INGEGNERIA PER LA SOSTENIBILITA' INDUSTRIALE - UnivPM

Fondamenti di Informatica

Prof. Ing. Loris Penserini, PhD

elpense@gmail.com

https://orcid.org/0009-0008-6157-0396

Materiale:

https://github.com/penserini/Lezioni_UnivPM.git

Esempio: Algoritmo di Euclide per il MCD

L'algoritmo di Euclide è un metodo efficiente per trovare il Massimo Comun Divisore (MCD) di due numeri interi, basato sulla proprietà che il MCD di due numeri è uguale al MCD di uno dei numeri e del loro resto, fino ad arrivare a un resto nullo; l'ultimo resto non nullo è il MCD cercato. Si esegue tramite divisioni successive, invece che sottrazioni, per velocizzare il processo, finché il resto non diventa zero.

Come funziona (Metodo delle Divisioni)

1. Dividi il numero maggiore per il numero minore e trova il resto: la formula è: **a = b * q + r**
2. Se il resto (**r**) è 0, il numero minore (**b**) è il MCD
3. Se il resto non è 0, sostituisci **a** con **b** e **b** con **r**, e **ripeti** il processo di divisione
4. Continua finché non ottieni un resto nullo
5. L'ultimo resto non nullo è il MCD.

Project Work: Applicazione di un Algoritmo

Utilizzando l'algoritmo precedente, calcolate il MCD di 96 e 36: risolvere il problema ragionando in termini algoritmici.

PW-1: Soluzione

Problema:

Utilizzando l'algoritmo precedente, calcoliamo il MCD di 96 e 36

Soluzione (vedi algoritmo del MCD):

96 = 36 * 2 + 24 (resto = 24)

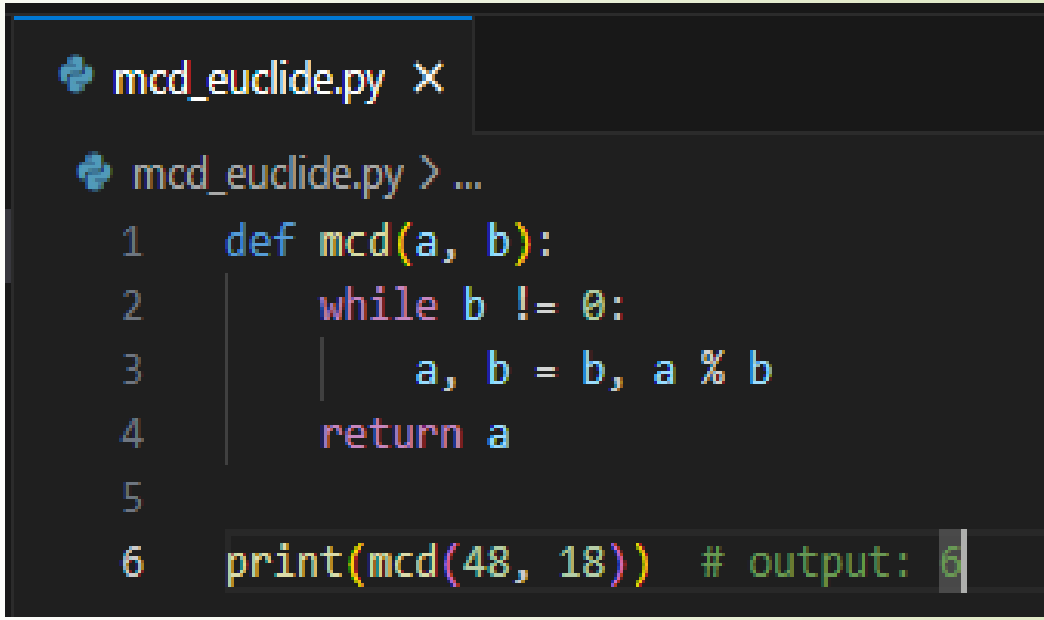
36 = 24 * 1 + 12 (resto = 12)

24 = 12 * 2 + 0 (resto = 0)

L'ultimo resto non nullo è 12. Quindi, il MCD(96, 36) = 12

Programma

Implementazione dell'algoritmo del MCD in PYTHON:



L'istruzione della riga n.3 va letta così:

(a, b) ← (b, a % b)

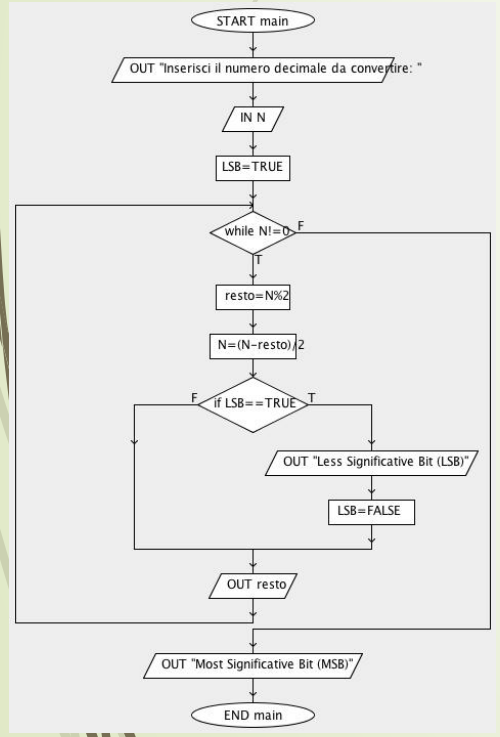
Cioè, "scompatta" i valori e li assegna a sinistra:

- **a** prende il valore di **b**
- **b** prende il valore di **a % b** (il resto della divisione intera)

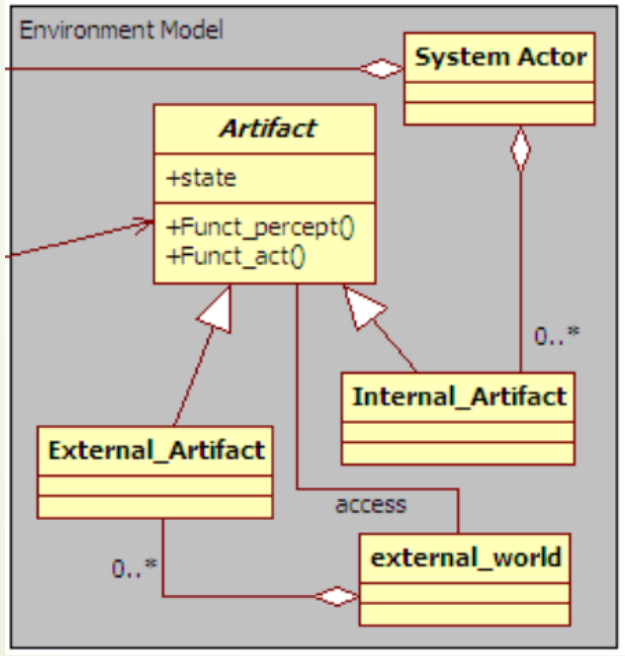
Paradigmi di progettazione…

Alcuni dei principali paradigmi di progettazione degli algoritmi e relativi linguaggi di programmazione.

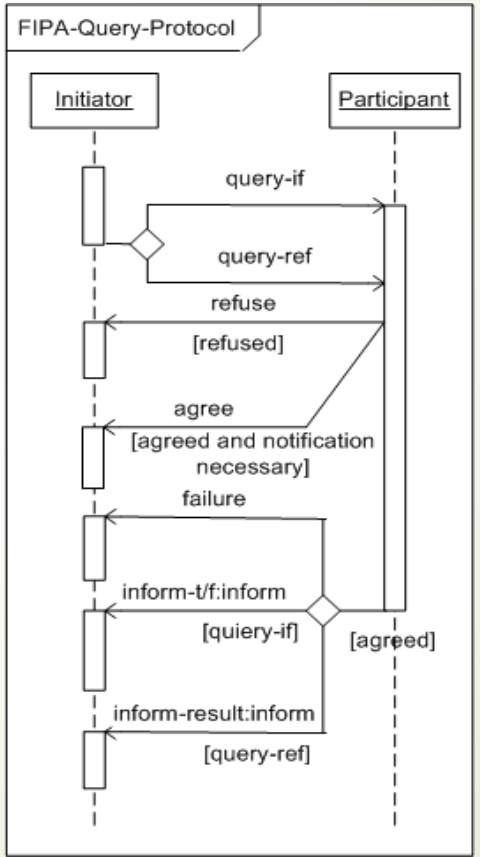
Flow-Chart (Structured Prog.)



UML (Object Oriented Prog.)



Agent-UML (Agent Oriented Prog.)



Scratch/Blockly (Block based Prog.)

Le fasi del ciclo di vita del software

Nell'ingegneria del software si spiegano le fasi di sviluppo di un'applicazione software come fosse un prodotto con un suo «ciclo di vita».

Il modello più noto di sviluppo del software è il «**modello a cascata**» (*waterfall model*) in cui ci sono una serie di fasi a cascata e ciascuna fase riceve ingressi dalla fase precedente e produce uscite, che sono a loro volta ingressi per la fase seguente.

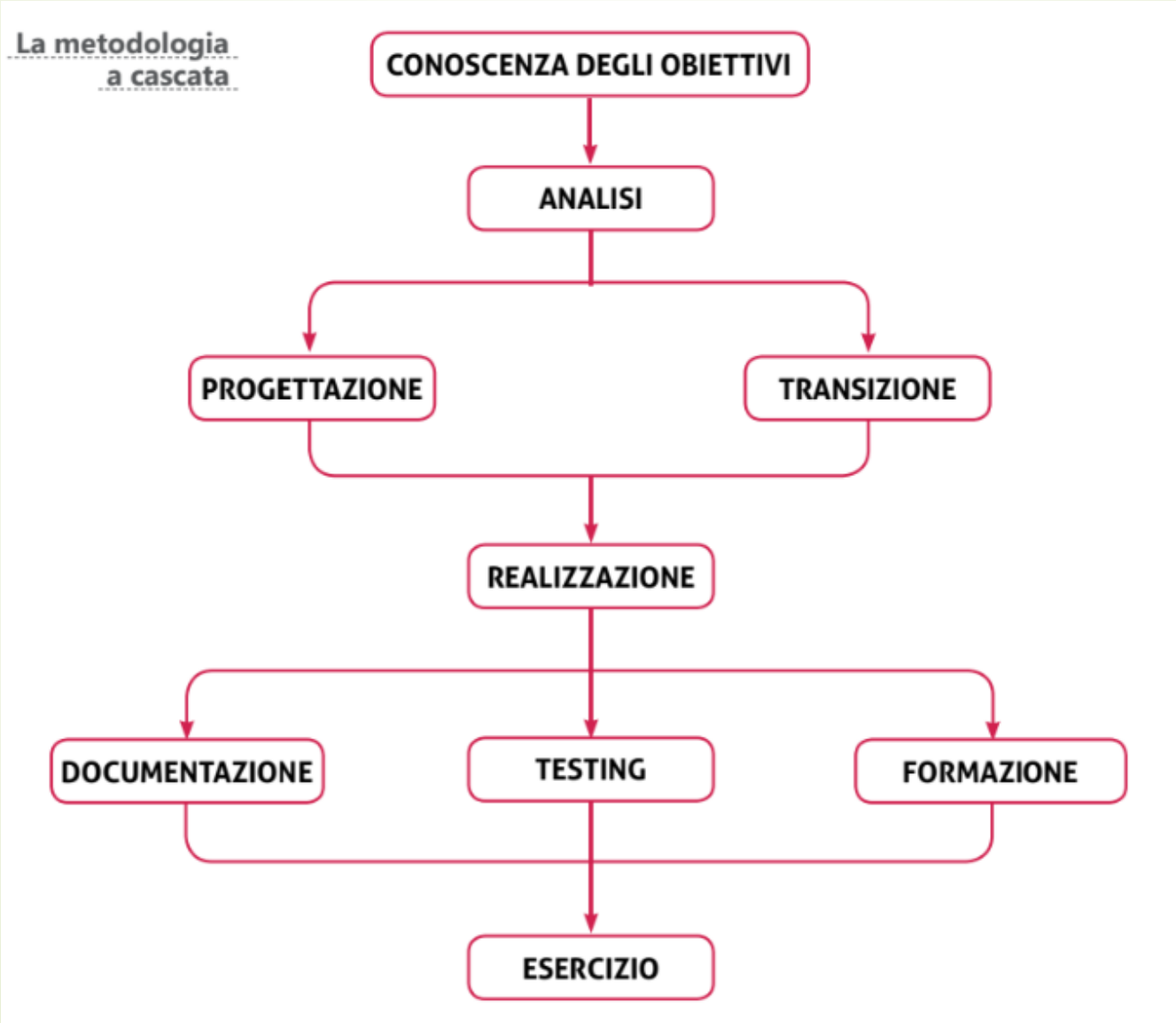
Altri **modelli** si sono evoluti nel tempo:

Spirale: prevedono la riesecuzione o il riciclo delle fasi con l'intento di raffinare, in modo crescente, i risultati di ogni fase.

Prototipale: punta direttamente alla fase di realizzazione di prototipi come simulazioni del prodotto finale, ma non completi nelle funzionalità, che possano essere sottoposti, a livello intermedio, alla valutazione del cliente.

Rilasci Incrementali: il prodotto si avvicina progressivamente alla versione definitiva con successivi interventi migliorativi nelle funzionalità e con continui raffinamenti delle specifiche.

Modello a Cascata



Paradigmi per Sistemi Complessi

Il software di oggi è più complesso che in passato, perché gli si richiede di funzionare in modo autonomo in ambienti dinamici, aperti e imprevedibili. Le metodologie di ingegneria del software che si ispirano ai sistemi autonomi o *self-adaptive* (*SAS*), offrono una soluzione promettente per far fronte al problema di sviluppo di sistemi complessi.

Uno dei paradigmi di programmazione più recente è quello orientato agli agenti: **AOP** (*Agent Oriented Programming*) e il relativo **AOSE** (*Agent Oriented Software Engineering*):

- **AOP: Jade**, **Jadex**, **Jake**, **3APL**, **Alan**, **JEAP**, **…**

[Morandini et al., 2008] [Penserini et al., 2007b] [Pagliarecci et al., 2007] [Panti et al., 2003]

- **AOSE: MaSE**, **GAIA**, **Tropos4AS**,**...**

[Penserini et al., 2007a] [Morandini et al., 2009]

Le fasi del «ciclo di vita» nella metodologia Tropos4AS sono un ibrido tra mod. trasformazionale (adotta approccio MDA) e mod. a spirale (si reitera sulle fasi precedenti) [Morandini et al., 2017]

Dal Sorgente al Programma Eseguibile

IL CALCOLATORE NON E' IN GRADO DI COMPRENDERE DIRETTAMENTE UN LINGUAGGIO AD ALTO LIVELLO, cioè un linguaggio utile a semplificare il lavoro del programmatore.

Per cui, il testo del programma scritto in un linguaggio di programmazione di alto livello, detto programma **sorgente** (source), deve essere tradotto in **linguaggio macchina** per poter essere eseguito dall'elaboratore.

I modelli dei Large Language Model (LLM), che adottano tecniche di AI generativa e Deep Learning, cambiano la visione dell'informatica tradizionale, aggiungendo un nuovo capitolo alla Scienza Informatica... Ci ritorneremo più avanti!

Assembly

Questo è il linguaggio con il quale si comunica con lo specifico processore del PC in uso. Per esempio, la famiglia dei Motorola (es. MC68000, …) e la famiglia degli INTEL (es. 8086, …, 80386, …, Pentium, ecc.)

Esempio:

```
if (D0 == 5)
       D1++;
D2 = D0;
```

Linguaggio di Alto Livello (C, C++, Java, …)

```
        CMPI.L  #5,D0
        BNE     SKIP
        ADDQ.L  #1,D1
SKIP    MOVE.L  D0,D2
```

Assembly del Processore Motorola 68000

Assembly

```
        CMPI.L  #5,D0
        BNE     SKIP
        ADDQ.L  #1,D1
SKIP    MOVE.L  D0,D2
```

Assembly del Processore Motorola 68000

Dettagli istruzione per istruzione

- **CMPI.L #5, D0**
  Confronta il *longword* in D0 con l'immediato 5. Non modifica D0, ma imposta i **flag** del CCR come una sottrazione D0 - 5:
  - Z (Zero) = 1 se D0 == 5
  - N, V, C impostati secondo il risultato della sottrazione (qui interessa soprattutto Z)
- **BNE SKIP** (*Branch if Not Equal*)
  Salta all'etichetta SKIP se Z = 0, cioè se D0 ≠ 5.
- **ADDQ.L #1, D1** (*Add Quick*)
  Esegue D1 = D1 + 1 solo quando **non** si è saltato (quindi solo se D0 == 5).
  Aggiorna i flag, ma qui non vengono poi usati.
- **MOVE.L D0, D2**
  Copia D0 in D2 sempre.

Tradurre il Programma

Il processo di traduzione può essere di due tipologie:

Interpretazione: un'applicazione che prende il nome di interprete considera il testo sorgente, istruzione per istruzione, e lo traduce mentre lo esegue; su questo principio lavorano linguaggi quali il Basic, PHP, Python e altri linguaggi per la gestione di basi di dati e per le applicazioni Web.

Compilazione: un'applicazione (compilatore) che trasforma l'intero programma sorgente in linguaggio macchina, memorizzando in un file il risultato del proprio lavoro. Così un programma compilato una sola volta, può essere eseguito senza bisogno di altri interventi, quante volte si vuole. Il risultato della compilazione si chiama programma oggetto (object). Linguaggi come: C/C++ e Java.

Tradurre il Programma

Scenario di base

- Abbiamo un **linguaggio di alto livello** L (es. C, Java, Python)
- Un **programma** $P \in L$ (cioè una stringa ben formata di L)
- Una **macchina astratta** M_L (il modello concettuale su cui L è definito, es. macchina a stack, macchina astratta OO, ecc.)
- Una **macchina ospite** M_0 (es. processore x86, ARM, Motorola 68000 o anche una macchina virtuale come la JVM)
- Un **linguaggio oggetto** L_0, comprensibile da M_0

Il problema: trasformare P scritto in L in qualcosa che M_o possa eseguire.

Compilazione

Definizione formale:
Un **compilatore** è una funzione di traduzione:

$$\boldsymbol{C: L \rightarrow L_0}$$

che, dato un programma sorgente P ∈ L, produce un programma oggetto C(P) ∈ L_0.
L'esecuzione avviene poi su M_0 :

$$\mathbf{M_0(C(P))= M_L(P)}$$

cioè: il comportamento del programma oggetto compilato su M_0 deve essere equivalente al comportamento del programma sorgente su M_L.

Caratteristiche:

- La traduzione avviene **prima dell'esecuzione**.
- Il programma compilato (C(P)) può essere eseguito più volte senza ripetere la compilazione.
- Esempi: C → codice macchina, Java → bytecode JVM.

Interpretazione

Definizione formale:
Un **interprete** per L è una funzione:

$$I: L \times D \rightarrow R$$

dove:

- P ∈ L è il programma sorgente
- D è l'insieme dei dati di input
- R è l'insieme dei risultati

è tale che:

$$I(P, d) = M_L(P)(d)$$

cioè l'interprete prende direttamente il programma sorgente P e i dati d di input, ed esegue passo passo le istruzioni definite da M_L, producendo il risultato.

Caratteristiche:

- Non produce un programma oggetto indipendente: traduce ed esegue "al volo".
- Ogni esecuzione di P richiede di nuovo l'interpretazione.
- Esempi: interprete Python (CPython), interprete Prolog.

Differenze

- **Compilazione**:
  - Trasforma P una volta in un equivalente P' (codice oggetto), poi delega l'esecuzione a M_o.
  - Si può scrivere come:

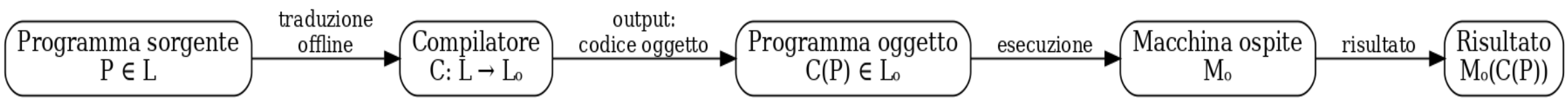
$$\boldsymbol{Esecuzione(P,d) = M_0(C(P))(d)}$$

- **Interpretazione**:
  - Esegue direttamente P su M_o attraverso un interprete che simula M_I.
  - Si può scrivere come:

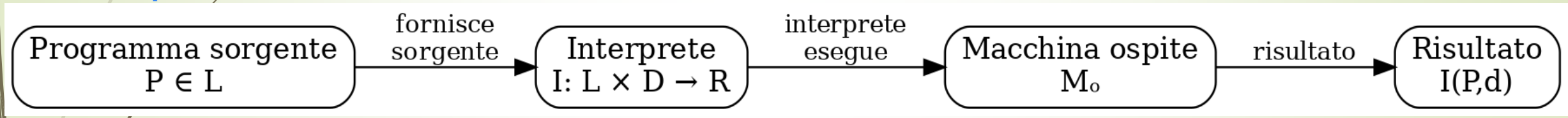
$$\boldsymbol{Esecuzione(P,d) = I(P,d)}$$

Tecniche a confronto

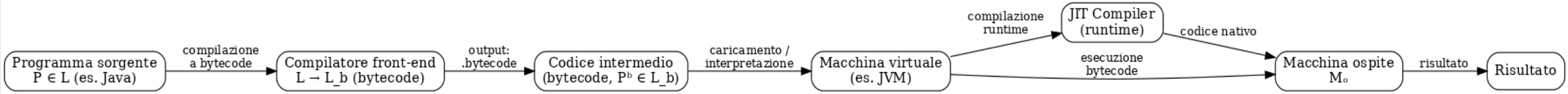
Compilare



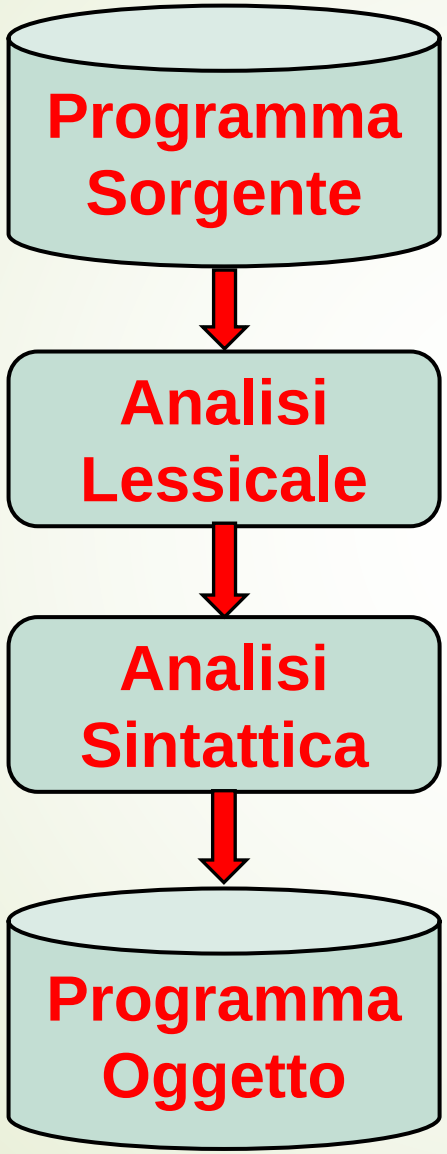
Interpretare



Soluzione Ibrida



Interpretazione e Compilazione



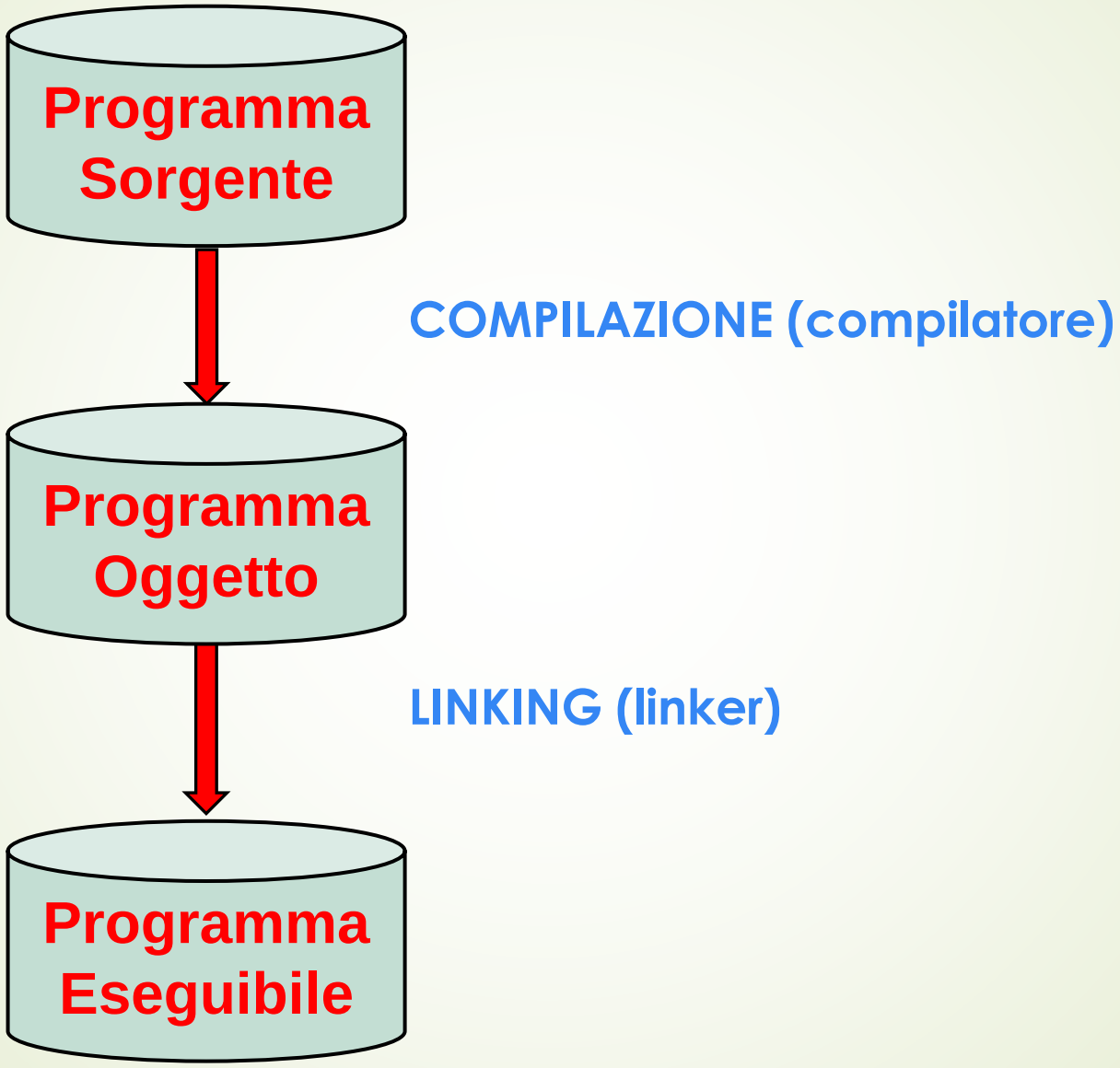
Testo del programma

Controllo della correttezza dei termini e dei simboli

Controllo della correttezza dei costrutti sintattici

Programma tradotto in linguaggio macchina

Compilazione



Pregi e difetti

Velocità di esecuzione: i programmi compilati hanno in genere prestazioni migliori (nella compilazione si possono attuare processi di ottimizzazione dell'eseguibile).

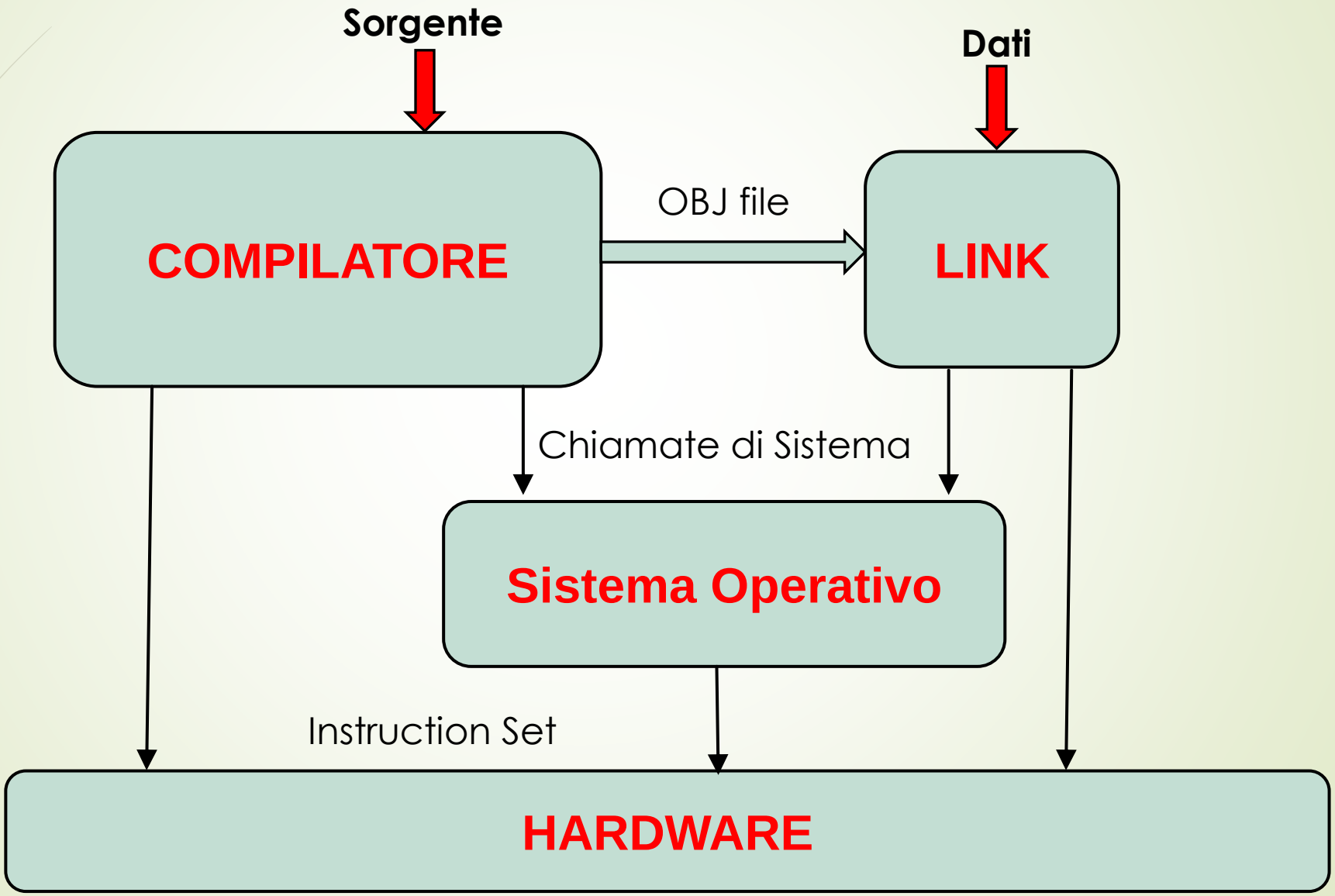
Messa a punto del programma: gli interpreti permettono di correggere gli errori non appena vengono scoperti, senza bisogno di ricompilare interamente il programma.

Sistemi Ibridi

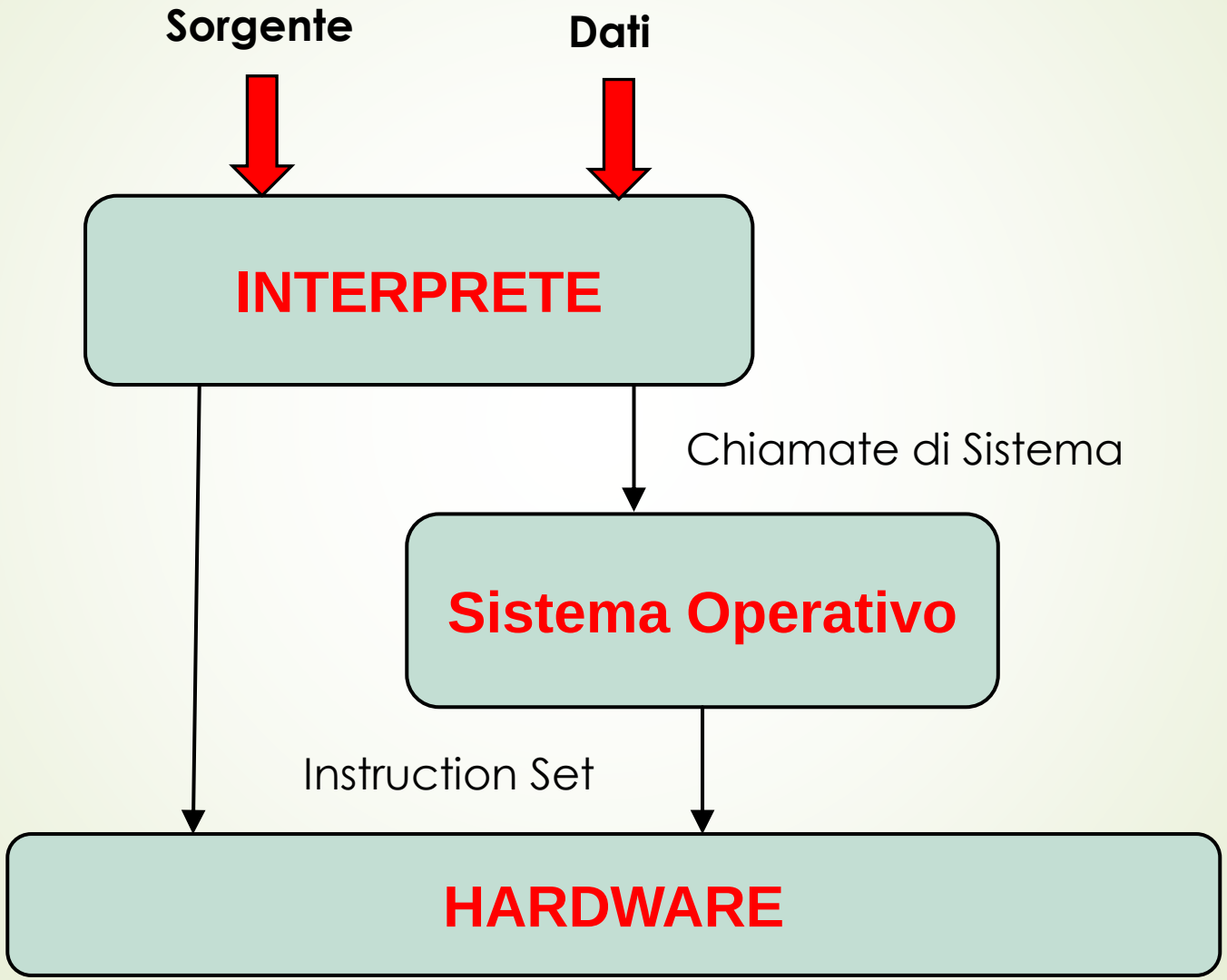
Per esempio nel linguaggio JAVA, la compilazione genera un file oggetto chiamato ByteCode che viene «interpretato» dall'interprete Java (Java Virtual Machine), per dare portabilità al codice.

Tuttavia, il ByteCode può anche essere tradotto in un eseguibile con il linker. In quest'ultimo caso, l'eseguibile è vincolato a funzionare sul particolare hardware e S.O. per il quale è stato generato.

Compilazione



Interpretazione



Spunti Bibliografici dell'Autore

[Morandini et al., 2017] Mirko Morandini, Loris Penserini, Anna Perini, Alessandro Marchetto: Engineering requirements for adaptive systems. Requirements Engineering Journal, 22(1): 77-103 (2017)

[Morandini et al., 2009] Morandini M., Penserini L., and Perini A. (2009b). Operational Semantics of Goal Models in Adaptive Agents. In 8th Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS'09). IFAAMAS.

[Morandini et al., 2008] Morandini, M., Penserini, L., and Perini, A. (2008b). Automated mapping from goal models to self-adaptive systems. In Demo session at the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), pages 485–486.

[Aldewereld et al., 2008] Huib Aldewereld, Frank Dignum, Loris Penserini, Virginia Dignum: Norm Dynamics in Adaptive Organisations. NORMAS 2008: 1-15

[Penserini et al., 2007a] Loris Penserini, Anna Perini, Angelo Susi, John Mylopoulos: High variability design for software agents: Extending Tropos. ACM Trans. Auton. Adapt. Syst. 2(4): 16 (2007)

[Penserini et al., 2007b] Loris Penserini, Anna Perini, Angelo Susi, Mirko Morandini, John Mylopoulos: A design framework for generating BDI-agents from goal models. AAMAS 2007: 149

[Pagliarecci et al., 2007] Francesco Pagliarecci, Loris Penserini, Luca Spalazzi: From a Goal-Oriented Methodology to a BDI Agent Language: The Case of Tropos and Alan. OTM Workshops (1) 2007: 105-114

[Penserini et al., 2006a] Loris Penserini, Anna Perini, Angelo Susi, John Mylopoulos: From Stakeholder Intentions to Software Agent Implementations. CAiSE 2006: 465-479

[Penserini et al., 2006b] Loris Penserini, Anna Perini, Angelo Susi, John Mylopoulos: From Capability Specifications to Code for Multi-Agent Software. ASE 2006: 253-256

[Panti et al., 2003] Maurizio Panti, Loris Penserini, Luca Spalazzi: A critical discussion about an agent platform based on FIPA specification. SEBD 2000: 345-356

[Panti et al., 2001] Maurizio Panti, Luca Spalazzi, Loris Penserini: A Distributed Case-Based Query Rewriting. IJCAI 2001: 1005-1010

GRAZIE!