-3 sono diversi da quelli del C++.

Convalida e verifica: il linguaggio TTCN-3 (5)

Componenti
Un componente TTCN-3 è definito da un insieme di *port* (analoghi a quelli visti per i componenti UML), a loro volta caratterizzati dal tipo dei messaggi trasmessi e dalla loro direzione (*in*, *out*, *inout*).

Un componente può dichiarare variabli, timer, etc.

```
type port DNSPort message {   // manda e riceve messaggi
    inout DNSQuery;           // asincroni di tipo
}                             // DNSQuery

type component DNSTester {    // un DNSTester ha
    port DNSPort P;           // un port di tipo DNSPort
    timer t := 3.0;           // e un timer di 3 s
}
```

Convalida e verifica: il linguaggio TTCN-3 (6)

Casi di test

A test case is [a] complete and independent specification of the actions required to achieve a specific test purpose

– ETSI ES 201 873-1

Un testcase è una procedura che esegue un test usando i port ed altre eventuali risorse dichiarati da un tipo di componente.

```
// "query" e "reply" sono due template di messaggi
testcase tc_testcase()
        runs on DNSTester {                  // usa DNSTester
    P.send(query);                           // invia richiesta
    t.start;                                 // avvia timer
    alt {                                    // eventi alternativi
    [] P.receive(reply) setverdict(pass); // risp. giusta
    [] P.receive setverdict(fail);           //  " sbagliata
    [] t.timeout setverdict(inconc);         // non pervenuta
    }
    stop;
}
```

Convalida e verifica: il linguaggio TTCN-3 (7)

Moduli
Un modulo ha una parte **dichiarativa** ed una **di controllo** (opzionale).

Nella parte dichiarativa si definiscono tipi di dati, tipi di port, tipi di componenti, template, casi di test, funzioni, etc.

Nella parte di controllo si eseguono i casi di test, invocati con l'istruzione `execute`. Si possono usare istruzioni complesse come *if*, *for*, etc.

Convalida e verifica: il linguaggio TTCN-3 (8)

Esempio: server DNS

```
(c) Copyright Wiley & Sons 2005

author: Colin Willcock, Thomas Deiß, Stephan Tobies,
         Stefan Keil, Federico Engler, Stephan Schulz
desc:   This is a strongly simplified Domain Name Server (DNS)
         test suite for testing some basic domain name
         resolution behaviour.
remark: This TTCN-3 code is based on the DNS example code
         presented in "C. Willock et al., An Introduction
         to TTCN-3, Wiley & Sons, 2005.  ISBN: 0-470-01224-2"
         This copyright notice shall not be removed in copies
         of this file.
```

Convalida e verifica: il linguaggio TTCN-3 (9)

Esempio: server DNS

```
module DNS {
  // tipi base
  type integer Identification( 0..65535 ); // 16-bit integer
  type enumerated MessageKind {e_Question, e_Answer};
  type charstring Question;
  type charstring Answer;

  // struttura generale dei messaggi
  type record DNSMessage {
    Identification identification,
    MessageKind messageKind,
    Question question,
    Answer answer optional      // campo opzionale
  }
```

Convalida e verifica: il linguaggio TTCN-3 (10)

Esempio: server DNS

```
template DNSMessage         // modello dei msg di richiesta
   a_DNSQuestion(Identification p_id, Question e_question) :=
{ identification := p_id,
  messageKind := e_Question,
  question := e_question,   // campo per hostname
  answer := omit            // non c'e' il campo della risposta
}
template DNSMessage         // modello dei msg di risposta
   a_DNSAnswer(Identification p_id, Answer e_answer) :=
{ identification := p_id,
  messageKind := e_Answer,
  question := ?,            // qualsiasi valore di tipo Question
  answer := e_answer        // campo per indirizzo IP
}
```

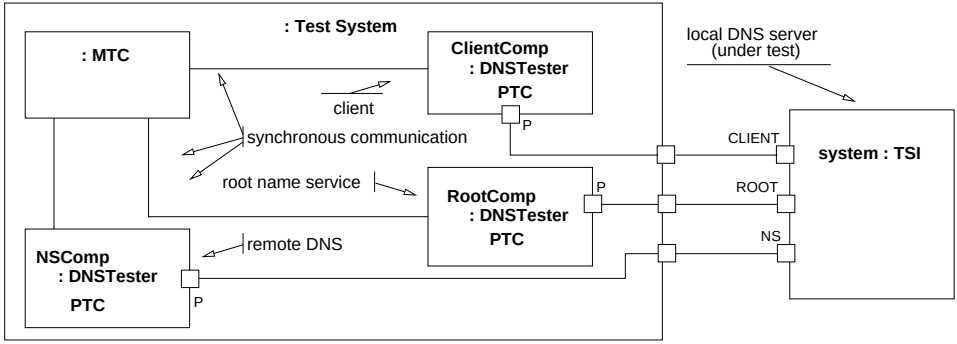
Convalida e verifica: il linguaggio TTCN-3 (11)

Esempio: server DNS

```
  type port DNSPort message { ... }
  type component DNSTester { ... }
  var query := a_DNSQuestion(12345, "www.research.nokia.com");
  var reply := a_DNSAnswer(12345, "172.21.56.98");
  testcase tc_testcase() runs on DNSTester { ... }

  control {
    execute(tc_testcase());
  }
}   // end module DNS
```

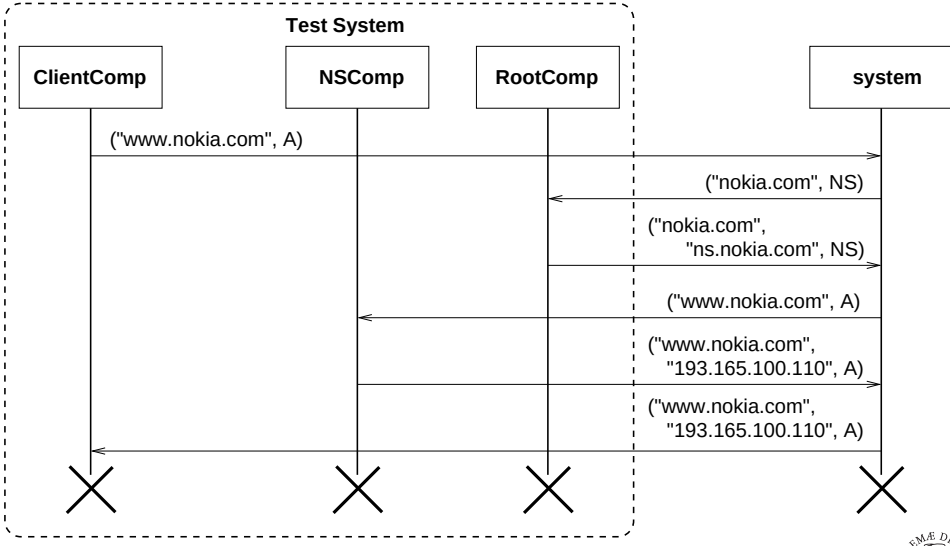
Convalida e verifica: il linguaggio TTCN-3 (12)

Esempio: server DNS con PTC



Convalida e verifica: il linguaggio TTCN-3 (13)

Esempio: server DNS con PTC



(from C. Willcock, *Introduction to TTCN-3 (14)*, http://www.ttcn-3.org/index.php/learn/tutorials, redrawn)

Convalida e verifica: il linguaggio TTCN-3 (14)

Esempio: server DNS con PTC

`DNSPort` e `DNSTester` come nei lucidi precedenti, le altre definizioni non vengono mostrate.

```
type component TSI {  // test system interface
    port DNSPort CLIENT,
    port DNSPort ROOT,
    port DNSPort NS
}

function RootBehaviour() runs on DNSTester {
    alt {
    [] P.receive(rootquery) {
        P.send(rootanswer);
        setverdict(pass);
       }
    [] P.receive { setverdict(fail);}
    }
    stop;
}
```

Convalida e verifica: il linguaggio TTCN-3 (15)

Esempio: server DNS con PTC

```
testcase Testcase3() runs on MTC system TSI {
  var DNSTester RootComp, NSComp, ClientComp;
  RootComp := DNSTester.create;      // crea componenti
  NSComp := DNSTester.create;        //              "
  ClientComp := DNSTester.create;    //              "
  map(RootComp:P, system:ROOT);      // crea collegamenti
  map(NSComp:P, system:NS);          //              "
  map(ClientComp:P, system:CLIENT); //               "
  RootComp.start(RootBehaviour());     // attivazione
  NSComp.start(NSBehaviour());         //             "
  ClientComp.start(ClientBehaviour()); //             "
  ClientComp.done;                   // attesa fine di ClientComp
  stop;                              // fine del testcase
}
```

CONCLUSIONI