I linguaggi di programmazione

A.S. 2026-2027

Classi Terze

Prof. Valentino Pietrobon





Attribuzione - Non commerciale - Non opere derivate 4.0 Internazionale (CC BY-NC-ND 4.0)

Tu sei libero di:

Condividere — riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare questo materiale con qualsiasi mezzo e formato

Il licenziante non può revocare questi diritti fintanto che tu rispetti i termini della licenza.

Alle seguenti condizioni:



Attribuzione — Devi riconoscere una menzione di paternità adeguata, fornire un link alla licenza e indicare se sono state effettuate delle modifiche. Puoi fare ciò in qualsiasi maniera ragionevole possibile, ma non con modalità tali da suggerire che il licenziante avalli te o il tuo utilizzo del materiale.



Non Commerciale — Non puoi utilizzare il materiale per scopi commerciali.



Non opere derivate — Se remixi, trasformi il materiale o ti basi su di esso, non puoi distribuire il materiale così modificato.

Divieto di restrizioni aggiuntive — Non puoi applicare termini legali o misure tecnologiche che impongano ad altri soggetti dei vincoli giuridici su quanto la licenza consente loro di fare.

Note:

Non sei tenuto a rispettare i termini della licenza per quelle componenti del materiale che siano in pubblico dominio o nei casi in cui il tuo utilizzo sia consentito da una eccezione o limitazione prevista dalla legge.

Non sono fornite garanzie. La licenza può non conferirti tutte le autorizzazioni necessarie per l'utilizzo che ti prefiggi. Ad esempio, diritti di terzi come i diritti all'immagine, alla riservatezza e i diritti morali potrebbero restringere gli usi che ti prefiggi sul materiale.

<table>
<tr><th colspan="3">SEZIONE A: Traguardi formativi</th></tr>
<tr><td>COMPETENZA CHIAVE EUROPEA:</td><td colspan="2">COMPETENZE DIGITALI – Secondo Biennio</td></tr>
<tr><td>Fonti di legittimazione:</td><td colspan="2">Raccomandazione del Parlamento Europeo e del Consiglio 18.12.2006
Raccomandazione del Parlamento Europeo e del Consiglio 23.04.2008
D.M.139/2007; Regolamento e Indicazioni Nazionali dei Licei 2010
DigComp 3.0 – Quadro europeo delle competenze digitali, quinta edizione (Cosgrove e Cachia, 2025; JRC144121) e relativa errata del 27/01/2026</td></tr>
<tr><th>COMPETENZE SPECIFICHE</th><th>ABILITA’</th><th>CONOSCENZE</th></tr>
<tr>
<td>1. Comprendere l'evoluzione storica e i concetti fondamentali dei linguaggi di programmazione, sapendoli confrontare tra loro e collocare nel contesto dei diversi paradigmi.

DigComp 3.0 · Area 3 Creazione di contenuti – 3.4 Pensiero computazionale e programmazione: CS3.4.02 (B), CS3.4.08 (I)

2. Progettare un algoritmo risolutivo e tradurlo in un linguaggio di programmazione, scegliendo consapevolmente strumenti, tecniche e metodologie di sviluppo.

3. DigComp 3.0 · Area 3 Creazione di contenuti – 3.4 Pensiero computazionale e programmazione: CS3.4.10* (I), CS3.4.11 (I), CS3.4.12* (A)</td>
<td>Storia dei linguaggi di programmazione
• Sa collocare nel tempo i principali linguaggi di programmazione e le relative famiglie. (OM)
• Sa descrivere l'evoluzione dei linguaggi di programmazione dai primi linguaggi macchina ai linguaggi moderni.
• Sa motivare l'attuale diffusione di Python nell'ambito dell'intelligenza artificiale e della scienza dei dati.

Linguaggio Macchina e Assembly
• Sa descrivere le caratteristiche di un'istruzione in Linguaggio Macchina e la sua struttura. (OM)
• Sa distinguere un salto condizionato da un salto incondizionato. (OM)
• Sa spiegare perché è stato introdotto il linguaggio Assembly e quali problemi risolve. (OM)

Concetti fondamentali dei linguaggi ad alto livello
• Sa distinguere un linguaggio ad alto livello da un linguaggio a basso livello. (OM)
• Sa utilizzare correttamente i concetti di variabile, costante, tipo di dato e struttura dati. (OM)
• Sa riconoscere le principali strutture di controllo e i sottoprogrammi. (OM)

Il codice sorgente, le librerie e le API
• Sa distinguere un Editor da un IDE e indicarne le funzionalità principali. (OM)
• Sa distinguere una libreria standard da una di terze parti. (OM)
• Sa spiegare cos'è un'API e farne un esempio.

Paradigmi di programmazione
• Sa definire il concetto di paradigma di programmazione e cosa lo differenzia da un altro. (OM)
• Sa associare i principali linguaggi di programmazione al relativo paradigma.
• Sa enunciare il teorema di Böhm-Jacopini e le tre strutture di controllo che ne derivano. (OM)

La programmazione: progettazione, algoritmo e metodologie</td>
<td>Storia dei linguaggi di programmazione
• Il Plankalkül e i primi linguaggi di programmazione. (OM)
• I linguaggi progenitori (Lisp, ALGOL, Simula) e la nascita della programmazione orientata agli oggetti.
• I linguaggi di programmazione più recenti (Go, Rust, Kotlin, TypeScript, Swift) e gli ambiti applicativi.

Linguaggio Macchina e Assembly
• Il Linguaggio Macchina: istruzioni, codice operativo e operandi. (OM)
• Le famiglie di istruzioni macchina (I/O, aritmetico-logiche, di salto). (OM)
• Il linguaggio Assembly: codici mnemonici ed etichette. (OM)

Concetti fondamentali dei linguaggi ad alto livello
• Linguaggio ad alto livello: caratteristiche e vantaggi. (OM)
• Processo e stato di un processo. (OM)
• Variabili, costanti, strutture dati, strutture di controllo, sottoprogrammi. (OM)

Il codice sorgente, le librerie e le API
• Codice/Programma sorgente. (OM)
• Editor e IDE: funzionalità. (OM)
• Librerie, moduli e API.

Paradigmi di programmazione
• Paradigma di programmazione e linguaggio ibrido. (OM)
• Teorema di Böhm-Jacopini: sequenza, selezione, iterazione. (OM)
• Pensiero computazionale (Computational Thinking) e programmazione a blocchi (puzzle programming).

La programmazione: progettazione, algoritmo e metodologie
• Analisi del problema, diagramma di flusso, pseudocodifica. (OM)
• Condizioni, iterazioni e sottoprogrammi come strumenti per la scelta dell'algoritmo. (OM)
• Metodologia agile: principi, pratiche, Extreme Programming.
• Controllo di versione con Git.

Trasformazione di un programma sorgente</td>
</tr>
</table>

	• Sa descrivere le fasi di progettazione di un programma, dall'analisi del problema alla rappresentazione dell'algoritmo. (OM) • Sa utilizzare diagramma di flusso e pseudocodifica per rappresentare un algoritmo. (OM) • Sa descrivere la metodologia di sviluppo agile e distinguerla dal modello tradizionale a cascata. • Sa utilizzare i comandi essenziali di Git per il controllo di versione. **Trasformazione di un programma sorgente** • Sa distinguere compilazione e interpretazione come modalità di traduzione di un programma. (OM) • Sa indicare vantaggi e svantaggi della compilazione e dell'interpretazione. (OM) • Sa spiegare il funzionamento degli approcci ibridi (Bytecode) adottati da Java e Python. **Il vibe coding e la Spec-Driven Development** • Sa descrivere cos'è il vibe coding e i rischi di un suo utilizzo privo di una specifica. • Sa descrivere le quattro fasi della Spec-Driven Development (Constitution, Specify, Plan, Tasks).	• Compilatore e interprete: funzionamento. (OM) • Front-end e back-end della compilazione. • Collegamento (linking) statico e dinamico. • Bytecode, P-code e compilazione Just In Time. **Il vibe coding e la Spec-Driven Development** • Vibe coding: origine e rischi. • Spec-Driven Development: Constitution, Specify, Plan, Tasks.

(**OM – Obiettivi Minimi**) indica che la voce fa riferimento alla **programmazione per obiettivi minimi** e quindi che interessa tutti gli studenti, anche quelli che hanno una programmazione per obiettivi minimi.

La programmazione riconducibile agli OBIETTIVI MINIMI conforme ai programmi ministeriali, o comunque ad essi globalmente corrispondente è prevista dall'art. 15 comma 3 dell'O.M. n.90 del 21/5/2001.

Per gli studenti che seguono obiettivi riconducibili ai programmi ministeriali è possibile prevedere:

1. Un **programma minimo**, con la ricerca dei **contenuti essenziali delle discipline**;
2. Un **programma equipollente** con la **riduzione parziale e/o sostituzione dei contenuti, ricercando la medesima valenza formativa** (art. 318 del D.L.vo 297/1994).

Sia per le verifiche che vengono effettuate durante l'anno scolastico, sia per le prove che vengono effettuate in sede d'esame, possono essere predisposte prove equipollenti, che verifichino il livello di preparazione culturale idoneo per il rilascio del diploma finale.

Gli alunni con gli OBIETTIVI MINIMI partecipano a pieno titolo agli esami di stato e acquisiscono il titolo di Studio.

Sommario

1 Linguaggi di programmazione 6
1.1 Storia dei linguaggi di programmazione 6
1.2 Il linguaggio Assembly e il Linguaggio Macchina 9
1.2.1 Il linguaggio Assembly 10
2 Linguaggi di Programmazione ad alto livello 12
2.1 Concetti fondamentali dei Linguaggi ad alto livello 12
2.2 Il codice sorgente 12
2.3 Librerie, moduli e API 14
2.4 Paradigmi di programmazione 15
2.5 La programmazione 17
2.5.1 La progettazione di un programma 18
2.5.2 La scelta dell'algoritmo 18
2.5.3 La scelta delle tecniche e delle metodologie 19
2.6 Trasformazione di un programma sorgente 21
2.6.1 La compilazione 21
2.6.2 Vantaggi e svantaggi della compilazione 22
2.6.3 Il collegamento (linking) 23
2.6.4 L'interpretazione 23
2.6.5 Approcci ibridi: Bytecode e P-code 24
3 Il vibe coding e la Spec-Driven Development 24
3.1 Le quattro fasi 25
3.2 Come si scrivono in pratica: il progetto che vi accompagnerà per tre anni 25
4 Appendice — Esempio fasi Vibe Coding 27
5 Appendice — Scheda Visual Studio Code 47
6 Appendice — Scheda Google Colab 48
7 Appendice — Scheda Terminale 50
8 Appendice — Scheda Git: comandi essenziali 51
9 Sitografia 53

1 Linguaggi di programmazione

Un Linguaggio di Programmazione (LdP) è un linguaggio "formale", dotato (al pari di un qualsiasi linguaggio naturale) di un lessico (parole - token), di una sintassi (struttura grammaticale di token) e di una semantica ben definiti (significati attribuiti alle frasi di token).

I LdP sono utilizzati per controllare il comportamento di una **macchina formale astratta** — un modello teorico, puramente matematico, di che cosa significhi calcolare — oppure di una sua **implementazione fisica concreta**, cioè un dispositivo reale che realizza quel modello (tipicamente un computer).

Si definisce programmazione l'attività, svolta da un programmatore, di scrittura di un programma comprensibile alla macchina formale o alla sua implementazione fisica, in modo da farle eseguire il comportamento desiderato.



Figura 1 - I linguaggi di programmazione

1.1 Storia dei linguaggi di programmazione

Il primo LdP della storia, se si esclude il linguaggio meccanico utilizzato da Ada Lovelace[1] per la programmazione della Macchina Analitica[2] di Charles Babbage[3], è a rigor di termini il Plankalkül[4] di Konrad Zuse[5], sviluppato nella Svizzera neutrale durante la Seconda Guerra Mondiale e pubblicato nel 1946. Plankalkül non venne mai realmente usato per programmare.

[1] https://it.wikipedia.org/wiki/Ada_Lovelace
[2] https://it.wikipedia.org/wiki/Macchina_analitica
[3] https://it.wikipedia.org/wiki/Charles_Babbage
[4] https://it.wikipedia.org/wiki/Plankalkül
[5] https://it.wikipedia.org/wiki/Konrad_Zuse

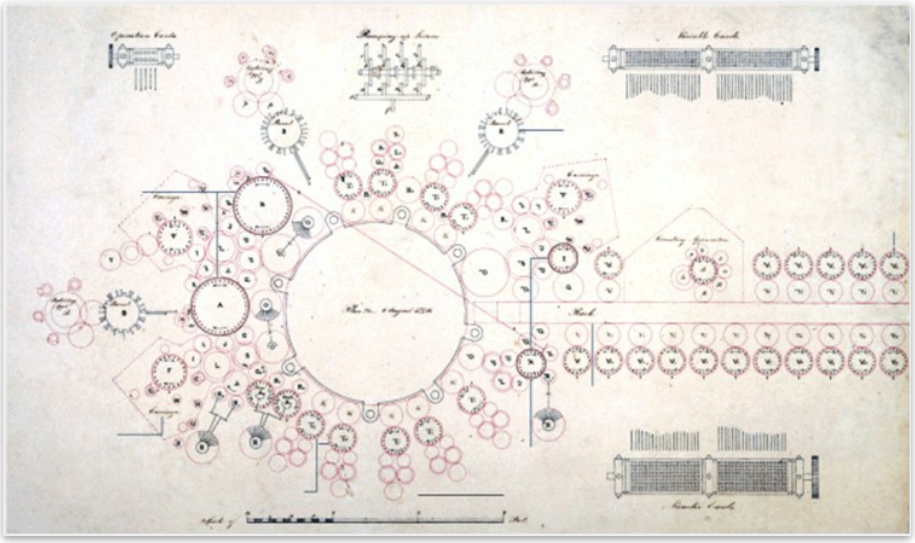


Figura 2 - Progetti della Macchina Analitica di Babbage (disegno di Charles Babbage, XIX secolo, pubblico dominio)

La programmazione dei primi elaboratori veniva fatta invece in Short Code (**Linguaggio a basso livello**), da cui poi si è evoluto **l'Assembly**, che costituisce una rappresentazione simbolica del **Linguaggio Macchina** (LM).

La maggior parte dei LdP successivi (**Linguaggi ad alto livello**) cercarono di astrarsi da tale livello basilare, dando la possibilità di rappresentare Strutture Dati e Strutture di Controllo più generali e più vicine alla modalità (umana) di rappresentare gli algoritmi risolutivi per i quali si scrivono i programmi.

FORTRAN – media di una serie di valori

```
C     MEDIA DI N VALORI LETTI DA SCHEDA
      DIMENSION X(100)
      READ (5,10) N
   10 FORMAT (I3)
      SOMMA = 0.0
      DO 20 I = 1, N
      READ (5,15) X(I)
   15 FORMAT (F8.2)
      SOMMA = SOMMA + X(I)
   20 CONTINUE
      WRITE (6,30) SOMMA / N
   30 FORMAT (6H MEDIA, F10.3)
      STOP
      END
```

Figura 3 - Programma scritto in FORTRAN

Tra i primi linguaggi ad alto livello a raggiungere una certa popolarità ci fu il **Fortran**, creato nel 1957 da John Backus, da cui derivò successivamente il **BASIC** (1964). Oltre al salto condizionato, reso con l'istruzione IF, questa nuova generazione di linguaggi introdusse nuove strutture di controllo di flusso come i cicli WHILE e FOR e le istruzioni CASE/SWITCH, in questo modo diminuì di molto il ricorso alle istruzioni di salto (GOTO), cosa che rese il codice più chiaro ed elegante, e quindi più facile da correggere e aggiornare.

ALGOL 60 – valore assoluto massimo in una matrice

```
procedure Absmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);
  value n, m; array a; integer n, m, i, k; real y;
begin integer p, q;
  y := 0; i := k := 1;
  for p := 1 step 1 until n do
    for q := 1 step 1 until m do
      if abs(a[p, q]) > y then
        begin y := abs(a[p, q]); i := p; k := q end
end Absmax
```

Figura 4 - Frammento di programma scritto in ALGOL

Dopo la comparsa del Fortran, nacquero tutta una serie di altri linguaggi di programmazione storici, che implementarono una serie di idee e paradigmi innovativi. I più importanti furono il **Lisp** (1959) e **l'ALGOL** (1960).

Tutti i linguaggi di programmazione oggi esistenti possono essere considerati discendenti da uno o più di questi primi linguaggi, da cui traggono molti concetti di base.

LISP – fattoriale ricorsivo e somma di una lista

```
(defun fattoriale (n)
  (if (<= n 1)
      1
      (* n (fattoriale (- n 1)))))

(defun somma (lista)
  (if (null lista)
      0
      (+ (car lista) (somma (cdr lista)))))

(fattoriale 5)        ; => 120
(somma '(3 4 5))      ; => 12
```

Figura 5 - Frammento di programma scritto in LISP

L'ultimo grande progenitore dei linguaggi moderni fu **Simula** (1967), che introdusse il concetto (allora appena abbozzato) di Programmazione Orientata agli Oggetti (OOP -Object Oriented Programming).

Nel 1970 Niklaus Wirth pubblica il **Pascal**, il primo linguaggio strutturato, a scopo didattico, peraltro ancora utilizzato ad esempio nella programmazione dei robot industriali.

Nel 1972 appare il **C**, che fu fin dall'inizio un grande successo e con il quale vennero scritti molti Sistemi Operativi (es. Linux, macOS, Windows, Android, iOS) e software di uso comune.

Nello stesso anno (1972) compare anche il **Prolog**, finora il principale esempio di linguaggio logico, che pur non utilizzato per lo sviluppo industriale del software (a causa della sua inefficienza) rappresenta una possibilità teorica estremamente affascinante.

PROLOG – fatti, regole e interrogazioni

```
padre(bruno, carlo).
padre(carlo, dario).
nonno(X, Z) :- padre(X, Y), padre(Y, Z).
?- nonno(bruno, Chi).      % Chi = dario
```

Figura 6 -Frammento di programma scritto in PROLOG

Con i primi mini e microcomputer e le ricerche a Palo Alto, nel 1983 vede la luce **Smalltalk**, il primo linguaggio realmente e completamente ad oggetti, che si ispira al Simula e al Lisp.

Smalltalk viene ricordato per l'enorme influenza esercitato sulla storia dei LdP, introducendo il paradigma Object-Oriented (OO) nella sua prima forma matura.

Esempi di linguaggi OO odierni sono: **Objective-C** (1983), **C++** (1986), **Java** (1995), **C#** (2000), ecc.

Come si può notare molti altri sono i linguaggi di programmazione che nel frattempo son comparsi e che si sono imposti, primi fra tutti **Python** del 1991 (OO), **PHP** del 1995 (linguaggio di scripting utilizzato nelle pagine web), **Javascript** del 1996 (OO e Event Driven utilizzato nelle pagine web), **C#** del 2000 (strettamente legato .NET Framework sviluppato da Microsoft), **SQL** del 1974 (non proprio un linguaggio di programmazione quanto piuttosto un linguaggio per i database).

Il duemila porta poi una nuova generazione di linguaggi, ciascuno nato per risolvere un problema specifico rimasto scoperto: Go (2009, semplicità e concorrenza per i server e il cloud, sviluppato in Google), Rust (2010, sicurezza della memoria senza garbage collector, tanto da essere entrato solo di recente — per la prima volta nella sua storia — nella top ten degli indici di popolarità più seguiti), Kotlin (2011, linguaggio moderno compatibile con Java, oggi ufficiale per Android), TypeScript (2012, JavaScript con i tipi, per applicazioni web di grandi dimensioni), Swift (2014, il rimpiazzo Apple per Objective-C).

Oggi Python occupa stabilmente la prima posizione negli indici di popolarità dei linguaggi (come il TIOBE Index), un primato dovuto in gran parte proprio al suo ruolo nell'intelligenza artificiale e nella scienza dei dati — un filo che riprenderemo più avanti, quando parleremo di librerie e API.

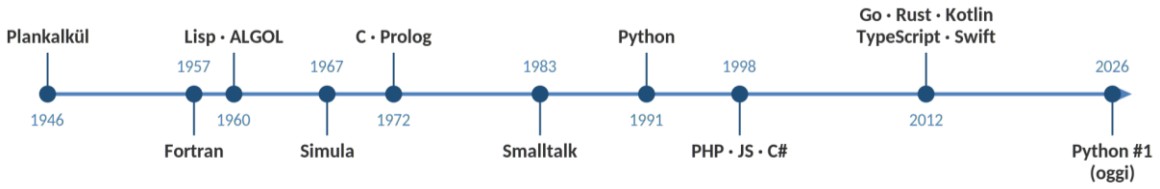


Figura 7 - Linea del tempo dei principali linguaggi di programmazione

Ciò che risulta evidente è che non esiste un linguaggio di programmazione migliore di altri in termini assoluti ma molto dipende dall'uso che se ne vuole fare e, in seconda battuta, da chi lo propone, anche se questo aspetto è meno rilevante di quanto lo era in passato.

Per un elenco completo di tutti i linguaggi di programmazione sviluppati si faccia riferimento al sito: https://it.wikipedia.org/wiki/Elenco_cronologico_dei_linguaggi_di_programmazione

1.2 Il linguaggio Assembly e il Linguaggio Macchina

Si dice Istruzione Macchina una istruzione corrispondente ad una operazione direttamente eseguibile dall'Hardware dell'elaboratore. L'insieme di tali istruzioni si chiama Linguaggio Macchina.

I programmi che devono essere eseguiti da un elaboratore elettronico devono risiedere in memoria centrale (RAM), secondo quanto proposto da John Von Neumann nel 1946.

L'esecuzione avviene istruzione dopo istruzione in modo sequenziale e le istruzioni devono essere espresse in Linguaggio Macchina.

Le Istruzioni Macchina hanno le seguenti caratteristiche:

- sono codificate in **forma binaria**;
- sono identificate da un **codice** che indica l'operazione da eseguire;
- sono le **uniche** istruzioni che un calcolatore può comprendere ed eseguire;
- **dipendono fortemente dall'architettura** dell'elaboratore: dai registri della CPU, dalle varie unità funzionali; ne consegue che **ogni elaboratore ha un proprio Linguaggio Macchina**;
- l'insieme delle istruzioni macchina eseguibili da un elaboratore costituisce il suo **repertorio**.

In generale una istruzione macchina può essere così schematizzata:

Codice Operativo	Operandi

Le Istruzioni Macchina possono essere classificate, in base al tipo di operazione svolta, in quattro famiglie: istruzioni di I/O (scambio dati con le periferiche), istruzioni aritmetico-logiche (elaborazione dei dati), istruzioni di spostamento (dati tra memoria e registri della CPU) e istruzioni di salto (modificano la sequenza di esecuzione di un programma).

La sola forma di controllo di flusso fornita dal Linguaggio Macchina è l'istruzione di salto, che porta a scrivere programmi difficili da comprendere per via dei continui salti da un punto all'altro del codice:

```
Es 1:
        …
        MOV AX, num
        MOV CH, pippo
        MOV BH, felix
        SUB CH, BH
        JZ oltre
        DIV CH
oltre:
        MOV CX, 20
        …
```

```
Es 2:
        …
        CMP sesso, "M"
        JNE donna
        MOV milit, "S"
        JMP oltre
donna:
        MOV milit, "N"
oltre:
        MOV BX, felix
        ADD AX, BX
        MOV CX, 20
        …
```

Figura 8 - Esempi di salto condizionato (JZ – Jump Zero, JNE – Jump Not Equal) e di salto incondizionato (JMP – Jump)

Fra le **istruzioni di salto** ne esistono di due tipi:

- **incondizionato**: in questo caso il salto viene comunque effettuato (JMP);
- **condizionato**: il salto viene effettuato solo se si è verificata una certa condizione; tale condizione viene testata tramite l'esame di un opportuno bit del **Registro di Stato (Jx)**.

1.2.1 Il linguaggio Assembly

Il linguaggio Assembly è il linguaggio di programmazione simbolico più a basso livello cioè più vicino alla logica di funzionamento della macchina.

Esso si distingue dai linguaggi di alto livello come, ad esempio, il C, il Pascal, il COBOL, che, invece, sono più vicini al modo di ragionare e operare dell'uomo e in varia misura specializzati per la determinazione di soluzioni di problemi applicativi negli ambiti più vari.

Il linguaggio Assembly, comunque, costituisce un importante passo in avanti rispetto al Linguaggio Macchina permettendo di superare alcuni grossi limiti di quest'ultimo.

Problemi nell'uso del Linguaggio Macchina

Da quanto visto, per le caratteristiche del linguaggio macchina, è evidente la difficoltà nello scrivere programmi servendosi di questo strumento; in sintesi:

- necessità di **ricordare i codici operativi** costituiti da sequenze di cifre binarie prive di significato immediato per l'uomo;
- necessità di **fare riferimento direttamente a indirizzi di celle di memoria** o a registri della CPU per potersi riferire agli operandi coinvolti nelle operazioni da svolgere;
- necessità di fare riferimento a **indirizzi di istruzioni per scrivere le istruzioni di salto**;
- **difficoltà nella correzione dei programmi**, in quanto qualsiasi piccola modifica come l'aggiunta di una istruzione può comportare l'esigenza di modificare molte delle altre istruzioni; infatti per «far posto» alla nuova istruzione si devono spostare in memoria quelle che la seguono e forse anche i dati su cui il programma opera, quindi tutte le istruzioni di salto verso istruzioni spostate, o che fanno riferimento a dati spostati, devono essere modificate.

Soluzioni ottenibili con l'Assembly

Questi problemi vengono risolti nei linguaggi di programmazione, sia a basso che alto livello con l'uso di:

- **codici operativi mnemonici**;
- **nomi simbolici** per i dati in memoria (le cosiddette variabili);

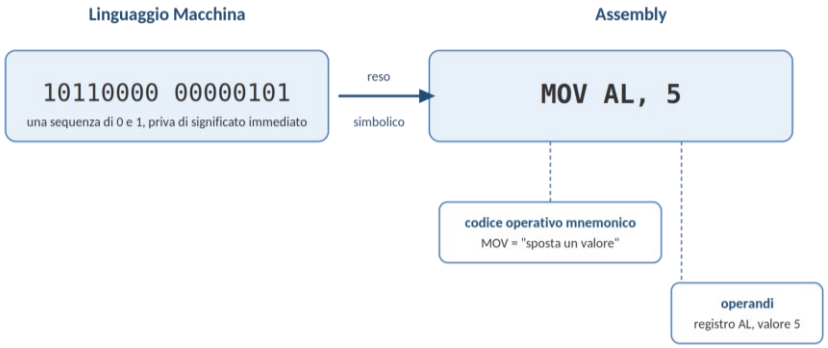


Figura 9 - Dal Linguaggio Macchina all'Assembly: la stessa istruzione, resa simbolica

- **etichette** per contrassegnare le istruzioni.

Esempio di programma scritto in assembly

```
 1 // Dati
 2 .data
 3 num1: .byte 150
 4 num2: .byte 74
 5 tot:  .word 0
 6 /* Istruzioni */
 7 .text
 8 .globl _start
 9 _start:
10 movw $0, %ax  # pulisce il reg. ax
11 movb (num1), %al
12 addb (num2), %al
13 movw %ax, (tot)
14 fine:
15 movl $1, %eax
16 int $0x80
```

2 Linguaggi di Programmazione ad alto livello

Un linguaggio di programmazione ad alto livello (High Level Programming Language), d'ora in poi indicato semplicemente con LdP, è un linguaggio che si caratterizza per una significativa astrazione dai dettagli del funzionamento di un calcolatore e dalle caratteristiche del Linguaggio Macchina (LM).

I LdP sono **progettati per essere facilmente comprensibili** dagli esseri umani, fino a includere alcuni elementi del linguaggio naturale. **Per essere eseguiti** da un calcolatore, però, i programmi scritti in un LdP **devono essere tradotti** (compilati o interpretati) **in istruzioni in LM** da un apposito programma.

2.1 Concetti fondamentali dei Linguaggi ad alto livello

Tutti i LdP esistenti sono definiti da un **lessico**, una **sintassi**, una **semantica** e permettono di scrivere:

- **Istruzioni**: comandi o regole descrittive.

A prescindere dal particolare LdP, ogni volta che un'istruzione viene eseguita, lo **stato del processo** cambia. Per **processo** si intende il **programma in esecuzione**.

Lo stato del processo sarà definito come l'insieme dei valori posseduti da tutte le variabili e le altre strutture di memoria coinvolte nell'esecuzione del programma.

Alcuni concetti sono poi presenti nella gran parte dei linguaggi:

- **Variabile e Costante**: **contenitore** per memorizzare un dato o un insieme di dati, noti o ignoti, già presenti o da inserire. Molti linguaggi, inoltre, attribuiscono a variabili e costanti un **tipo**, con differenti proprietà (stringhe di testo, numeri, liste, atomi ecc.).
- **Espressione**: una combinazione di variabili/costanti e operatori. Inizialmente introdotte per rappresentare le espressioni matematiche, in seguito la loro funzionalità si è estesa. Una espressione viene valutata per produrre un valore.
- **Strutture dati**: meccanismi che permettono di aggregare, organizzare e gestire dati in modo complesso.
- **Strutture di controllo**: permettono di governare il flusso di esecuzione del programma, alterandolo in base al risultato ottenuto dalla valutazione di espressioni logiche (ad esempio IF, SWITCH-CASE e cicli FOR, WHILE e DO-WHILE).
- **Sottoprogrammi**: un blocco di istruzioni che può essere richiamato da un qualsiasi altro punto del programma.
- **Funzionalità di input dati da tastiera e visualizzazione dati in output** (video) attraverso i cosiddetti canali standard (standard input, standard output).
- **Commenti sul codice**, per esplicitare le funzionalità a beneficio della leggibilità del codice.

2.2 Il codice sorgente

Programmare utilizzando un LdP significa, generalmente, scrivere uno o più semplici file di testo ASCII che, nel loro insieme, costituiscono il Codice Sorgente (detto anche Programma Sorgente).

Si definisce Codice Sorgente il testo che esprime l'algoritmo tradotto nel LdP scelto. Il codice sorgente, per essere eseguito, dovrà poi essere tradotto in un linguaggio comprensibile alla macchina.

I font, i colori e in generale l'aspetto grafico sono irrilevanti ai fini della programmazione in sé, i colori sono solo utilizzati come aiuto per migliorare la lettura delle istruzioni distinguendo le varie componenti di ogni istruzione.

Per questo motivo, i programmatori non usano programmi di videoscrittura, ma degli **Editor di testo** specifici (Geany, Note+, ecc) che offrono **funzioni avanzate per il trattamento dei testi** (con strumenti per scrivere

espressioni regolari, effettuare sostituzioni condizionali e ricerche su file multipli, possibilità di richiamare strumenti esterni ecc).

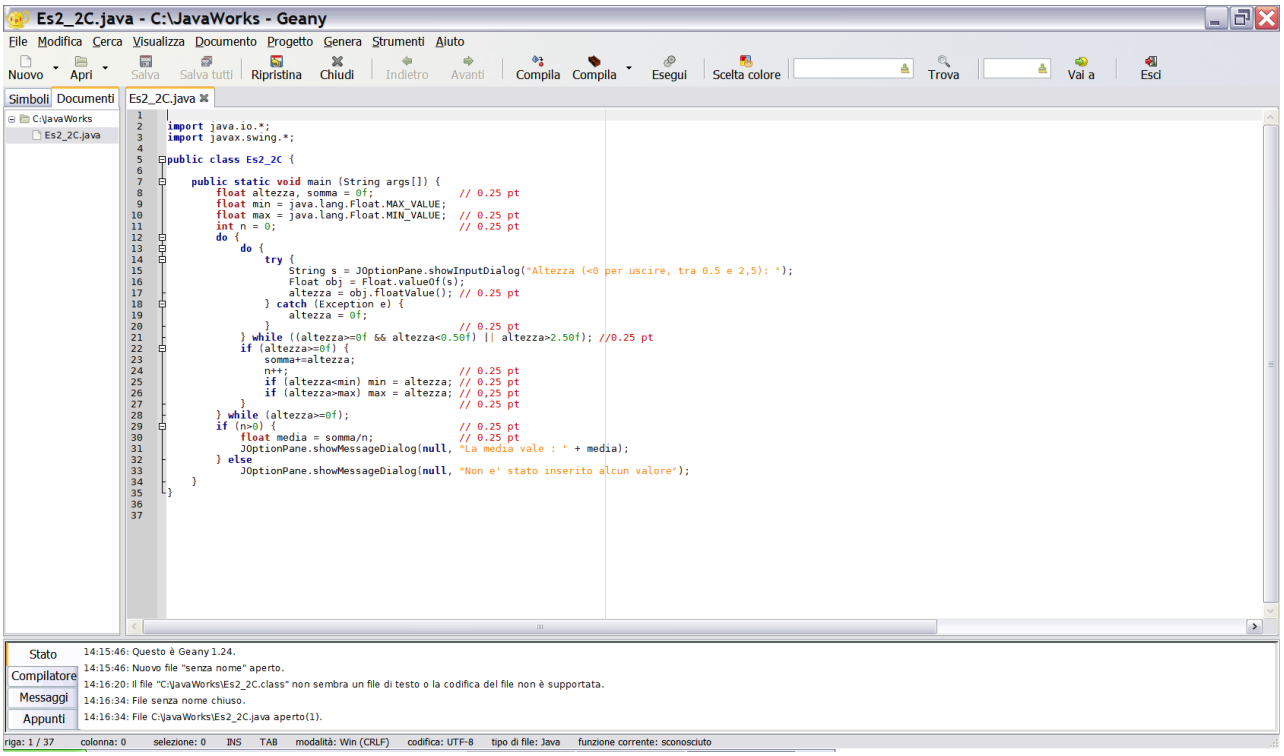


Figura 10 - Editor "Geany"

Si definisce Integrated Development Environment (IDE), o ambiente di sviluppo integrato, un Editor in grado di lavorare a stretto contatto con gli altri strumenti di sviluppo (compilatore, linker, interprete e così via).

Va notato che alcuni linguaggi di programmazione recenti consentono anche una forma mista di programmazione, in cui alla stesura di Codice Sorgente (Source Code) si associano anche operazioni di **programmazione visuale**: il programmatore costruisce alcuni elementi del programma disegnandoli direttamente a video (finestre, menù, bottoni, campi di vario tipo e così via), invece di scriverli come codice.

L'insieme di questi elementi, attraverso cui l'utente finale interagirà poi con il programma, costituisce la Graphical User Interface (GUI), o interfaccia grafica.

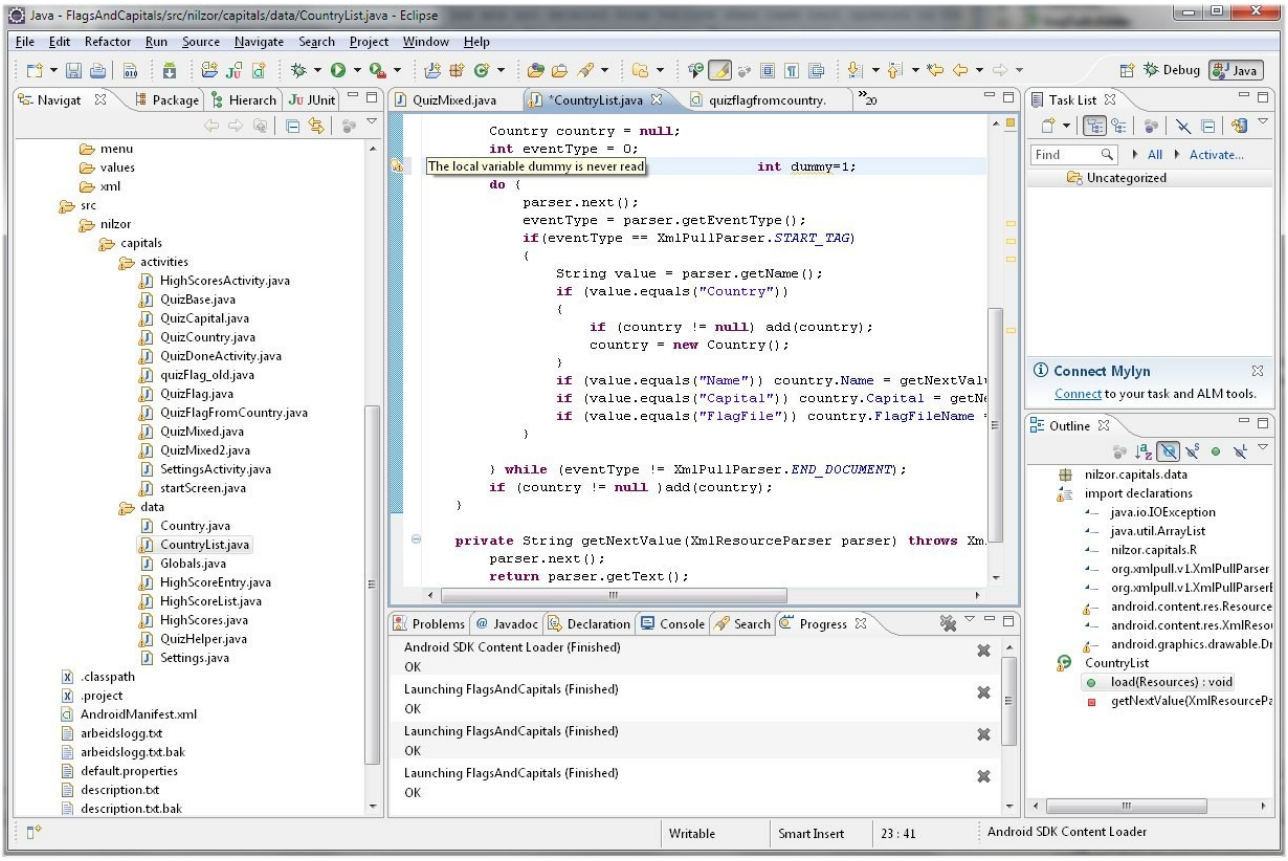


Figura 11 - IDE “Eclipse”

2.3 Librerie, moduli e API

Raramente un programma viene scritto interamente da zero: la maggior parte dei linguaggi di programmazione mette a disposizione un'ampia raccolta di codice già scritto, testato e pronto all'uso, organizzato in librerie (o moduli).

Una libreria raccoglie funzioni, procedure e strutture dati che risolvono problemi ricorrenti — calcoli matematici, gestione di testo, comunicazione in rete — evitando al programmatore di doverli reinventare ogni volta.

Si distingue solitamente fra **libreria standard**, fornita insieme al linguaggio stesso e disponibile su ogni installazione senza bisogno di configurazioni aggiuntive, e **librerie di terze parti**, create dalla comunità di sviluppatori e installabili separatamente, spesso attraverso un gestore di pacchetti dedicato.

Quando invece un programma deve comunicare non con una libreria installata sul proprio computer, ma con un altro programma o servizio — magari in esecuzione su un server remoto, raggiungibile attraverso Internet — la comunicazione avviene tramite una API (Application Programming Interface).

Si definisce API (Application Programming Interface) un insieme di regole con cui un servizio “espone” le proprie funzionalità, permettendo ad altri programmi di richiederle senza conoscerne i dettagli interni di funzionamento.

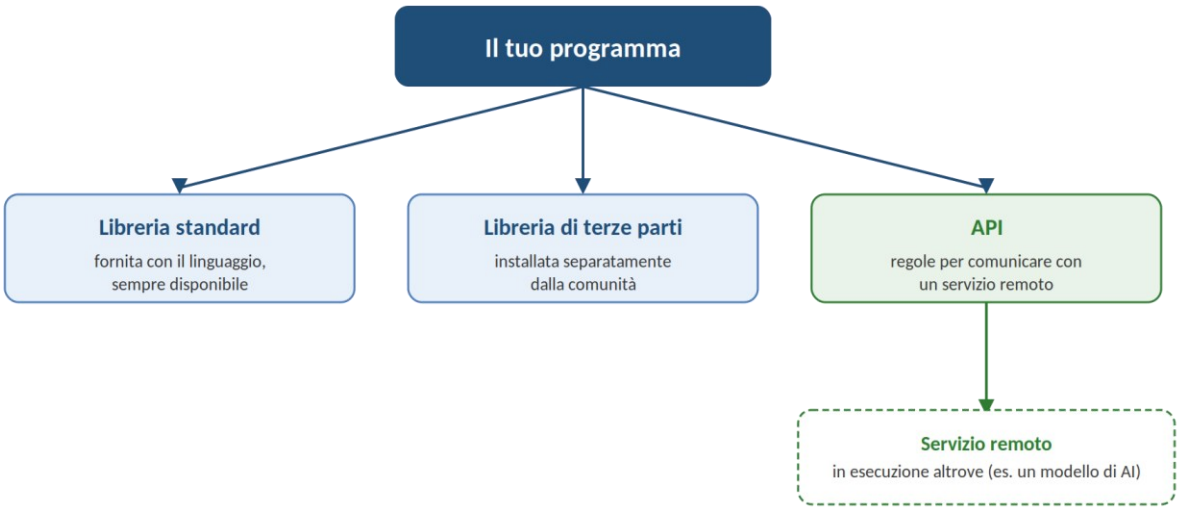


Figura 12 - Da dove arriva il codice che il tuo programma riusa

Le API sono ovunque nella programmazione moderna: un'app meteo interroga l'API di un servizio che raccoglie dati climatici, un sito di e-commerce interroga l'API di un circuito di pagamento. Un esempio di crescente rilievo, per la sua diffusione recentissima, sono le API dei servizi di intelligenza artificiale: attraverso di esse un programma può chiedere direttamente a un modello linguistico di elaborare un testo, tradurlo, riassumerlo o generare un'immagine, senza dover realizzare né eseguire il modello stesso, che rimane in esecuzione altrove.

Vedremo l'uso concreto delle **librerie in Python** — sia della libreria standard sia di eventuali librerie esterne — nel testo dedicato al linguaggio.

2.4 Paradigmi di programmazione

Per scrivere un programma si possono utilizzare approcci e filosofie risolutive diverse detti **paradigmi**.

Un paradigma di programmazione è uno stile fondamentale di programmazione, ovvero un insieme di strumenti concettuali forniti da un LdP per la stesura del "codice sorgente" di un programma, definendo dunque il modo in cui il programmatore concepisce e percepisce l'algoritmo risolutivo e il programma stesso.

I paradigmi si differenziano per i **concetti** e le **astrazioni** usate **per rappresentare gli elementi** di un programma (come ad esempio le funzioni, gli oggetti, le variabili, i vincoli, ecc.) e **per descrivere i procedimenti usati** per l'elaborazione dei dati (assegnazione, calcolo, iterazione, data flow, ecc).

Ogni LdP è generalmente ispirato e riconducibile a uno o più paradigmi di programmazione.

Un programma scritto con un linguaggio **object-oriented**, ad esempio, è costituito da oggetti che interagiscono fra loro, mentre un programma scritto con il **paradigma funzionale**, ad esempio nel Lisp, è rappresentato da una composizione di funzioni.

Fortran, Pascal e C sono basati sul **paradigma imperativo della programmazione strutturata**, viceversa il **Prolog** si basa sul **paradigma logico** e il programma risulta essere una sequenza non ordinata di predicati logici.

Alcuni linguaggi di programmazione sono **influenzati da molteplici paradigmi**; un esempio macroscopico è dato dall'**Ada**, che fu esplicitamente progettato come summa di concetti provenienti dalla **programmazione strutturata**, dalla **programmazione modulare**, dalla **programmazione concorrente** e dalla **programmazione per tipi di dati astratti**.

La storia dei paradigmi di programmazione si presta a essere rappresentata con un albero genealogico; in molti casi, infatti, un nuovo paradigma nasce come evoluzione di un altro, aggiungendo nuovi concetti, ma mantenendo anche quelli fondamentali del precedente.

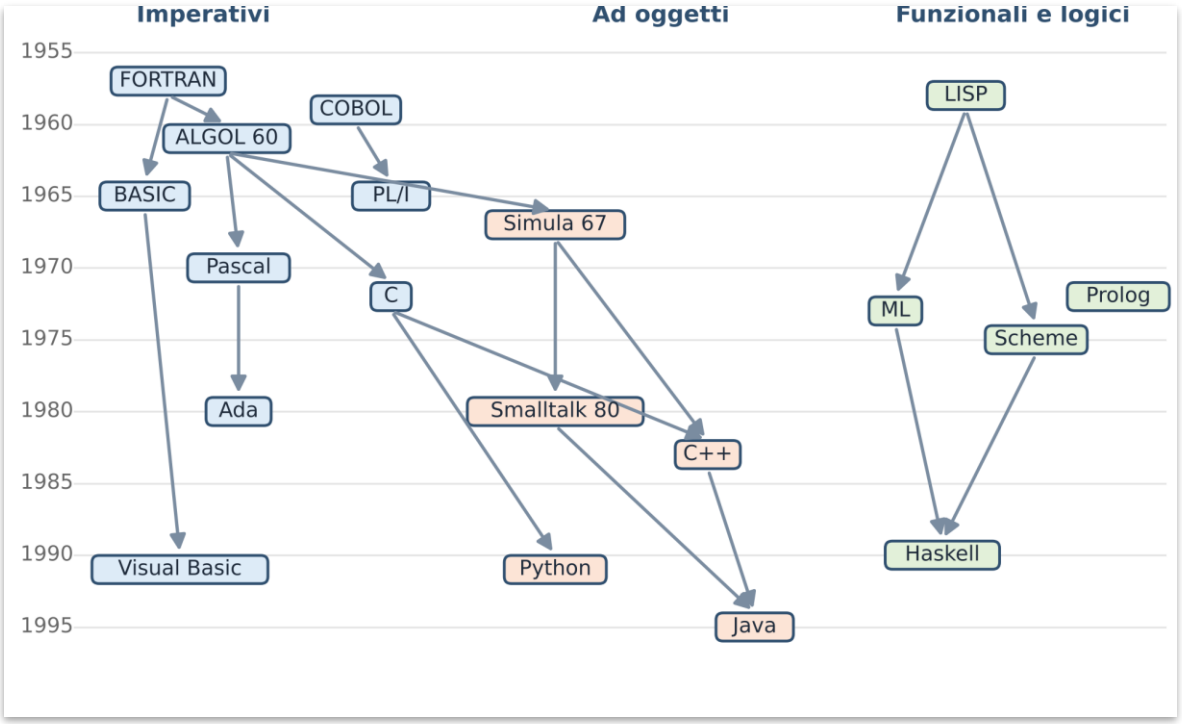


Figura 13 - Vincoli evolutivi tra linguaggi di programmazione fino a metà degli anni '90talvolta questo "salto evolutivo" consiste nel rendere obbligatoria e/o supportare esplicitamente quella che nei paradigmi precedenti si era affermata come regola di buona programmazione.

Per esempio, la **programmazione strutturata** ha introdotto **strutture di controllo standard** (sequenze, cicli e alternative) e ha **vietato l'uso dell'istruzione di salto** (goto), della quale si era dimostrata la **non necessità** con il teorema di Böhm-Jacopini.

Il teorema di Böhm-Jacopini, enunciato nel 1966 dagli informatici Corrado Böhm e Giuseppe Jacopini, afferma che qualunque algoritmo può essere implementato utilizzando tre sole strutture dette strutture di controllo: la sequenza, la selezione ed il ciclo (iterazione).

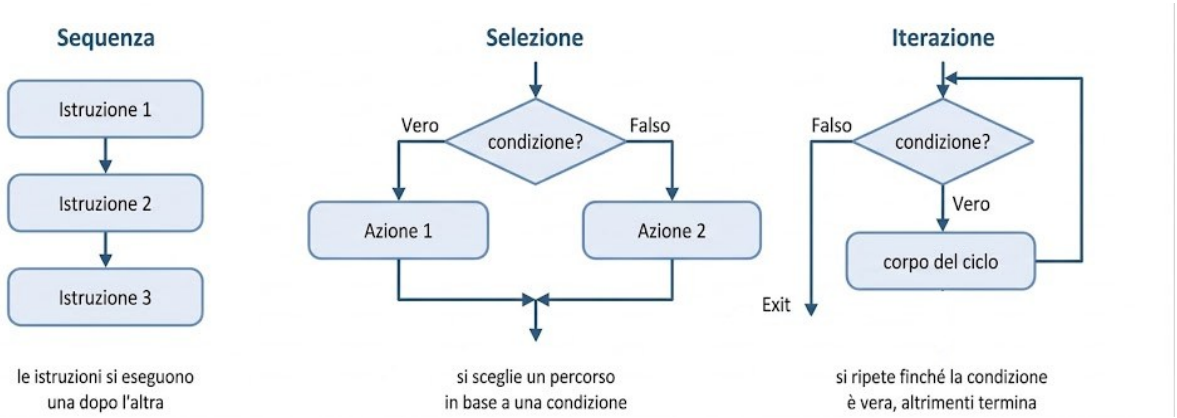


Figura 14 - Le tre strutture di controllo del teorema di Böhm-Jacopini

Le stesse strutture di controllo si ritrovano anche in un approccio alla scrittura del codice profondamente diverso da quello testuale visto finora: quello grafico (**puzzle programming**). Nato tra la fine degli anni 2000, con **Scratch (2007)** e **App Inventor (2010)** come esempi pionieristici, si è poi diffuso soprattutto nella scuola primaria e secondaria di primo grado, dove ha accompagnato la diffusione del **pensiero computazionale**

(Computational Thinking) e della **programmazione (Coding)** tra i più giovani.

Questi linguaggi di programmazione si presentano come veri e propri ambienti di sviluppo grafico in cui la scrittura dei programmi avviene mediante la giustapposizione di blocchi predefiniti, ognuno con un determinato significato.

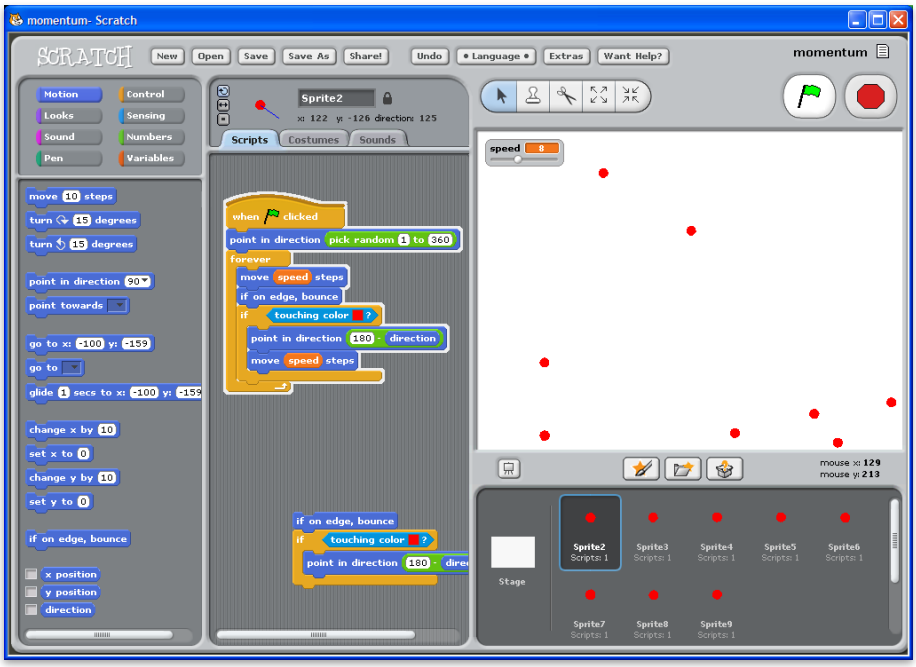


Figura 15 - Interfaccia grafica di Scratch (Scratch è sviluppato dal Lifelong Kindergarten Group del MIT Media Lab, scratch.mit.edu)

2.5 La programmazione

La programmazione è l'insieme delle attività e tecniche che, una o più persone specializzate: programmatori (programmers) o sviluppatori (developers), svolgono per creare un programma scrivendo il relativo codice sorgente (source code) in un certo LdP.

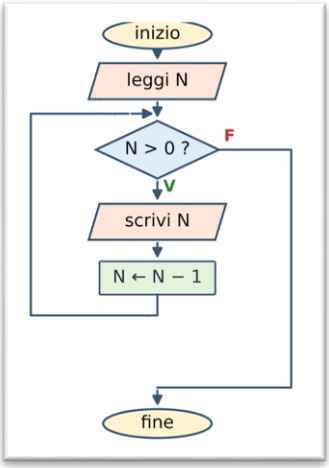
Con l'avvento dell'**ingegneria del software (software engineering)**, l'attività di programmazione rappresenta solo la fase implementativa dell'intero **ciclo di sviluppo del software**. Essa ha come obiettivo finale quello di soddisfare le specifiche funzionalità richieste dal committente ed emerse durante la fase iniziale di analisi.

2.5.1 La progettazione di un programma

Superata la fase di definizione del problema (**Analisi del problema**), in cui si evidenziano gli **input** (con eventuali vincoli) e gli **output** che definiscono la soluzione del problema, occorre definire delle possibili soluzioni (algoritmi risolutivi), porle a confronto e individuare quella più efficiente.

Per definire le varie soluzioni (algoritmi risolutivi) possono essere utilizzati 2 diversi strumenti:

- **Diagramma di flusso (Flow chart), una rappresentazione grafica molto intuitiva che può illustrare un algoritmo risolutivo in tutti i singoli passi che lo caratterizzano. Ogni passo è collegato agli altri con delle frecce che indicano la direzione del processo di esecuzione.**
- **Pseudocodifica, una forma di scrittura affine alla lingua parlata dal programmatore. La stesura dell'algoritmo è simile ad una scaletta di operazioni che si devono eseguire per ottenere la soluzione. La scrittura della pseudo-codifica deve essere dettagliata almeno quanto il codice finale, ma rispetto ad esso non richiede una sintassi precisa lasciando il programmatore libero di concentrarsi sulle questioni concettuali.**



Pseudocodice: prodotto come somme ripetute

```
INIZIO
   leggi N, M
   prodotto ← 0
   ripeti M volte:
      prodotto ← prodotto + N
   scrivi prodotto
FINE
```

Figura 16 - Esempi di Diagramma di flusso (a sinistra) e di Pseudo-codifica (a destra)

Il programmatore esperto salta spesso questa fase di sviluppo e passa direttamente alla fase successiva pensando direttamente la soluzione algoritmica in termini di istruzioni in un determinato linguaggio di programmazione. Questo approccio può essere controproducente, infatti nell'approccio object-oriented (OO) è prevista una fase iniziale dichiarativa con strumenti specifici che successivamente genereranno in modo automatico parte del codice sorgente e solo in un secondo momento il programmatore procederà con la scrittura diretta del codice.

2.5.2 La scelta dell'algoritmo

Per creare un programma efficiente occorre progettarne con attenzione l'algoritmo risolutivo.

Ad esempio, potremmo creare un algoritmo che esegue i seguenti passi:

- Esamina una lista di numeri e riconosci il minore;
- Assegna il numero minore alla variabile X;
- Stampa la variabile X.

È importante precisare che ogni successione di passi deve contenere istruzioni chiare, precise e non ambigue, il cui significato non dipenda dal contesto in cui lo si scrive.

Per dare al programma una struttura più complessa si utilizzano i seguenti strumenti:

- **Le condizioni**: consistono nel fornire al codice delle funzionalità di controllo che permettono di cambiare lo svolgimento dell'algoritmo a seconda del risultato della condizione espressa.

- **I cicli**: sono sequenze di istruzioni eseguite ripetutamente fin quando non viene soddisfatta una determinata condizione. Ogni ripetizione è detta iterazione.

I programmi possono, inoltre, essere divisi in parti in modo che i vari compiti possano essere eseguiti attraverso sequenze più semplici di passi. A tale scopo il programma viene suddiviso in **funzioni / procedure / metodi**.

2.5.3 La scelta delle tecniche e delle metodologie

In passato si poneva particolare attenzione alla scelta del paradigma di programmazione da adottare e quindi al linguaggio di programmazione. A partire dal 2000 si è iniziato a porre l'attenzione su aspetti diversi, precedentemente trascurati o considerati poco rilevanti ma che concorrono ad una metodologia di programmazione agile.

Nell'ingegneria del software, l'espressione metodologia agile (o sviluppo agile del software, in inglese agile software development, abbreviato in ASD) si riferisce ad un insieme di metodi per lo sviluppo del software fondati su principi direttamente o indirettamente derivati dai princìpi del "Manifesto per lo sviluppo agile del software" (Manifesto for Agile Software Development, impropriamente chiamato anche "Manifesto Agile") pubblicato nel 2001 da Kent Beck, Robert C. Martin, Martin Fowler e altri.[6]

I **metodi agili** si contrappongono al tradizionale **modello a cascata**[7] (waterfall) o ad altri modelli di progettazione del software (**modello evolutivo**[8]**, modello trasformazionale**[9]**, prototipizzazione rapida**[10]**, modello a spirale**[11]**, modello a fontana**[12]), proponendo un approccio meno strutturato e focalizzato sull'obiettivo di consegnare al cliente, in tempi brevi e aggiornato frequentemente (**early delivery/frequent delivery**), software funzionante e di qualità.

Fra le pratiche promosse dai metodi agili ci sono:

- la **formazione di team di sviluppo piccoli**, **cross-funzionali e auto-organizzati**,
- lo **sviluppo iterativo e incrementale**,
- la **pianificazione adattiva**, e
- il **coinvolgimento diretto e continuo del cliente** nel processo di sviluppo.

Obiettivi

L'obiettivo è la **piena soddisfazione del cliente** e non solo l'adempimento di un contratto. Il corretto uso di queste metodologie, inoltre, può consentire di **abbattere i costi e i tempi di sviluppo** del software, **aumentandone la qualità**.

Essa è esplosa proprio in concomitanza con la crisi successiva al boom di Internet prendendo spunto dai metodi applicati in piccole software house che meglio di altre sono sopravvissute alla crisi.

Principi

I principi su cui si basa una metodologia agile sono solo 4:

- **le persone e le interazioni sono più importanti dei processi e degli strumenti** (ossia le relazioni e la comunicazione tra gli attori di un progetto software sono la miglior risorsa del progetto);

[6] http://agilemanifesto.org
[7] https://it.wikipedia.org/wiki/Modello_a_cascata
[8] https://it.wikipedia.org/wiki/Modello_evolutivo
[9] https://it.wikipedia.org/wiki/Modello_trasformazionale
[10] https://it.wikipedia.org/wiki/Prototipazione_rapida
[11] https://it.wikipedia.org/wiki/Modello_a_spirale
[12] https://it.wikipedia.org/wiki/Modello_a_fontana

- **è più importante avere software funzionante che documentazione** (bisogna rilasciare nuove versioni del software ad intervalli frequenti, e bisogna mantenere il codice semplice e avanzato tecnicamente, riducendo la documentazione al minimo indispensabile);
- **bisogna collaborare con i clienti e non solo rispettare il contratto** (la collaborazione diretta offre risultati migliori dei rapporti contrattuali);
- **bisogna essere pronti a rispondere ai cambiamenti e non solo seguire l'iniziale pianificazione** (quindi il team di sviluppo dovrebbe essere pronto, in ogni momento, a modificare le priorità di lavoro nel rispetto dell'obiettivo finale).

In sintesi, il manifesto, sottolinea l'importanza di tali principi **fermo restando il valore dei processi, degli strumenti, della documentazione, dei contratti e della pianificazione**.

Pratiche

Le singole pratiche applicabili all'interno di una metodologia agile sono decine e dipendono essenzialmente dalle necessità dell'azienda e dall'approccio del project manager. Nella scelta però bisogna tenere conto delle caratteristiche di ogni pratica, dei benefici apportati e delle conseguenze che esse comportano.

Le pratiche più diffuse sono simili fra di loro e possono essere raggruppate in categorie:

- **Automazione** - quando l'obiettivo è concentrarsi sulla programmazione senza dedicarsi alle attività collaterali, ad esempio, eliminare la documentazione aumentando il testing, ma non entrambe;
- **Comunicazione stretta** - quando l'unico vero aspetto nodale è la comunicazione interpersonale, fra tutti gli attori del progetto, cliente compreso;
- **Coinvolgimento del cliente** - ad esempio in Extreme Programming il coinvolgimento è totale, il cliente partecipa persino alle riunioni settimanali dei programmatori; in altri casi, il cliente è coinvolto in una prima fase di progettazione e poi non più; in altri ancora il cliente partecipa indirettamente e viene usato come test della versione rilasciata;
- **Progettazione e documentazione** - Le metodologie leggere introducono una iterazione nel ciclo di vita del progetto; quanta progettazione fare e quanta documentazione produrre, escludendo i casi estremi, è una scelta lasciata a chi gestisce il progetto;
- **Consegne frequenti** - Effettuare rilasci frequenti di versioni intermedie del software permette di ottenere più risultati contemporaneamente: si ricomincia l'iterazione avendo già a disposizione un blocco di codice funzionante in tutti i suoi aspetti, si offre al cliente "qualcosa con cui lavorare" e lo si distrae così da eventuali ritardi nella consegna del progetto completo, si usa il cliente come se fosse un test visto che utilizzerà il software e riscontrerà eventuali anomalie, si ottengono dal cliente informazioni più precise sui requisiti che probabilmente non sarebbe riuscito ad esprimere senza avere a disposizione utilità e carenze del progetto;
- **Pair programming** - Lo sviluppo viene fatto da coppie di programmatori che si alternano alla tastiera;
- **Miglioramento della conoscenza** - Nata con l'avvento della programmazione Object-Oriented, non è altro che la presa di coscienza della produzione di conoscenza che si fa in un'azienda man mano che si produce codice; questa conoscenza prodotta non deve andare perduta;
- **Retroingegneria** - Ossia ottenere, in maniera automatica, la documentazione a partire dal codice già prodotto;
- **Semplicità** - semplicità nel codice, semplicità nella documentazione, semplicità nella progettazione, semplicità nella modellazione;

- **Test** - Pratica diffusissima anche prima della nascita delle metodologie leggere, ha prodotto una letteratura vastissima ed una serie di approcci differenti come il Rapid Testing o il Pair Testing;
- **Controllo della versione** — Una delle conseguenze dirette dell'iterazione nella produzione è la necessità di un sistema per tracciare le modifiche al codice nel tempo, permettendo di tornare indietro, capire chi ha cambiato cosa e lavorare in parallelo senza sovrascriversi il lavoro a vicenda. Lo strumento oggi universalmente adottato è Git, spesso affiancato da piattaforme come GitHub o GitLab che ne facilitano la condivisione e la collaborazione in team.

2.6 Trasformazione di un programma sorgente

Ragionare in termini di istruzioni in Linguaggio Macchina è lungo e faticoso, porta facilmente ad errori e in pratica non consente di affrontare problemi complessi. Per questo motivo ha senso concepire LdP che fanno uso di istruzioni concettualmente più complesse ed evolute rispetto alle Istruzioni Macchina.

Tali linguaggi consentono di formulare algoritmi che sfruttano operazioni complesse e dunque di ragionare ad un più alto livello concettuale, permettendo di affrontare problemi più complessi. Ma un programma scritto con un LdP è inutile senza la possibilità di tradurlo automaticamente, sostituendo a ciascuna istruzione di alto livello una sequenza opportuna di istruzioni macchina.

Si definiscono traduttori i programmi che svolgono automaticamente questo compito, individuando anche eventuali errori sintattici presenti nel codice sorgente.

La traduzione può essere effettuata utilizzando 2 approcci distinti.

- la **compilazione**, in cui il traduttore (**compilatore**) accetta in input un programma sorgente scritto in un linguaggio ad alto livello e produce in output un programma eseguibile, equivalente a quello dato;
- l'**interpretazione**, in cui il traduttore (**interprete**) considera singolarmente ciascuna istruzione di alto livello, la traduce e la esegue per poi procedere con l'istruzione successiva.

2.6.1 La compilazione

La compilazione è quel processo di traduzione per cui il programma, scritto in un LdP, viene tradotto in un codice eseguibile per mezzo di un altro programma detto compilatore. Il compilatore prende in ingresso un programma scritto in un LdP (codice sorgente) ed esegue una serie di operazioni in modo da ottenere, in assenza di errori, un programma in Linguaggio Macchina e quindi direttamente eseguibile. In generale, i compilatori sono in grado di riconoscere alcune classi di errori presenti nel programma, e di suggerire in che modo correggerli.

I compilatori attuali dividono l'operazione di compilazione in 2 fasi principali:

- Fase di **front end:** il compilatore traduce il codice sorgente in un linguaggio intermedio.
- Fase di **back end**: determina la generazione del codice oggetto (normalmente il Linguaggio Macchina).

Stadio di front end

Questo stadio si suddivide in più fasi:

- **Analisi lessicale**. Attraverso un analizzatore lessicale, spesso chiamato scanner o lexer, il compilatore divide il codice sorgente in tanti pezzetti chiamati token.
  I token sono gli elementi minimi (non ulteriormente divisibili) di un linguaggio, ad esempio parole chiave (for, while), nomi di variabili (pippo), operatori (+, -, «).
- **Analisi sintattica.** L'analisi sintattica prende in ingresso la sequenza di token generata nella fase precedente ed esegue il controllo sintattico in modo da individuare eventuali errori sintattici.
- **Analisi semantica**. L'analisi semantica si occupa di controllare il significato delle istruzioni presenti nel codice in ingresso e descritto mediante l'albero sintattico. Controlli tipici di questa fase sono: il

type checking, ovvero il controllo di tipo, controllare che gli identificatori siano stati dichiarati prima di essere usati e così via.
Come supporto a questa fase viene creata una **tabella dei simboli** (symbol table) che contiene informazioni su tutti gli elementi simbolici incontrati quali **il nome**, **lo scope** (porzione di codice in cui il simbolo esiste), **il tipo** (se presente) etc. Il risultato di questa fase è l'albero sintattico astratto (AST).

- **Generazione del codice intermedio**. Dall'albero di sintassi (AST) viene generato il codice intermedio.

Stadio di back end

Anche lo stadio di back end si divide in più fasi:

- **Ottimizzazione del codice** intermedio.
- **Generazione del codice target**: in questa fase viene generato il codice nella forma del linguaggio target, spesso il Linguaggio Macchina.
- Inoltre, vengono anche collegate le **librerie** necessarie al corretto funzionamento del programma, tale processo viene anche detto **linking**.

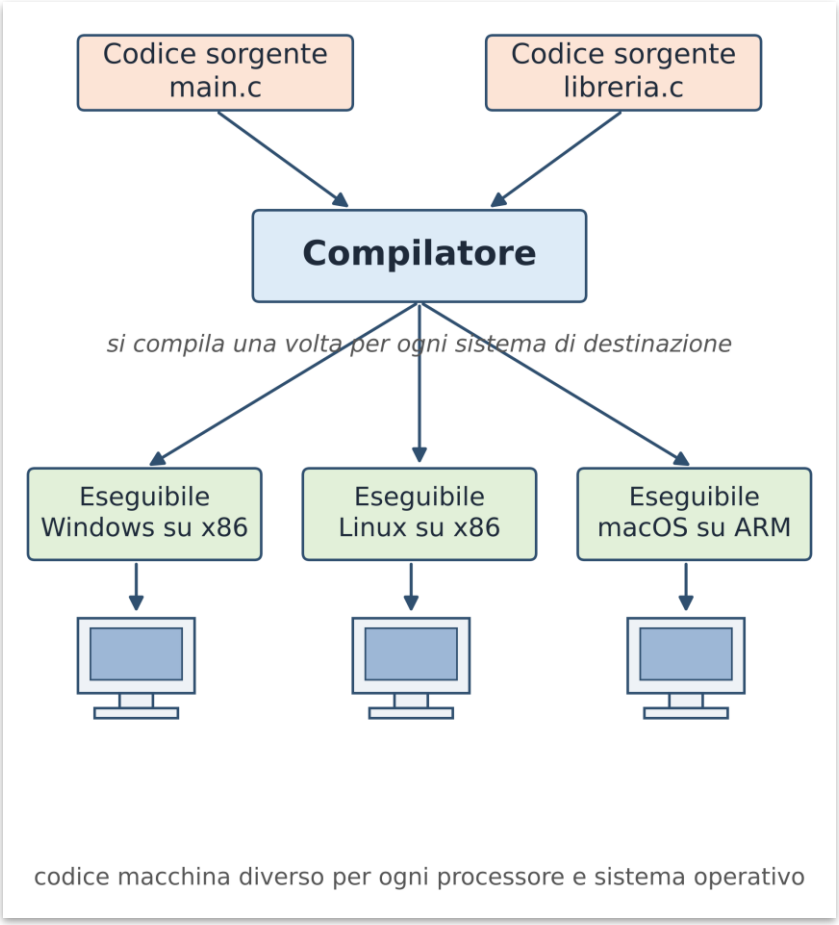


Figura 17 - Schema generale di compilazione di un programma con 2 codici sorgenti compilato per più di un sistema

2.6.2 Vantaggi e svantaggi della compilazione

La compilazione offre numerosi vantaggi:

- permette di ottenere **eseguibili velocissimi** nella fase di **run** (esecuzione) adattando l'eseguibile all'hardware a disposizione;
- la trasformazione avviene un'**unica volta** e, successivamente, il programma sarà pronto per essere eseguito tutte le volte che lo si desidera senza dover ripetere il processo di trasformazione;
- durante l'esecuzione del programma precedentemente compilato non è necessaria la presenza in memoria di alcun software di supporto alla trasformazione.

La compilazione presenta anche alcuni svantaggi:

- **è necessario compilare un eseguibile diverso per ogni sistema operativo o hardware (piattaforma)** sul quale si desidera rendere disponibile l'esecuzione ovvero viene a mancare la cosiddetta portabilità.
- in fase di sviluppo **è necessario ricompilare spesso il programma**, in quanto ogni modifica richiede la ricompilazione del codice sorgente modificato per poterlo provare, determinando una perdita di tempo rilevante durante la fase di sviluppo del software.

2.6.3 Il collegamento (linking)

Se il programma, come spesso accade, usa delle librerie, queste devono essere 'collegate' al programma. Lo strumento che effettua questa operazione è detto linker.

Esistono principalmente 2 tipi differenti di collegamento:

- **Collegamento (linking) statico**
  Le librerie utilizzate vengono incluse nell'eseguibile, che risulta così di grandi dimensioni, ma contiene tutto quanto necessario per la sua esecuzione. Se si rende necessaria una modifica ad una delle librerie, per correggere un errore o un problema di sicurezza, tutti i programmi che la usano con collegamento statico devono essere ricompilati con le nuove versioni della libreria.

- **Collegamento(linking) dinamico**
  Le librerie utilizzate sono caricate dal Sistema Operativo (SO) quando necessario; le librerie sono esterne al programma compilato e sono chiamate "DLL", Dynamic-link libraries nei sistemi Microsoft Windows, mentre "SO" Shared Object nei sistemi Unix-like.

  L'eseguibile è più compatto, ma dipende dalla presenza nel SO delle librerie utilizzate per poter essere eseguito. Con questa modalità, le librerie possono essere aggiornate a livello di SO, senza la necessità di ricompilare i programmi che le utilizzano. Diventa anche possibile usare diverse versioni della stessa libreria, o usare librerie personalizzate con caratteristiche specifiche.

  In genere si preferisce il collegamento dinamico, in modo da creare programmi piccoli e ridurre la memoria RAM occupata durante l'esecuzione, assumendo che le librerie necessarie siano già presenti nel sistema, o, talvolta, distribuendole insieme al programma e aggiunte al SO in fase di installazione del programma.

2.6.4 L'interpretazione

Un interprete è un programma in grado di eseguire altri programmi interpretando il codice sorgente senza doverlo compilare, cioè permette di eseguire le istruzioni scritte con un linguaggio ad alto livello, traducendole, di volta in volta, in istruzioni in linguaggio macchina.

Questa soluzione è stata adottata per cercare di eliminare il **problema della portabilità** (la dipendenza del linguaggio dalla piattaforma), adottando linguaggi il cui codice viene interpretato in fase di esecuzione.

In questo modo, **non c'è la necessità di una compilazione per ogni tipologia di sistema operativo** su cui viene eseguito il programma. Il grosso difetto di questi linguaggi è la **lentezza dell'esecuzione**; però hanno il grosso pregio di permetterne l'uso su SO differenti senza la necessità di modifiche. Si dice, in questo caso, che il programma è **portabile**.

La perdita di prestazioni è data dal doppio lavoro (interpretazione ed esecuzione) che è affidato alla macchina che si accinge ad eseguire il programma. Si tende ad utilizzare l'approccio interpretato nella fase di messa a punto di un programma per **evitare di effettuare numerose compilazioni** o quando si vuole creare software non ottimizzati ma che traggono più vantaggio dalla portabilità. Esempi di linguaggi di scripting sono quelli orientati al Web: PHP, Perl, Tcl/Tk e JavaScript.

2.6.5 Approcci ibridi: Bytecode e P-code

Una soluzione intermedia fra compilazione e interpretazione è stata introdotta nelle prime versioni di **Pascal** (compresa quella realizzata nel 1975 dal suo inventore, Niklaus Wirth) e successivamente adottata nei linguaggi **Java e Python**, con il **Bytecode**, e nei linguaggi **Visual Basic e .NET** di Microsoft con il **P-code**.

In entrambi i casi, il codice sorgente dei programmi non viene trasformato in linguaggio macchina, ma in un codice intermedio "**ibrido**" interpretato al momento dell'esecuzione del programma.

Il motivo di questo doppio passaggio è, da una parte, avere la portabilità dei linguaggi interpretati, dall'altra, con la pre-compilazione, avere una fase di interpretazione più semplice e quindi più veloce.

Nel caso del **Bytecode** di Java siamo di fronte a un vero linguaggio Assembly, che in origine doveva essere eseguito da un processore mai realizzato. Alcuni microprocessori moderni, come gli ARM dei dispositivi mobili, implementano nativamente molte istruzioni bytecode e sono quindi in grado di eseguire Bytecode Java come fosse linguaggio macchina.

Tuttavia, il codice intermedio è facile sia da interpretare che da compilare. Per questo motivo sia per Java che per i linguaggi .NET sono stati **sviluppati compilatori JIT** (Just In Time), che al momento del lancio di un programma Java o .NET compilano al volo il codice sorgente e mandano in esecuzione un codice macchina nativo, eliminando completamente la necessità dell'interprete e rendono i programmi veloci quasi quanto i corrispondenti programmi compilati.

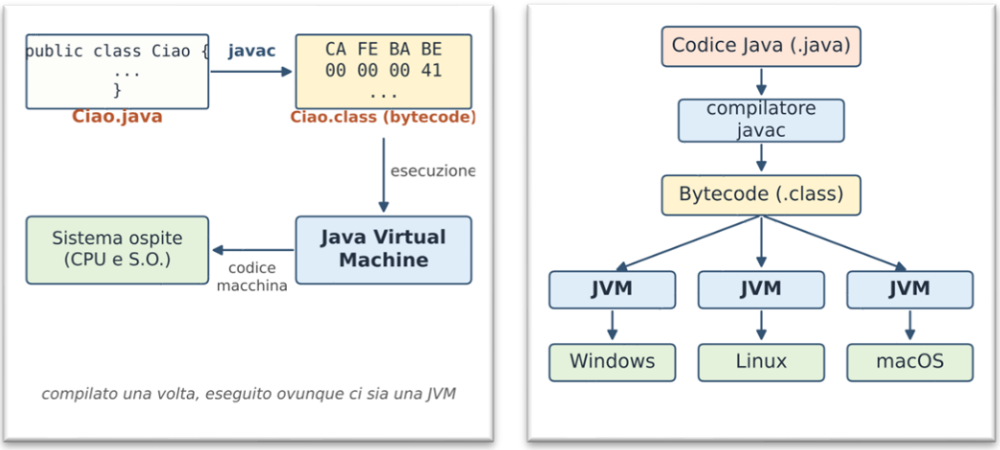


Figura 18 - Compilazione-Interpretazione nel caso del linguaggio Java

3 Il vibe coding e la Spec-Driven Development

Il termine vibe coding nasce da un post di Andrej Karpathy — ex direttore dell'intelligenza artificiale di Tesla e co-fondatore di OpenAI — nel febbraio 2025: descriveva un modo di programmare in cui ci si "abbandona alle vibrazioni", dimenticando quasi che il codice esista, affidandosi a un'intelligenza artificiale per tradurre in codice funzionante una richiesta espressa in linguaggio naturale, accettandone l'output così com'è e correggendo per approssimazioni successive attraverso il dialogo.

È un approccio molto rapido, ma con un limite evidente: senza una guida esplicita, il codice prodotto tende a essere fragile e poco controllato. Uno studio del dicembre 2025 (CodeRabbit) stima che circa il 45% del codice generato da AI in questo modo contenga vulnerabilità di sicurezza — un dato che spiega perché, nel giro di pochi mesi, siano nati diversi tentativi di dare struttura al processo.

Il più diffuso si chiama Spec-Driven Development (SDD): invece di partire subito dal prompt, si scrive prima una specifica formale, che l'intelligenza artificiale dovrà rispettare durante la generazione del codice.

Strumenti come GitHub Spec Kit, Kiro (Amazon) e OpenSpec organizzano questo processo in quattro fasi, ciascuna delle quali produce un file di testo (in formato **Markdown** - .md) che resta nel progetto, leggibile e modificabile come un qualsiasi altro file di codice:

3.1 Le quattro fasi

Constitution — scritto una sola volta all'inizio del progetto, fissa i principi non negoziabili a cui l'intelligenza artificiale dovrà attenersi in ogni fase successiva (per esempio: quale approccio ai test adottare, quali convenzioni architetturali rispettare). *(vedi appendice)*

Specify — la specifica della singola funzionalità da realizzare: cosa deve fare, per chi, e perché. Si concentra sul "cosa" e sul "perché", senza mai citare tecnologie specifiche; per ogni esigenza dell'utente (una User Story, nella forma "Come [ruolo], voglio [capacità], così da [beneficio]") elenca criteri di accettazione verificabili uno per uno. *(vedi appendice)*

Plan — la traduzione tecnica della specifica: qui, e solo qui, entrano in gioco le scelte implementative (linguaggio, framework, architettura, struttura dei dati), sempre nel rispetto dei principi fissati dalla Constitution. *(vedi appendice)*

Tasks — la scomposizione del piano in una lista ordinata di compiti piccoli e verificabili, ciascuno tracciabile fino al requisito che lo ha generato, pronti per essere eseguiti uno alla volta, con revisione umana ad ogni passaggio. *(vedi appendice)*

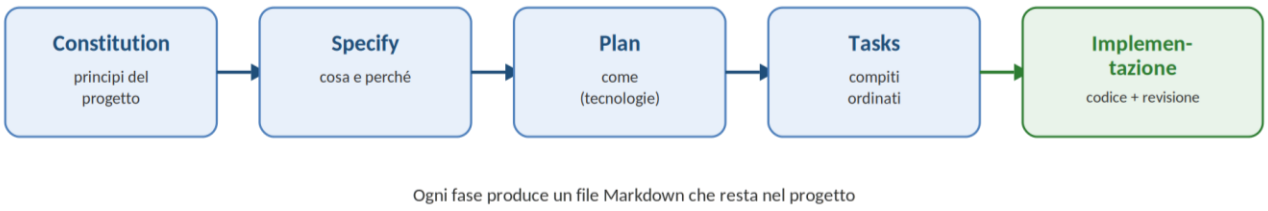


Figura 19 - Il flusso delle quattro fasi della Spec-Driven Development

Solo al termine di queste quattro fasi si passa all'implementazione vera e propria: l'intelligenza artificiale genera il codice, il programmatore lo rivede, e il ciclo si ripete per ogni nuova funzionalità (la Constitution, una volta scritta, resta invece valida per tutto il progetto).

Va notato che la Spec-Driven Development non è pensata come sostituto del vibe coding, quanto piuttosto come una cornice che lo contiene: la specifica pone i vincoli entro cui il dialogo in linguaggio naturale con l'intelligenza artificiale può muoversi liberamente.

3.2 Come si scrivono in pratica: il progetto che vi accompagnerà per tre anni

Ogni fase si avvia con un comando che inizia con "/" (uno "slash command"), che lo strumento AI utilizzato (Copilot, Claude Code, Cursor...) riconosce e traduce in un'istruzione strutturata. Proviamo a scrivere le prime quattro fasi per il progetto che svilupperete insieme in classe nei prossimi tre anni: un gioco sull'ambiente basato sull'Agenda 2030.

Si parte dalla Constitution, scritta una sola volta e valida per tutti e tre gli anni:

```
/constitution Il progetto è un gioco in Python sull'ambiente e sull'Agenda 2030 e dura tre anni. I dati della partita vanno sempre tenuti separati dalla logica di gioco; ogni funzione deve avere un nome chiaro e un commento che ne spieghi lo scopo; ogni nuova funzionalità deve continuare a far funzionare quanto già costruito negli anni precedenti; una tecnica nuova si introduce solo quando il progetto ne mostra la necessità.
```

Che porta a un documento come questo:

- **Separazione dati/logica**: lo stato della partita (progressi, scelte, punteggio) è sempre separato dal codice che lo elabora, così il formato di salvataggio può cambiare (file di testo → JSON → database) senza dover riscrivere la logica di gioco.
- **Si estende, non si riscrive**: ogni funzionalità aggiunta deve continuare a far funzionare quanto già costruito negli anni precedenti.
- **Codice comprensibile prima che elegante**: ogni funzione ha un nome chiaro e un commento che ne spiega lo scopo, anche a costo di essere più verbosi del necessario.
- **Una tecnica nuova si introduce solo quando serve**: mai come argomento isolato, ma nel momento in cui il progetto stesso ne mostra la necessità.

Con due emendamenti già previsti per il futuro: in quarta la Constitution verrà aggiornata per includere database e interfaccia web; in quinta per i requisiti di sicurezza e l'integrazione dell'AI nel secondo quadrimestre.

Per la prima funzionalità dell'anno — permettere di iniziare una partita:

```
/specify Il giocatore deve poter iniziare una partita inserendo il proprio nome, così che il gioco ricordi i suoi progressi da una sessione all'altra. Un nuovo giocatore parte con un punteggio di sostenibilità pari a 0; chi rientra con lo stesso nome ritrova il punteggio raggiunto. Il nome non può essere vuoto.
```

Risultato:

Come giocatore, voglio poter iniziare una partita inserendo il mio nome, così che il gioco possa ricordare i miei progressi da una sessione all'altra.

Criteri di accettazione:

- Il gioco chiede il nome del giocatore all'avvio e non accetta un nome vuoto.
- Un nuovo giocatore viene accolto con un messaggio di benvenuto e parte con punteggio di sostenibilità 0.
- Un giocatore che rientra con lo stesso nome viene accolto con "Bentornato" e ritrova il punteggio raggiunto.
- I progressi restano conservati anche dopo la chiusura del gioco.

Poi il Plan, con le scelte tecniche per questo primo anno (il JSON arriverà più avanti):

```
/plan Per ora salva su un semplice file di testo, uno per giocatore, nella cartella salvataggi; il nome del file è il nome del giocatore in minuscolo, senza spazi. Solo due funzioni devono leggere e scrivere il file, così il resto del gioco non sa come i dati sono salvati. Se il file non esiste ancora, il caricamento lo gestisce con try/except.
```

Risultato: due funzioni, salva_partita(nome, punteggio) e carica_partita(nome), sono le uniche a toccare il file — tutto il resto del gioco le usa senza sapere come i dati sono effettivamente salvati (primo principio della Constitution). Il caricamento gestisce con un try/except il caso in cui il file non esista ancora, invece di controllarlo prima con una verifica separata.

Infine i Tasks:

```
/tasks Compiti piccoli, da fare uno alla volta, ciascuno verificabile da solo, con un collaudo finale.
```

Risultato:

- **T001** — Scrivere salva_partita(nome, punteggio): crea la cartella salvataggi se manca e crea o sovrascrive il file del giocatore.

- **T002** — Scrivere carica_partita(nome): legge il file, gestendo con try/except il caso in cui non esista ancora.
- **T003** — Scrivere il programma principale: chiede il nome finché non è vuoto, prova a caricare la partita e mostra "Benvenuto" (punteggio 0, partita salvata) oppure "Bentornato" (punteggio recuperato).
- **T004** — Collaudo manuale: avviare il programma due volte con lo stesso nome, la seconda volta scritto con maiuscole diverse, e verificare che compaia "Bentornato" con il punteggio recuperato.

Da qui si passa all'implementazione: il codice delle funzioni viene scritto (dall'intelligenza artificiale o da voi), rivisto insieme, e un task alla volta diventa il primo mattoncino del progetto che vi accompagnerà fino alla quinta.

4 Appendice — Esempio fasi Vibe Coding

```
# Constitution — Progetto "BiblioClasse"

**Versione:** 1.0.0 · **Data:** 23/09/2026 · **Team:** 3C, gruppo 2

## Che cos'è questo documento?

La Constitution è l'insieme delle **regole fondamentali** del nostro progetto.
Come la Costituzione di uno Stato, sta "sopra" tutte le altre decisioni: se una scelta la
contraddice, la scelta è sbagliata.
Vale per tutti: per noi, per i compagni che entrano nel gruppo dopo, e anche per
l'intelligenza artificiale che ci aiuta a scrivere il codice.

---

## I nostri principi

### Art. 1 — Prima si pensa, poi si scrive codice
Prima di programmare una funzionalità, scriviamo in italiano **cosa deve fare** e **per
chi**.
Se qualcosa non è chiaro, lo scriviamo come domanda: `[DA CHIARIRE]`.

> *Perché?* Correggere un'idea sbagliata costa 5 minuti;
> correggere 200 righe di codice sbagliato costa ore.

### Art. 2 — Ogni funzione ha il suo test
Per ogni funzione importante scriviamo almeno un test che ne controlla il risultato, sia
nei casi normali sia in quelli strani (lista vuota, numero negativo, testo mancante...).

> *Perché?* Così, se modifichiamo qualcosa, ci accorgiamo subito
> se abbiamo rotto qualcos'altro.

### Art. 3 — Semplice è meglio
- Una funzione fa **una sola cosa** e non supera le 30 righe
  di codice (docstring e commenti esclusi)
- Nomi chiari in italiano: `calcola_ritardo()`, non `f2()` o `x()`
- Niente codice "che forse servirà in futuro"

> *Perché?* Il codice si legge molte più volte di quante si scriva.

### Art. 4 — I dati personali sono sacri
- Nel programma salviamo solo nome, cognome e classe: niente altro
```

```
- Nessuna password scritta nel codice
- I dati di prova usano nomi inventati, mai quelli di compagni veri

> *Perché?* I dati delle persone vanno protetti, anche in un progetto
> scolastico (lo dice anche il GDPR).

### Art. 5 — Il codice si spiega
Ogni file inizia con un commento che dice a cosa serve.
Ogni funzione ha una docstring che spiega cosa riceve e cosa restituisce.

> *Perché?* Tra un mese nemmeno noi ricorderemo come funziona.

---

## Strumenti che usiamo

| Cosa              | Strumento               |
|-------------------|-------------------------|
| Linguaggio        | Python 3.12             |
| Dati              | file JSON               |
| Test              | pytest                  |
| Condivisione      | GitHub                  |

---

## Come lavoriamo

1. Scriviamo cosa deve fare la funzionalità (Art. 1)
2. Scriviamo il test (Art. 2)
3. Scriviamo il codice finché il test passa
4. Un compagno rilegge il codice prima di unirlo al progetto

**Il codice scritto con l'AI segue le stesse regole:**
va letto, capito e testato. Se non sappiamo spiegarlo, non lo usiamo.

---

## Come si cambia la Constitution

Le regole si possono cambiare, ma non di nascosto:
1. chi vuole cambiarla lo propone al gruppo spiegando il motivo;
2. il gruppo decide insieme (con il via libera del docente);
3. si aggiorna il numero di versione e la data in cima al file.
```

Specifica — Prestito e restituzione dei libri

```
**Progetto:** BiblioClasse · **Numero:** 001 · **Data:** 23/09/2026
**Autori:** 3C, gruppo 2 · **Stato:** bozza

## Che cos'è questo documento?

Una specifica descrive **una funzionalità** del programma: **cosa** deve fare e **per
chi**, ma NON **come** la programmeremo.
Qui non troverai codice, nomi di variabili o librerie Python: quelli arrivano dopo, nel
piano tecnico.

> *Regola d'oro:* se una frase parla di `for`, `dizionari` o `file JSON`,
> non appartiene a questo documento.
```

```
---

## 1. Il problema

In classe abbiamo circa 60 libri che i compagni si prestano.
Oggi usiamo un foglio di carta appeso alla bacheca: si perde,
non si capisce chi ha cosa, e nessuno sa quali libri sono in ritardo.

## 2. Chi usa questa funzionalità

- **Bibliotecario di classe:** uno studente scelto a turno, registra prestiti e restituzioni
- **Compagni:** vogliono sapere se un libro è disponibile

---

## 3. Storie d'uso

Una storia d'uso racconta una situazione reale, dal punto di vista di chi usa il programma.
Ogni storia ha degli **scenari di verifica** scritti nella forma *Dato… Quando… Allora…*:
diventeranno i nostri test.

### Storia 1 — Aggiungere un libro al catalogo (priorità ALTA)
*Come bibliotecario, voglio inserire un libro nel catalogo, così può essere prestato ai
compagni.*

> *Nota:* questa storia serve sia per il **primo caricamento**
> dei libri che la classe possiede già, sia per i **libri nuovi**
> che arriveranno durante l'anno. È la stessa operazione:
> all'inizio la faremo tante volte di fila, poi ogni tanto.

**Scenari di verifica:**
1. **Dato** che il catalogo è vuoto,
   **quando** inserisco "Il nome della rosa" di Umberto Eco,
   **allora** il libro compare nel catalogo come "disponibile"
   con il codice B-001.
2. **Dato** che nel catalogo c'è già una copia di "Il nome della rosa",
   **quando** inserisco una seconda copia dello stesso libro,
   **allora** il programma la accetta e le assegna un codice diverso (B-002).
3. **Dato** che sto inserendo un libro,
   **quando** lascio vuoto il titolo,
   **allora** il programma non lo salva e mi chiede di completarlo.

### Storia 2 — Prestare un libro (priorità ALTA)
*Come bibliotecario, voglio registrare che un compagno prende un libro, così sappiamo
sempre chi lo ha.*

**Scenari di verifica:**
1. **Dato** che "Il nome della rosa" è disponibile,
   **quando** lo presto a Giulia Rossi,
   **allora** il libro risulta "in prestito a Giulia Rossi" con la data di oggi.
2. **Dato** che "Il nome della rosa" è già in prestito,
   **quando** provo a prestarlo a un altro compagno,
   **allora** il programma mi avvisa che non è disponibile.

### Storia 3 — Restituire un libro (priorità ALTA)
*Come bibliotecario, voglio registrare la restituzione, così il libro torna disponibile
per gli altri.*

**Scenari di verifica:**
1. **Dato** che Giulia ha in prestito un libro,
   **quando** lo restituisce,
```

```
   **allora** il libro torna "disponibile".
2. **Dato** che un libro NON è in prestito,
   **quando** provo a registrarne la restituzione,
   **allora** il programma mi avvisa dell'errore.

### Storia 4 — Vedere i ritardi (priorità MEDIA)
*Come bibliotecario, voglio vedere l'elenco dei libri non restituiti in tempo, così posso
ricordarlo ai compagni.*

**Scenari di verifica:**
1. **Dato** che un libro è in prestito da 35 giorni,
   **quando** apro l'elenco dei ritardi,
   **allora** il libro compare con il nome di chi lo ha e i giorni di ritardo.

---

## 4. Casi particolari

> *Perché?* I programmi di solito non si rompono nei casi normali,
> ma in quelli strani. Meglio pensarci adesso.

- Cosa succede se cerco un libro che non esiste nell'elenco?
- Cosa succede se scrivo il nome del compagno con errori di battitura?
- Un compagno può avere più libri contemporaneamente?
  `[DA CHIARIRE: c'è un limite? 2? 3?]`
- Cosa succede se abbiamo **due copie dello stesso libro**?
  Devono essere distinguibili, altrimenti non sappiamo quale copia
  ha Giulia. Per questo ogni copia riceve un **codice** (B-001, B-002...)
  da scrivere anche su un'etichetta attaccata al libro.
- Cosa succede se inserisco per sbaglio un libro due volte?
  `[DA CHIARIRE: serve poter eliminare o correggere un libro inserito?]`
- Per il primo caricamento, inseriamo i libri uno alla volta o da un elenco già pronto?
  `[DA CHIARIRE: per ora uno alla volta; l'elenco sarà una spec futura?]`
- Errori di battitura nei nomi: le maiuscole vengono uniformate ("giulia" = "Giulia"); gli
altri errori di battitura restano possibili `[DA CHIARIRE: basta così?]`

---

## 5. Requisiti

Ogni requisito è una frase breve, numerata e **verificabile**: deve essere possibile dire
con certezza se è rispettato o no.

- **R-01** Il programma DEVE permettere di prestare un libro disponibile.
- **R-02** Il programma DEVE impedire di prestare un libro già in prestito.
- **R-03** Il programma DEVE registrare la data di ogni prestito.
- **R-04** Il programma DEVE permettere di registrare una restituzione.
- **R-05** Il programma DEVE mostrare i libri in prestito da più di `[DA CHIARIRE: 30
giorni? un mese?]`.
- **R-06** Il programma DEVE salvare solo nome, cognome e classe dello studente
(Constitution, Art. 4).
- **R-07** Il programma DEVE permettere di aggiungere un libro indicando titolo e autore.
- **R-08** Il programma DEVE assegnare a ogni copia un codice unico.
- **R-09** Un libro appena inserito DEVE risultare "disponibile".
- **R-10** Il programma NON DEVE accettare un libro senza titolo.
- **R-11** Il programma DEVE impedire a uno studente di avere più di `[valore provvisorio:
3]` libri in prestito contemporaneamente.
- **R-12** I dati DEVONO restare salvati anche dopo la chiusura del programma.

---
```

```
## 6. Le "cose" del programma

Quali informazioni deve conoscere il programma? (Solo i concetti, non il codice.)

- **Libro:** codice (unico per ogni copia), titolo, autore, stato (disponibile / in prestito)
- **Studente:** nome, cognome, classe
- **Prestito:** quale libro, a quale studente, data di inizio, data di restituzione

---

## 7. Come capiamo che abbiamo finito

- Tutti gli scenari di verifica delle storie 1, 2 e 3 funzionano
- Un compagno che non ha partecipato al progetto riesce a registrare un prestito in meno di 1 minuto, senza chiedere aiuto
- Nessun `[DA CHIARIRE]` è rimasto aperto
- Il bibliotecario riesce a inserire tutti i libri della classe in una sola ora di lezione

## 8. Cosa NON fa questa funzionalità

> *Perché?* Dire cosa resta fuori evita di allargare il lavoro all'infinito.

- Non invia email o notifiche ai compagni in ritardo
- Non gestisce l'acquisto di nuovi libri
- Non ha un login con password (per ora)
```

Piano tecnico — Catalogo, prestito e restituzione dei libri

```
**Progetto:** BiblioClasse · **Spec di riferimento:** 001 · **Data:** 30/09/2026
**Autori:** 3C, gruppo 2 · **Stato:** bozza

## Che cos'è questo documento?

Il piano traduce la specifica in **scelte tecniche**: linguaggio, struttura dei file, formato dei dati, funzioni da scrivere.
Qui il codice non c'è ancora, ma c'è tutto quello che serve per scriverlo.

> *Differenza chiave:* la spec si può mostrare al preside,
> il piano si mostra a un programmatore.

---

## 1. Riassunto

Un programma Python che si usa da terminale con un menu a numeri.
I dati (libri e prestiti) sono salvati in un file JSON, così non si perdono quando chiudiamo il programma.

---

## 2. Decisioni sui punti `[DA CHIARIRE]`

Prima di progettare, il gruppo ha sciolto i dubbi della spec (discussione del 28/09, approvata dal docente):

| Dubbio nella spec                          | Decisione                          |
```

```
|-----------------------------------------|---------------------------------|
| Quanti libri può avere uno studente?    | Massimo 3 contemporaneamente    |
| Dopo quanto un prestito è in ritardo?   | Dopo 30 giorni                  |
| Si può correggere un libro inserito?    | Sì, titolo e autore             |
| Primo caricamento da elenco?            | No, uno alla volta (per ora)    |
```

> *Perché?* Se si inizia a programmare con dei dubbi aperti,
> ognuno li risolve a modo suo e il codice non torna.

Nota: queste decisioni vanno riportate anche nella spec, aggiornandone la data.

---

3. Contesto tecnico

Cosa	Scelta
Linguaggio	Python 3.12
Interfaccia	Menu testuale nel terminale
Salvataggio dati	Un file `biblioteca.json`
Test	pytest
Librerie esterne	Nessuna (solo `json` e `datetime`)
Dimensioni previste	circa 100 libri, 25 studenti
Configurazione test	File `pytest.ini`

---

4. Controllo della Constitution

Prima di andare avanti, verifichiamo di rispettare le nostre regole.

- [x] **Art. 1** — Esiste una spec approvata (001)
- [x] **Art. 2** — Ogni requisito ha almeno un test previsto (vedi sez. 8)
- [x] **Art. 3** — Nessuna libreria esterna, funzioni piccole con nomi chiari
- [x] **Art. 4** — Salviamo solo nome, cognome e classe; dati di prova inventati
- [x] **Art. 5** — Ogni file e funzione avrà la sua docstring

> *Perché?* Se una casella non si può spuntare, ci fermiamo e ne
> discutiamo **adesso**, non dopo aver scritto il codice.

---

5. Struttura dei file

```
biblioclasse/
├── main.py
├── catalogo.py
├── prestiti.py
├── archivio.py
├── pytest.ini            ← dice a pytest dove trovare i nostri file
├── biblioteca.json       ← creato automaticamente al primo salvataggio
├── tests/
│   ├── dati_prova.json   ← dati inventati per i test (Art. 4)
│   ├── test_archivio.py
│   ├── test_catalogo.py
│   └── test_prestiti.py
└── README.md
```

Regola importante: solo `main.py` usa `input()` e `print()`.
Le funzioni degli altri file **ricevono dati e restituiscono risultati**.

> *Perché?* Una funzione che chiede dati all'utente non si può testare
> in automatico: pytest non sa "scrivere sulla tastiera".
> Separando il menu dalla logica, possiamo testare tutto il resto.

---

6. Formato dei dati

Esempio del file `biblioteca.json`:

```
    {
      "libri": [
        {"codice": "B-001", "titolo": "Il nome della rosa", "autore": "Umberto Eco"},
        {"codice": "B-002", "titolo": "Il nome della rosa", "autore": "Umberto Eco"}
      ],
      "prestiti": [
        {
          "codice_libro": "B-001",
          "studente": {"nome": "Giulia", "cognome": "Rossi", "classe": "3C"},
          "data_inizio": "2026-09-23",
          "data_restituzione": null
        }
      ]
    }
```

Tre scelte da notare:

1. **Lo stato del libro non viene salvato, si calcola.**
   Un libro è "in prestito" se esiste un suo prestito con `data_restituzione` uguale a `null`. Altrimenti è "disponibile".
   > *Perché?* Se salvassimo lo stato in due posti, prima o poi
   > i due posti direbbero cose diverse. Un'informazione, un posto solo.
2. **I prestiti restituiti non si cancellano: ** si compila la data di restituzione. Così resta lo storico.
3. **Le date sono scritte come `AAAA-MM-GG`**, un formato che Python sa leggere e che si può ordinare alfabeticamente.

---

7. Le funzioni da scrivere

Questo è il "contratto" di ogni funzione: cosa riceve, cosa restituisce, quando segnala un errore. Chi la scrive e chi la usa si mettono d'accordo qui.

`catalogo.py`

Funzione	Restituisce	Errore (`ValueError`) se...
`nuovo_codice(dati)`	es. `"B-003"`	—
`aggiungi_libro(dati, titolo, autore)`	il codice creato	il titolo è vuoto
`correggi_libro(dati, codice, titolo, autore)` `[ATTESA]`	niente	il codice non esiste
`trova_libro(dati, codice)`	il libro	il codice non esiste

`[ATTESA]`: da scrivere solo se il gruppo conferma che serve > (vedi sez. 2 e tasks T024-T025).

`prestiti.py`

Funzione	Restituisce	Errore (`ValueError`) se...

```
| `e_disponibile(dati, codice)`              | `True` / `False`  | il codice non esiste           |
| `presta_libro(dati, codice, studente, oggi)` | niente          | libro non disponibile, oppure lo
studente ha già 3 libri |
| `restituisci_libro(dati, codice, oggi)`    | niente            | il libro non è in prestito     |
| `libri_in_ritardo(dati, oggi)`             | lista di prestiti | —                              |
| `prestito_in_corso(dati, codice)`          | il prestito o `None` | —                           |
| `libri_dello_studente(dati, studente)`     | un numero         | —                              |
| `giorni_di_prestito(prestito, oggi)`       | un numero di giorni| —                             |
```

> *Perché passiamo `oggi` come parametro invece di leggere la data
> dal computer?* Per poter testare un ritardo di 35 giorni senza
> aspettare 35 giorni: nel test scegliamo noi che giorno è "oggi".

`archivio.py`

```
| Funzione                | Cosa fa                                              |
|-------------------------|------------------------------------------------------|
| `carica(percorso)`      | legge il JSON; se il file non esiste, parte vuoto |
| `salva(dati, percorso)` | scrive il JSON                                       |
```

---

8. Dai requisiti ai test

Ogni requisito della spec deve finire da qualche parte nel codice e deve avere il suo test. Questa tabella ci assicura di non dimenticarne nessuno.

```
| Requisito | Funzione            | Test                                          |
|-----------|---------------------|-----------------------------------------------|
| R-01      | `presta_libro`      | `test_presta_libro_disponibile`               |
| R-02      | `presta_libro`      | `test_non_presta_libro_gia_in_prestito`       |
| R-03      | `presta_libro`      | `test_prestito_registra_data`                 |
| R-04      | `restituisci_libro` | `test_restituzione_rende_disponibile`,
`test_non_restituisce_libro_non_in_prestito` |
| R-05      | `libri_in_ritardo`  | `test_ritardo_dopo_i_giorni_previsti`,
`test_prestito_recente_non_in_ritardo`, `test_esattamente_al_limite_non_e_ritardo` |
| R-06      | `presta_libro`      | `test_studente_ha_solo_nome_cognome_classe`   |
| R-07      | `aggiungi_libro`    | `test_aggiungi_libro`                         |
| R-08      | `nuovo_codice`      | `test_primo_codice_e_B001`, `test_codice_successivo`,
`test_due_copie_hanno_codici_diversi` |
| R-09      | `e_disponibile`     | `test_libro_nuovo_e_disponibile`              |
| R-10      | `aggiungi_libro`    | `test_non_accetta_titolo_vuoto`,
`test_non_accetta_titolo_di_soli_spazi` |
| R-11      | `presta_libro`      | `test_limite_libri_per_studente`              |
| R-12      | `carica`, `salva`   | `test_file_inesistente_da_archivio_vuoto`,
`test_salvo_e_ricarico_gli_stessi_dati` |
```

---

9. Eccezioni alla Constitution

Nessuna. Se in futuro servisse violare un articolo, lo scriveremo qui spiegando il motivo e perché non c'era un'alternativa più semplice.

---

10. Prossimo passo

Dal piano ricaveremo i **tasks**: l'elenco dei piccoli lavori da fare, in ordine, ognuno assegnato a un membro del gruppo.

Tasks — Catalogo, prestito e restituzione dei libri

```
**Progetto:** BiblioClasse · **Spec:** 001 · **Piano:** 001 · **Data:** 05/10/2026
**Autori:** 3C, gruppo 2

## Che cos'è questo documento?

È la **lista delle cose da fare**, in ordine, per trasformare il piano in un programma funzionante. Ogni task è un lavoro piccolo (al massimo un'ora di laboratorio) e si può spuntare quando è finito.

## Legenda

- `[P]` — si può fare **in parallelo** con gli altri task `[P]` della stessa fase (toccano file diversi, quindi non ci si pesta i piedi)
- `[ATTESA]` — dipende da una decisione **non ancora presa**: si fa per ultimo, dopo che il gruppo ha deciso
- 🔴 — task di **test**: va scritto e fatto girare **prima** del codice, e deve **fallire** (Constitution, Art. 2)

---

## ⚠️ Punti ancora aperti

Alcune decisioni del piano (sez. 2) **non sono ancora definitive**.
Per non restare fermi, lavoriamo così:

| Dubbio                        | Come procediamo intanto                              |
|-------------------------------|------------------------------------------------------|
| Massimo libri per studente    | Costante provvisoria `MAX_LIBRI = 3`                 |
| Giorni prima del ritardo      | Costante provvisoria `GIORNI_PRESTITO = 30`          |
| Correggere un libro inserito | Task in `[ATTESA]`, spostato in fondo                |
| Caricamento da elenco         | Fuori da questa spec: non ci sono task               |

> *Perché funziona?* Se il numero "3" è scritto in un solo posto,
> cambiarlo in "2" domani richiede dieci secondi.
> Se fosse sparso in cinque funzioni, sarebbe una caccia al tesoro.
> Anche i test useranno le costanti, non i numeri scritti a mano.

---

## Fase 1 — Preparazione

- [ ] **T001** Creare la struttura di cartelle e file del piano (sez. 5),
      compreso il file `pytest.ini`, ognuno con il commento iniziale
      che dice a cosa serve (Art. 5)
- [ ] **T002** Installare pytest e verificare che il comando `pytest`
      funzioni (anche se non trova ancora test)
- [ ] **T003** `[P]` Creare `tests/dati_prova.json` con 5 libri
      e 3 studenti **inventati** (Art. 4)
- [ ] **T004** `[P]` Scrivere in cima a `prestiti.py` le costanti
      provvisorie, con un commento ben visibile:
      `# PROVVISORIO: in attesa di decisione del gruppo`

## Fase 2 — Archivio (leggere e salvare i dati)

- [ ] **T005** 🔴 Scrivere i test di `carica` e `salva`:
      salvo, ricarico, ottengo gli stessi dati;
      se il file non esiste, ottengo un archivio vuoto
- [ ] **T006** Scrivere `carica` e `salva` in `archivio.py`
      finché i test di T005 passano
```

```
## Fase 3 — Catalogo

- [ ] **T007** 🔴 `[P]` Test di `nuovo_codice`:
      catalogo vuoto → `B-001`; dopo `B-001` → `B-002` (R-08)
- [ ] **T008** 🔴 `[P]` Test di `aggiungi_libro`:
      il libro compare nel catalogo (R-07);
      titolo vuoto → errore (R-10);
      due copie dello stesso titolo → codici diversi (R-08)
- [ ] **T009** Scrivere `nuovo_codice`
- [ ] **T010** Scrivere `aggiungi_libro` e `trova_libro`

## Fase 4 — Prestiti

- [ ] **T011** 🔴 Test di `e_disponibile`:
      un libro appena aggiunto è disponibile (R-09)
- [ ] **T012** 🔴 Test di `presta_libro`:
      libro disponibile → prestito registrato con la data (R-01, R-03);
      libro già prestato → errore (R-02);
      studente con `MAX_LIBRI` libri → errore;
      lo studente salvato ha solo nome, cognome e classe (R-06)
- [ ] **T013** 🔴 Test di `restituisci_libro`:
      dopo la restituzione il libro è disponibile (R-04);
      libro non in prestito → errore
- [ ] **T014** 🔴 Test di `libri_in_ritardo`:
      prestito di `GIORNI_PRESTITO + 5` giorni fa → compare;
      prestito di ieri → non compare;
      prestito di esattamente `GIORNI_PRESTITO` giorni fa → non compare (R-05)
- [ ] **T015** Scrivere `e_disponibile`
- [ ] **T016** Scrivere `presta_libro`
- [ ] **T017** Scrivere `restituisci_libro`
- [ ] **T018** Scrivere `libri_in_ritardo`

## Fase 5 — Il menu

- [ ] **T019** Scrivere in `main.py` il menu con le voci:
      1) Aggiungi libro 2) Presta 3) Restituisci 4) Ritardi 0) Esci
- [ ] **T020** Fare in modo che ogni errore (`ValueError`) venga
      mostrato come messaggio chiaro, senza far chiudere il programma
- [ ] **T021** Prova sul campo: un compagno di un altro gruppo
      registra un prestito senza aiuto, in meno di 1 minuto
      (spec, sez. 7)

## Fase 6 — Decisioni e chiusura

- [ ] **T022** Il gruppo prende le decisioni rimaste aperte
      e aggiorna spec e piano
- [ ] **T023** Aggiornare le costanti con i valori definitivi,
      togliere il commento `PROVVISORIO` e rilanciare **tutti** i test
- [ ] **T024** `[ATTESA]` 🔴 Test di `correggi_libro`
      (solo se il gruppo decide che serve)
- [ ] **T025** `[ATTESA]` Scrivere `correggi_libro` e aggiungere
      la voce al menu
- [ ] **T026** Scrivere il `README.md`: come si avvia il programma
- [ ] **T027** Controllo finale: tutti i test passano,
      nessun `[DA CHIARIRE]` aperto, checklist della Constitution rispettata

---
```

Ordine e dipendenze

```
    Fase 1 → Fase 2 → Fase 3 → Fase 4 → Fase 5 → Fase 6
                        |         |
                        └── i test (🔴) prima del codice, sempre
```

- Le fasi 3 e 4 hanno bisogno dell'archivio (fase 2) solo per i dati di prova: i loro test possono partire anche prima, usando dati scritti nel test.
- Il menu (fase 5) si scrive per ultimo: usa tutte le altre funzioni.

Proposta di divisione del lavoro

Studente	Si occupa di	Task principali
A	Archivio e menu	T005-T006, T019-T021
B	Catalogo	T007-T010
C	Prestiti	T011-T018
Tutti	Preparazione e chiusura	T001-T004, T022-T027

> *Regola del gruppo:* chi finisce un task lo spunta e avvisa gli altri.
> Chi scrive il codice di un task NON è chi rilegge il codice
> prima di unirlo al progetto (Constitution, "Come lavoriamo", punto 4).

Il codice

Tre punti che meritano attenzione in classe.

Il codice ha "tradito" leggermente il piano. Scrivendolo sono servite tre piccole funzioni di aiuto non previste nella sezione 7 (prestito_in_corso, libri_dello_studente, giorni_di_prestito), e il prestito ora uniforma le maiuscole dei nomi, così "giulia" e "Giulia" contano come la stessa persona ai fini del limite. È del tutto normale: l'implementazione scopre sempre qualcosa. Ma la lezione è che queste scoperte vanno riportate nel piano, non lasciate solo nel codice.

Il test sul confine. test_esattamente_al_limite_non_e_ritardo controlla che al 30° giorno il libro non sia ancora in ritardo, perché la spec dice "più di 30 giorni". È il tipico errore da > contro >= che i ragazzi fanno spesso, e qui si vede perché i casi limite vanno testati esplicitamente.

L'ordine test-prima-del-codice. Nei file consegnati test e codice sono entrambi già scritti, ma in laboratorio l'Art. 2 chiede di vedere prima il test fallire. Per renderlo possibile, potrei preparare una versione per l'esercitazione: stessi file di test, ma con le funzioni di catalogo.py e prestiti.py ridotte alla sola docstring e a un pass. Gli studenti lanciano pytest, vedono tutto rosso, e lo fanno diventare verde una funzione alla volta seguendo l'ordine dei task.

Per avviare: python main.py; per i test, dalla cartella del progetto, pytest -v (il file pytest.ini incluso evita i classici problemi di import).

File prestiti.py

```
"""
prestiti.py — Prestare e restituire i libri, trovare i ritardi.

Le date arrivano sempre come parametro "oggi" e non vengono lette dal computer: così nei test possiamo scegliere noi che giorno è (piano, sez. 7).
"""
```

```
from datetime import date

from catalogo import trova_libro

# PROVVISORIO: in attesa di decisione del gruppo (tasks, T004 e T023).
# Sono scritti UNA volta sola: per cambiarli basta modificare queste righe.
MAX_LIBRI = 3          # libri che uno studente può avere contemporaneamente
GIORNI_PRESTITO = 30   # dopo quanti giorni un prestito è in ritardo


def prestito_in_corso(dati, codice):
    """
    Cerca il prestito non ancora restituito di una copia.

    Restituisce: il prestito, oppure None se la copia non è in prestito.
    """
    for prestito in dati["prestiti"]:
        if prestito["codice_libro"] == codice and prestito["data_restituzione"] is None:
            return prestito
    return None


def e_disponibile(dati, codice):
    """
    Dice se una copia si può prestare.

    Restituisce: True se disponibile, False se è in prestito
    Errore:      ValueError se il codice non esiste

    Lo stato non è salvato da nessuna parte: si calcola.
    Un libro è in prestito se ha un prestito senza data di restituzione.
    """
    libro = trova_libro(dati, codice)
    return prestito_in_corso(dati, libro["codice"]) is None


def libri_dello_studente(dati, studente):
    """Conta quanti libri lo studente ha in prestito in questo momento."""
    conta = 0
    for prestito in dati["prestiti"]:
        if prestito["studente"] == studente and prestito["data_restituzione"] is None:
            conta += 1
    return conta


def presta_libro(dati, codice, studente, oggi):
    """
    Registra il prestito di una copia a uno studente (R-01, R-03).

    Riceve:      i dati, il codice della copia, lo studente
                 (dizionario con nome, cognome, classe) e la data di oggi
    Restituisce: niente
    Errore:      ValueError se il libro non è disponibile (R-02)
                 o se lo studente ha già MAX_LIBRI libri
    """
    libro = trova_libro(dati, codice)

    if not e_disponibile(dati, libro["codice"]):
        raise ValueError(f"Il libro {libro['codice']} è già in prestito.")

    # Salviamo SOLO nome, cognome e classe (R-06, Constitution Art. 4),
    # anche se il dizionario ricevuto contenesse altro.
    # Uniformiamo le maiuscole: "giulia" e "Giulia" sono la stessa persona.
    studente = {
        "nome": studente["nome"].strip().title(),
        "cognome": studente["cognome"].strip().title(),
        "classe": studente["classe"].strip().upper(),
    }
```

```
    if libri_dello_studente(dati, studente) >= MAX_LIBRI:
        raise ValueError(f"Lo studente ha già {MAX_LIBRI} libri in prestito.")

    dati["prestiti"].append({
        "codice_libro": libro["codice"],
        "studente": studente,
        "data_inizio": oggi.isoformat(),   # data nel formato AAAA-MM-GG
        "data_restituzione": None,         # None diventa null nel file JSON
    })


def restituisci_libro(dati, codice, oggi):
    """
    Registra la restituzione di una copia (R-04).

    Riceve:      i dati, il codice della copia e la data di oggi
    Restituisce: niente
    Errore:      ValueError se il libro non è in prestito

    Il prestito NON viene cancellato: si scrive la data di restituzione,
    così resta lo storico (piano, sez. 6).
    """
    libro = trova_libro(dati, codice)
    prestito = prestito_in_corso(dati, libro["codice"])

    if prestito is None:
        raise ValueError(f"Il libro {libro['codice']} non risulta in prestito.")

    prestito["data_restituzione"] = oggi.isoformat()


def giorni_di_prestito(prestito, oggi):
    """Calcola da quanti giorni un prestito è iniziato."""
    inizio = date.fromisoformat(prestito["data_inizio"])
    return (oggi - inizio).days


def libri_in_ritardo(dati, oggi):
    """
    Trova i prestiti in corso da più di GIORNI_PRESTITO giorni (R-05).

    Riceve:      i dati e la data di oggi
    Restituisce: la lista dei prestiti in ritardo (può essere vuota)
    """
    in_ritardo = []
    for prestito in dati["prestiti"]:
        if prestito["data_restituzione"] is None:
            if giorni_di_prestito(prestito, oggi) > GIORNI_PRESTITO:
                in_ritardo.append(prestito)
    return in_ritardo
```

File catalogo.py

```
"""
"""
catalogo.py — Aggiungere e cercare i libri del catalogo.

Regola del piano (sez. 5): qui NON si usano input() e print().
Le funzioni ricevono dati e restituiscono risultati, così possono essere controllate in automatico
con pytest.
"""


def nuovo_codice(dati):
    """
```

```
    Calcola il codice da dare alla prossima copia inserita.

    Riceve:      i dati della biblioteca
    Restituisce: un codice nel formato "B-001", "B-002", ...

    Il nuovo codice è il più alto già usato + 1 (R-08).
    """
    numero_piu_alto = 0
    for libro in dati["libri"]:
        # "B-007" -> tolgo "B-" -> "007" -> 7
        numero = int(libro["codice"][2:])
        if numero > numero_piu_alto:
            numero_piu_alto = numero

    # :03d scrive il numero con 3 cifre, aggiungendo zeri davanti (7 -> 007)
    return f"B-{numero_piu_alto + 1:03d}"


def aggiungi_libro(dati, titolo, autore):
    """
    Aggiunge una nuova copia al catalogo (R-07).

    Riceve:      i dati, il titolo e l'autore del libro
    Restituisce: il codice assegnato alla copia
    Errore:      ValueError se il titolo è vuoto (R-10)

    Serve sia per il primo caricamento dei libri già posseduti
    sia per i libri nuovi: è la stessa operazione (spec, storia 1).
    """
    titolo = titolo.strip()      # tolgo gli spazi all'inizio e alla fine
    autore = autore.strip()

    if titolo == "":
        raise ValueError("Il titolo non può essere vuoto.")

    codice = nuovo_codice(dati)
    dati["libri"].append({"codice": codice, "titolo": titolo, "autore": autore})

    # Nota: NON salviamo lo stato "disponibile".
    # Lo stato si calcola dai prestiti (piano, sez. 6).
    return codice


def trova_libro(dati, codice):
    """
    Cerca una copia nel catalogo a partire dal suo codice.

    Riceve:      i dati e il codice (es. "B-001")
    Restituisce: il dizionario del libro trovato
    Errore:      ValueError se nessun libro ha quel codice
    """
    codice = codice.strip().upper()   # accetta anche "b-001"
    for libro in dati["libri"]:
        if libro["codice"] == codice:
            return libro
    raise ValueError(f"Nessun libro con codice {codice}.")


# correggi_libro(dati, codice, titolo, autore)
# Task T024-T025 in [ATTESA]: si scrive solo se il gruppo decide che serve.
```

File archivio.py

```
"""
archivio.py — Leggere e salvare i dati di BiblioClasse.

Questo è l'unico file che sa come sono salvati i dati (un file JSON).
```

```
Tutti gli altri file lavorano con un normale dizionario Python:
    {"libri": [...], "prestiti": [...]}
"""

import json


def carica(percorso):
    """
    Legge i dati dal file JSON indicato.

    Riceve:      il percorso del file (es. "biblioteca.json")
    Restituisce: un dizionario con le chiavi "libri" e "prestiti".
                 Se il file non esiste ancora (primo avvio),
                 restituisce un archivio vuoto.
    """
    try:
        with open(percorso, "r", encoding="utf-8") as file:
            return json.load(file)
    except FileNotFoundError:
        # Primo avvio: nessun file, quindi si parte da zero.
        return {"libri": [], "prestiti": []}


def salva(dati, percorso):
    """
    Scrive i dati nel file JSON indicato.

    Riceve:      i dati (dizionario) e il percorso del file
    Restituisce: niente
    """
    with open(percorso, "w", encoding="utf-8") as file:
        # indent=2 rende il file leggibile anche a occhio;
        # ensure_ascii=False permette di salvare le lettere accentate.
        json.dump(dati, file, indent=2, ensure_ascii=False)
```

File main.py

```
"""
main.py — Il menu di BiblioClasse.

È l'UNICO file che usa input() e print() (piano, sez. 5):
chiede i dati all'utente, chiama le funzioni degli altri file
e mostra i risultati. Non contiene logica sui libri o sui prestiti.

Per avviare il programma:  python main.py
"""

from datetime import date

from archivio import carica, salva
from catalogo import aggiungi_libro, trova_libro
from prestiti import GIORNI_PRESTITO, giorni_di_prestito, libri_in_ritardo, presta_libro,
restituisci_libro

PERCORSO = "biblioteca.json"


def chiedi_studente():
    """Chiede all'utente nome, cognome e classe dello studente."""
    return {
        "nome": input("Nome dello studente: "),
        "cognome": input("Cognome dello studente: "),
        "classe": input("Classe (es. 3C): "),
    }
```

```
def menu_aggiungi(dati):
    """Chiede titolo e autore e aggiunge il libro al catalogo."""
    titolo = input("Titolo: ")
    autore = input("Autore: ")
    codice = aggiungi_libro(dati, titolo, autore)
    print(f"Libro aggiunto con codice {codice}. Scrivilo sull'etichetta!")


def menu_presta(dati):
    """Chiede codice e studente e registra il prestito con la data di oggi."""
    codice = input("Codice del libro (es. B-001): ")
    studente = chiedi_studente()
    presta_libro(dati, codice, studente, date.today())
    print("Prestito registrato.")


def menu_restituisci(dati):
    """Chiede il codice e registra la restituzione con la data di oggi."""
    codice = input("Codice del libro restituito: ")
    restituisci_libro(dati, codice, date.today())
    print("Restituzione registrata. Il libro è di nuovo disponibile.")


def menu_ritardi(dati):
    """Mostra i libri in ritardo, con chi li ha e da quanti giorni."""
    oggi = date.today()
    ritardi = libri_in_ritardo(dati, oggi)
    if len(ritardi) == 0:
        print("Nessun libro in ritardo.")
        return
    for prestito in ritardi:
        libro = trova_libro(dati, prestito["codice_libro"])
        s = prestito["studente"]
        giorni = giorni_di_prestito(prestito, oggi) - GIORNI_PRESTITO
        print(f"- {libro['codice']} «{libro['titolo']}»: "
              f"{s['nome']} {s['cognome']} ({s['classe']}), {giorni} giorni di ritardo")


def main():
    """Carica i dati, mostra il menu e salva dopo ogni operazione."""
    dati = carica(PERCORSO)
    azioni = {
        "1": menu_aggiungi,
        "2": menu_presta,
        "3": menu_restituisci,
        "4": menu_ritardi,
    }

    while True:
        print("\n=== BiblioClasse ===")
        print("1) Aggiungi libro")
        print("2) Presta libro")
        print("3) Restituisci libro")
        print("4) Libri in ritardo")
        print("0) Esci")
        scelta = input("Scelta: ").strip()

        if scelta == "0":
            print("Arrivederci!")
            break
        if scelta not in azioni:
            print("Scelta non valida.")
            continue

        # Se una funzione segnala un errore, lo mostriamo con un messaggio
        # chiaro e il programma continua (task T020).
        try:
```

```
            azioni[scelta](dati)
            salva(dati, PERCORSO)   # si salva dopo ogni operazione
        except ValueError as errore:
            print(f"Attenzione: {errore}")


# Questa riga fa partire il menu solo quando si esegue "python main.py",
# e non quando un altro file (per esempio un test) importa main.py.
if __name__ == "__main__":
    main()
```

File test prestiti.py

```
"""
Test di prestiti.py (tasks T011-T014).

I test usano le costanti MAX_LIBRI e GIORNI_PRESTITO invece di scrivere 3 e 30: quando il gruppo
cambierà i valori, i test resteranno validi.
"""

from datetime import date, timedelta

import pytest

from catalogo import aggiungi_libro
from prestiti import (GIORNI_PRESTITO, MAX_LIBRI, e_disponibile,
                      libri_in_ritardo, presta_libro, restituisci_libro)

OGGI = date(2026, 10, 15)   # scegliamo noi che giorno è "oggi"
GIULIA = {"nome": "Giulia", "cognome": "Rossi", "classe": "3C"}   # inventata
MARCO = {"nome": "Marco", "cognome": "Bianchi", "classe": "3C"}   # inventato


def dati_con_un_libro():
    """Una biblioteca con una sola copia, con codice B-001."""
    dati = {"libri": [], "prestiti": []}
    aggiungi_libro(dati, "Il nome della rosa", "Umberto Eco")
    return dati


def test_libro_nuovo_e_disponibile():                                  # R-09
    assert e_disponibile(dati_con_un_libro(), "B-001")


def test_presta_libro_disponibile():                                   # R-01
    dati = dati_con_un_libro()
    presta_libro(dati, "B-001", GIULIA, OGGI)
    assert not e_disponibile(dati, "B-001")


def test_prestito_registra_data():                                     # R-03
    dati = dati_con_un_libro()
    presta_libro(dati, "B-001", GIULIA, OGGI)
    assert dati["prestiti"][0]["data_inizio"] == "2026-10-15"


def test_non_presta_libro_gia_in_prestito():                           # R-02
    dati = dati_con_un_libro()
    presta_libro(dati, "B-001", GIULIA, OGGI)
    with pytest.raises(ValueError):
        presta_libro(dati, "B-001", MARCO, OGGI)


def test_limite_libri_per_studente():
    dati = {"libri": [], "prestiti": []}
    for i in range(MAX_LIBRI + 1):              # un libro in più del limite
        aggiungi_libro(dati, f"Libro {i}", "Autore")
```

```
    for i in range(MAX_LIBRI):                  # arrivo esattamente al limite
        presta_libro(dati, f"B-{i + 1:03d}", GIULIA, OGGI)
    with pytest.raises(ValueError):             # il successivo va rifiutato
        presta_libro(dati, f"B-{MAX_LIBRI + 1:03d}", GIULIA, OGGI)


def test_studente_ha_solo_nome_cognome_classe():                    # R-06
    dati = dati_con_un_libro()
    studente_con_troppi_dati = dict(GIULIA, telefono="3331234567")
    presta_libro(dati, "B-001", studente_con_troppi_dati, OGGI)
    salvato = dati["prestiti"][0]["studente"]
    assert set(salvato.keys()) == {"nome", "cognome", "classe"}


def test_restituzione_rende_disponibile():                          # R-04
    dati = dati_con_un_libro()
    presta_libro(dati, "B-001", GIULIA, OGGI)
    restituisci_libro(dati, "B-001", OGGI)
    assert e_disponibile(dati, "B-001")


def test_non_restituisce_libro_non_in_prestito():
    with pytest.raises(ValueError):
        restituisci_libro(dati_con_un_libro(), "B-001", OGGI)


def test_ritardo_dopo_i_giorni_previsti():                          # R-05
    dati = dati_con_un_libro()
    inizio = OGGI - timedelta(days=GIORNI_PRESTITO + 5)
    presta_libro(dati, "B-001", GIULIA, inizio)
    assert len(libri_in_ritardo(dati, OGGI)) == 1


def test_prestito_recente_non_in_ritardo():                         # R-05
    dati = dati_con_un_libro()
    presta_libro(dati, "B-001", GIULIA, OGGI - timedelta(days=1))
    assert libri_in_ritardo(dati, OGGI) == []


def test_esattamente_al_limite_non_e_ritardo():                     # R-05
    # La spec dice "PIÙ di 30 giorni": il 30° giorno non è ancora ritardo.
    dati = dati_con_un_libro()
    presta_libro(dati, "B-001", GIULIA, OGGI - timedelta(days=GIORNI_PRESTITO))
    assert libri_in_ritardo(dati, OGGI) == []
```

File test catalogo.py

```
"""Test di catalogo.py (tasks T007-T008)."""

import pytest

from catalogo import aggiungi_libro, nuovo_codice, trova_libro


def dati_vuoti():
    """Una biblioteca senza libri e senza prestiti, per partire puliti."""
    return {"libri": [], "prestiti": []}


def test_primo_codice_e_B001():                                     # R-08
    assert nuovo_codice(dati_vuoti()) == "B-001"


def test_codice_successivo():                                       # R-08
    dati = dati_vuoti()
    aggiungi_libro(dati, "Il barone rampante", "Italo Calvino")
```

```
    assert nuovo_codice(dati) == "B-002"


def test_aggiungi_libro():                                          # R-07
    dati = dati_vuoti()
    codice = aggiungi_libro(dati, "Il nome della rosa", "Umberto Eco")
    libro = trova_libro(dati, codice)
    assert libro["titolo"] == "Il nome della rosa"
    assert libro["autore"] == "Umberto Eco"


def test_non_accetta_titolo_vuoto():                                # R-10
    with pytest.raises(ValueError):
        aggiungi_libro(dati_vuoti(), "", "Umberto Eco")


def test_non_accetta_titolo_di_soli_spazi():                        # R-10
    with pytest.raises(ValueError):
        aggiungi_libro(dati_vuoti(), "    ", "Umberto Eco")


def test_due_copie_hanno_codici_diversi():                          # R-08
    dati = dati_vuoti()
    primo = aggiungi_libro(dati, "Il nome della rosa", "Umberto Eco")
    secondo = aggiungi_libro(dati, "Il nome della rosa", "Umberto Eco")
    assert primo != secondo


def test_trova_libro_inesistente():
    with pytest.raises(ValueError):
        trova_libro(dati_vuoti(), "B-999")
```

File test archivio.py

```
"""Test di archivio.py (tasks T005). Dati di prova inventati (Art. 4)."""

from pathlib import Path

from archivio import carica, salva

CARTELLA_TEST = Path(__file__).parent


def test_file_inesistente_da_archivio_vuoto(tmp_path):
    # tmp_path è una cartella temporanea che pytest crea e poi cancella
    dati = carica(tmp_path / "non_esiste.json")
    assert dati == {"libri": [], "prestiti": []}


def test_salvo_e_ricarico_gli_stessi_dati(tmp_path):
    percorso = tmp_path / "prova.json"
    dati = {"libri": [{"codice": "B-001", "titolo": "Città", "autore": "Anonimo"}],
            "prestiti": []}
    salva(dati, percorso)
    assert carica(percorso) == dati


def test_carica_dati_di_prova():
    dati = carica(CARTELLA_TEST / "dati_prova.json")
    assert len(dati["libri"]) == 5
    assert len(dati["prestiti"]) == 1
```

File readme.md

```
# BiblioClasse

Programma per gestire i prestiti dei libri della biblioteca di classe.
Progetto della 3C, gruppo 2.

## Documenti del progetto

1. `constitution.md` — le regole fondamentali
2. `spec.md` — cosa fa il programma (spec 001)
3. `plan.md` — come è costruito
4. `tasks.md` — la lista dei lavori

## Requisiti

- Python 3.12
- pytest (solo per i test): `pip install pytest`

## Come si avvia

    python main.py

Al primo avvio viene creato il file `biblioteca.json` con i dati.

## Come si lanciano i test

Dalla cartella principale del progetto:

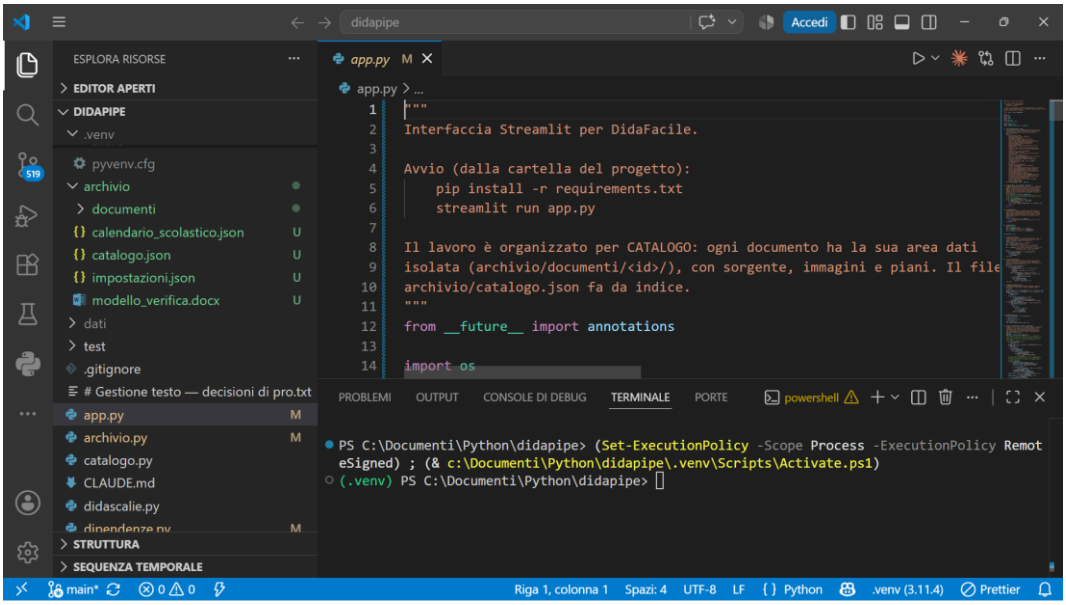
    pytest -v

## Valori provvisori

In `prestiti.py` si trovano `MAX_LIBRI = 3` e `GIORNI_PRESTITO = 30`.
Sono in attesa della decisione del gruppo (tasks, T022–T023).
```

5 Appendice — Scheda Visual Studio Code

Visual Studio Code è un editor di testo pensato apposta per scrivere codice: a differenza di un elaboratore di testi, sa riconoscere la sintassi di Python (colora parole chiave, segnala errori mentre scrivi) e integra gli strumenti che servono per eseguire e correggere un programma, senza dover aprire altre finestre.



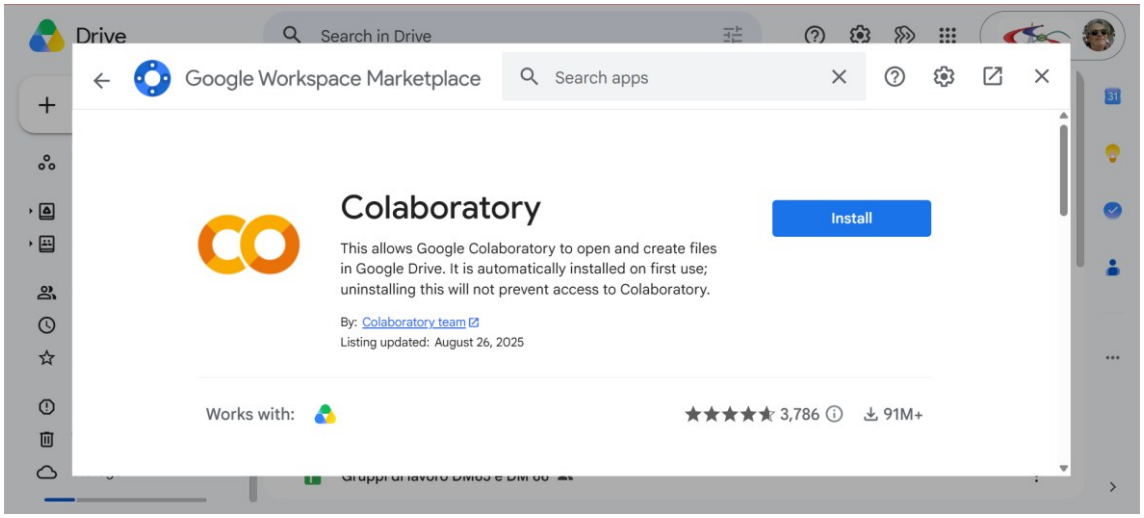
Primo utilizzo guidato: aprite la cartella del vostro progetto (File → Apri cartella), create un nuovo file chiamato prova.py e scrivete print("Ciao mondo"). Premete il pulsante ▷ in alto a destra: si apre il terminale integrato in basso e compare l'output. Ora mettete un punto di interruzione (clic a sinistra del numero di riga) sulla riga del print e premete F5 invece di ▷: l'esecuzione si ferma su quella riga e a sinistra compare il pannello delle variabili — è lì che, più avanti, guarderete il contenuto delle variabili mentre un programma gira, invece di indovinarlo.

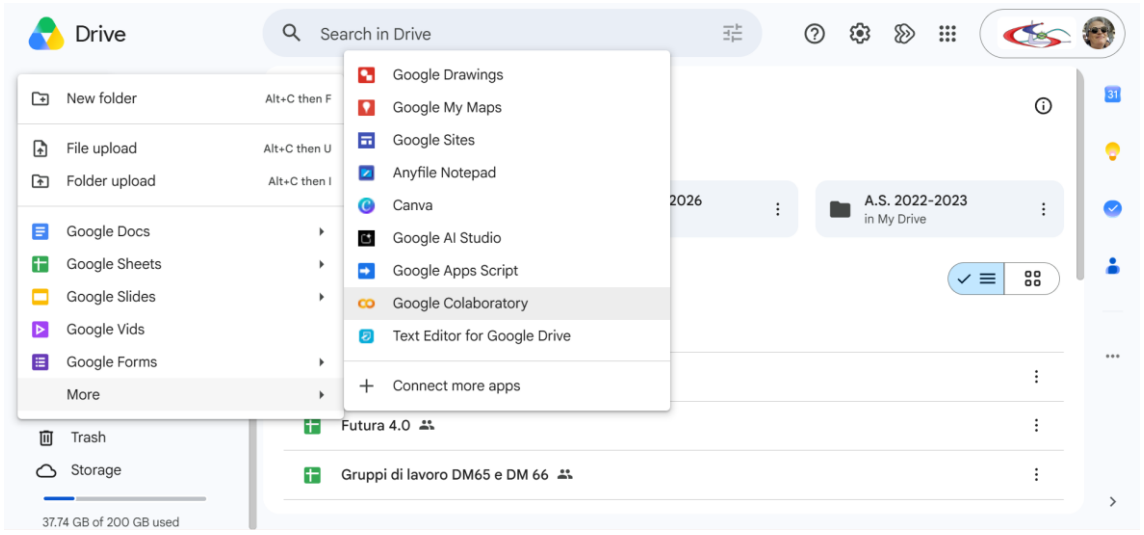
Azione	Scorciatoia / comando
Eseguire il file aperto	Pulsante ▷ in alto a destra, oppure Ctrl+F5
Eseguire riga per riga (debug)	F5 (si ferma sugli eventuali breakpoint)
Impostare un breakpoint	Clic a sinistra del numero di riga
Aprire il terminale integrato	Ctrl+`
Salvare il file	Ctrl+S
Commentare/decommentare la riga	Ctrl+/
Formattare automaticamente il codice	Shift+Alt+F
Autocompletamento	Appare mentre si scrive; Tab per accettare
Cercare un file nel progetto	Ctrl+P

Nota: il terminale integrato (Ctrl+`) è lo stesso terminale del sistema operativo, comodo perché resta nella stessa finestra dell'editor — è lì che vale la scheda del Terminale più avanti in questa appendice.

6 Appendice — Scheda Google Colab

Google Colab è un servizio online che fa girare Python direttamente nel browser: non serve installare nulla sul proprio computer, basta un account Google. Il documento su cui si lavora si chiama notebook ed è diviso in celle: ciascuna cella contiene o codice (che si può eseguire singolarmente) o testo descrittivo, e i risultati compaiono subito sotto la cella che li ha generati.





Primo utilizzo guidato: da Google Drive, Nuovo → Altro → Google Colaboratory. Nella prima cella scrivete print("Ciao mondo") e premete Shift+Invio: il risultato appare subito sotto e il cursore passa a una nuova cella. Con + Testo aggiungete una cella descrittiva sopra, per esempio "Il mio primo notebook" — utile per documentare cosa fa ogni pezzo di codice man mano che il notebook cresce.

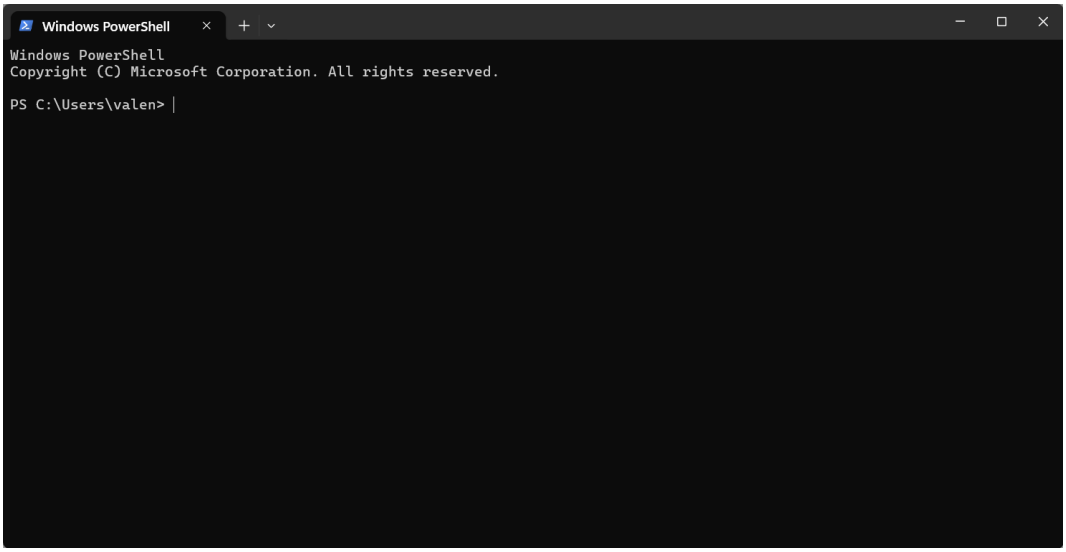
Azione	Come si fa
Creare un nuovo notebook	Da Google Drive: Nuovo → Altro → Google Colaboratory
Eseguire una cella	▷ a sinistra della cella, oppure Shift+Invio
Aggiungere una cella di codice	+ Codice in alto, o tra due celle esistenti
Aggiungere una cella di testo	+ Testo (supporta Markdown)
Installare una libreria non inclusa	!pip install nome_libreria in una cella
Caricare un file dal proprio PC	Icona cartella a sinistra → icona di upload

Collegare il proprio Google Drive	Icona cartella a sinistra → icona Drive
Salvare il lavoro	Automatico su Google Drive

Quando usarlo: nessuna installazione, comodo per esperimenti rapidi e per condividere codice con un link; per progetti più strutturati con più file resta preferibile VS Code.

7 Appendice — Scheda Terminale

Il terminale (o prompt dei comandi) permette di dare istruzioni al computer scrivendole invece di usare mouse e finestre. Prima delle interfacce grafiche, era l'unico modo per farlo, ed è ancora oggi il modo più diretto: userete il terminale per lanciare script Python da riga di comando o per usare Git. I comandi cambiano leggermente tra Windows e Mac/Linux.



Primo utilizzo guidato: aprite il terminale (cercatelo come "Terminale" o "Prompt dei comandi" nel menù di sistema, oppure usate Ctrl+` da dentro VS Code). Scrivete cd Desktop per spostarvi sul desktop, poi mkdir prova per creare una cartella, poi cd prova per entrarci. Da qui, se avete già un file prova.py, python3 prova.py (o python prova.py su Windows) lo esegue: è esattamente quello che succede dietro le quinte quando premete ▷ in VS Code.

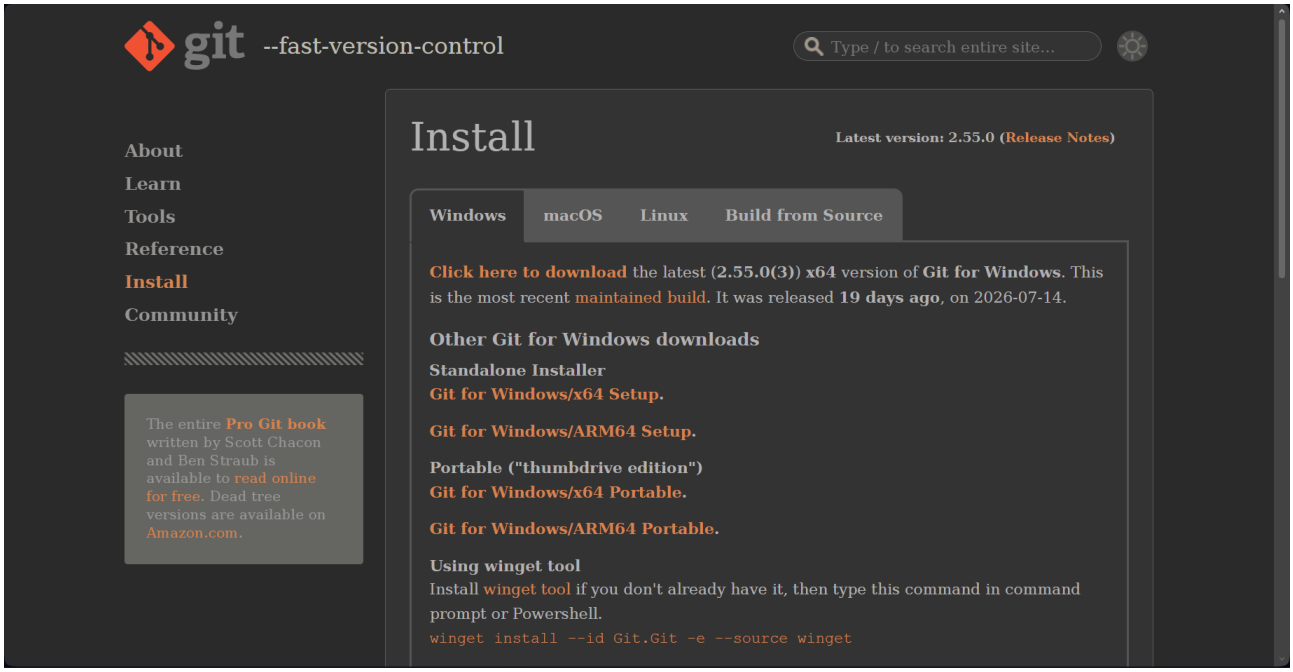
Azione	Windows (PowerShell/CMD)	Mac / Linux
Vedere in che cartella ci si trova	cd (senza argomenti)	pwd
Vedere il contenuto della cartella	dir	ls
Spostarsi in una cartella	cd nome_cartella	cd nome_cartella
Tornare alla cartella superiore	cd ..	cd ..
Creare una nuova cartella	mkdir nome_cartella	mkdir nome_cartella
Eseguire uno script Python	python nome_file.py	python3 nome_file.py

Attenzione: i comandi per cancellare file o cartelle (del, rm, rmdir) non sono inclusi qui — sono irreversibili (niente cestino) e non servono per gli usi di questa dispensa; se in futuro vi servono, chiedete prima di usarli.

Nota su Python: su Windows il comando è spesso python, su Mac/Linux quasi sempre python3 (perché python da solo può non esistere o puntare a Python 2). Se un comando dà errore "comando non trovato", è la prima cosa da controllare.

8 Appendice — Scheda Git: comandi essenziali

Git tiene traccia delle modifiche a un progetto nel tempo attraverso degli “scatti” chiamati commit. Il progetto vive in un repository, che può essere solo sul proprio computer (locale) o condiviso anche su una piattaforma come GitHub o GitLab (remoto).



Come si installa: su Windows si scarica l'installer da git-scm.com e si procede con le impostazioni predefinite; su macOS basta digitare git in un Terminale — se non è già presente, il sistema propone di installare gli Strumenti da riga di comando di Xcode, che lo includono — oppure si può usare Homebrew (brew install git); su Linux si installa dal gestore di pacchetti della propria distribuzione (per esempio sudo apt install git su Ubuntu/Debian). Per verificare che l'installazione sia riuscita, aprite un terminale e digitate git --version: se compare un numero di versione, Git è pronto all'uso.

Primo utilizzo guidato: da terminale, dentro la cartella del vostro progetto, git init lo trasforma in un repository. Scrivete o modificate un file, poi git status: vedrete il file elencato come non ancora tracciato. git add nome_file.py lo prepara, git commit -m "Primo commit" lo salva nella cronologia. Da questo momento, git log mostra quel commit, e ogni futura modifica seguirà lo stesso schema: modifica → add → commit. Solo quando il progetto sarà condiviso su GitHub servirà anche push.

Comando	Cosa fa
git init	Crea un nuovo repository nella cartella corrente
git clone <url>	Scarica in locale una copia di un repository remoto esistente
git status	Mostra quali file sono stati modificati, aggiunti o non tracciati
git add <file> (o git add .)	Prepara i file per il prossimo commit
git commit -m "messaggio"	Salva le modifiche preparate come nuovo commit
git log	Mostra l'elenco dei commit effettuati
git diff	Mostra le differenze non ancora salvate rispetto all'ultimo commit

git push	Invia i commit locali al repository remoto
git pull	Scarica e integra le modifiche presenti sul remoto
git remote -v	Mostra a quale repository remoto è collegato il progetto
git branch	Elenca i rami esistenti, o ne crea uno nuovo (git branch nome)
git switch <ramo>	Passa a un altro ramo
git merge <ramo>	Unisce le modifiche di un ramo in quello corrente

9 Sitografia

- https://it.wikipedia.org/wiki/Linguaggio_di_programmazione
- https://it.wikipedia.org/wiki/Elenco_cronologico_dei_linguaggi_di_programmazione
- http://www.linuxdidattica.org/docs/altre_scuole/planck/arch-sis/arch-sis4.html
- https://it.wikipedia.org/wiki/Linguaggio_di_programmazione_ad_alto_livello
- https://it.wikipedia.org/wiki/Programmazione_(informatica)
- https://it.wikipedia.org/wiki/Metodologia_agile
- https://it.wikipedia.org/wiki/Compilatore
- https://it.wikipedia.org/wiki/Interprete_(informatica)
- https://it.wikipedia.org/wiki/Teorema_di_Böhm-Jacopini
- http://it.wikipedia.org/wiki/Analisi_lessicale
- http://it.wikipedia.org/wiki/Grammatica_formale
- http://it.wikipedia.org/wiki/Linguaggio_regolare
- http://it.wikipedia.org/wiki/Grammatica_regolare
- http://it.wikipedia.org/wiki/Grammatica_lineare
- http://it.wikipedia.org/wiki/Espressione_regolare
- http://it.wikipedia.org/wiki/Gerarchia_di_Chomsky
- http://homes.di.unimi.it/~palano/lfa-cap2.pdf
- http://cabibbo.dia.uniroma3.it/it/pdf/it_02_grammatiche_formali.pdf
- http://www.dipmat.unict.it/~barba/FONDAMENTI/ESERCIZI/LINGUAGGI/esercizi.html