



CONVEGNO ITALIANO
SULLA DIDATTICA DELL'INFORMATICA

Dalle foglie alle radici: imparare il *Debugging* dalle sue componenti fondamentali

Gabriele Pozzan
Tullio Vardanega

4 ottobre 2025, Salerno



UNIVERSITÀ
DEGLI STUDI
DI PADOVA



DIPARTIMENTO
MATEMATICA
DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

Fare **debugging** significa: diagnosticare e correggere **errori** di programmazione (**bug**), spesso ciclicamente

Fare debugging è...

... **inevitabile**

- È improbabile costruire software senza errori al primo tentativo
- Lo stile di pensiero associato al debugging aiuta anche a raffinare progressivamente i prodotti



... **difficile**

- Viene spesso citato come una causa di frustrazione sia per chi insegna che per chi impara

... **complesso**

- È un processo che richiede più passi
- È una abilità complessa che richiede il coordinamento di altre abilità

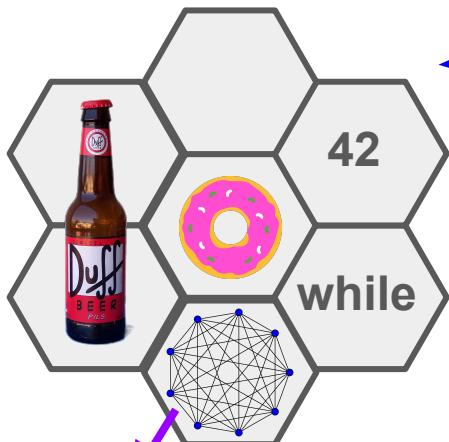


Complessità e apprendimento (1)

Teoria del **carico cognitivo** (John Sweller, Jeroen J. G. van Merriënboer)

Memoria di lavoro

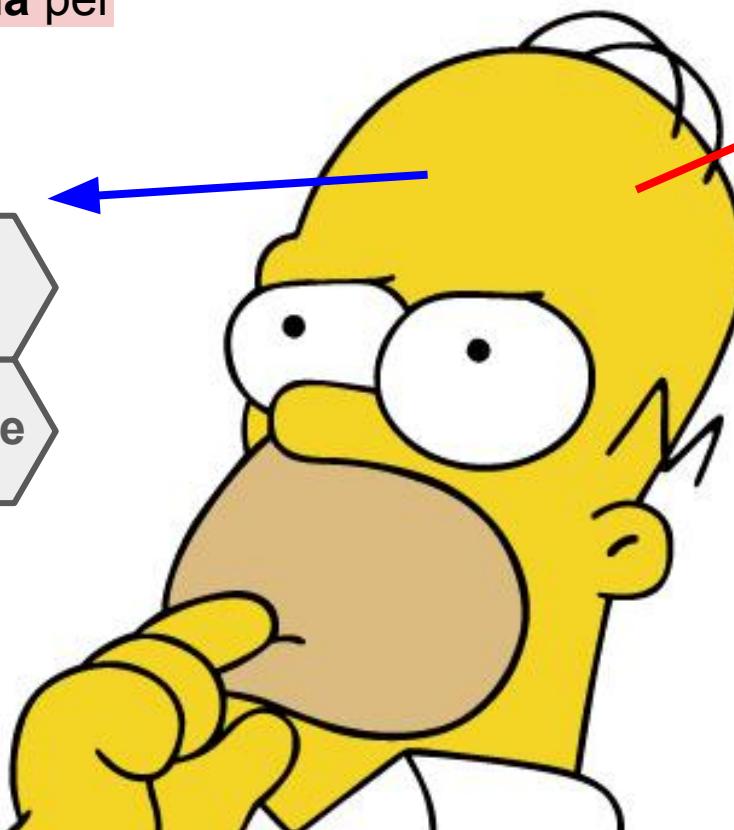
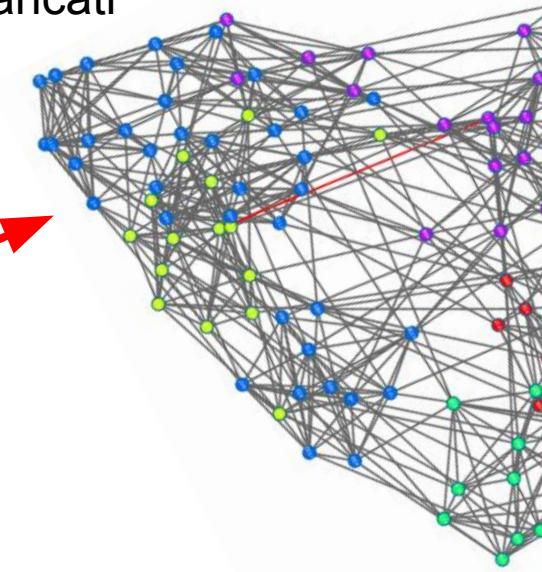
- Limitata
 - **Spazio**: pochi elementi contemporaneamente
 - **Tempo**: mantiene gli elementi per poco tempo
- **Collo di bottiglia** per l'apprendimento



Gli **schemi** hanno basso **carico cognitivo**

Memoria a lungo termine

- Sostanzialmente priva di limiti
- Funziona per **schemi**
- Gli schemi vengono “caricati” in memoria di lavoro



Obiettivo: evitare il sovraccarico della memoria di lavoro

Complessità e apprendimento (2)



Due tipologie di **processi mentali** sono associate al pensiero che “risolve problemi”



Sistema 1 (abilità ricorrenti)

- Veloce
- Inconscio
- Automatico
- Richiede esercitazioni **frequenti** per **lunghi periodi** di tempo
- Istruzioni **chiare** e applicabili facilmente
- **Riscontro correttivo immediato** sugli esercizi
- Esempio: esercizi di tipo “*drill*”
- Inflessibile
- Intuitivo

Sistema 2 (abilità non ricorrenti)

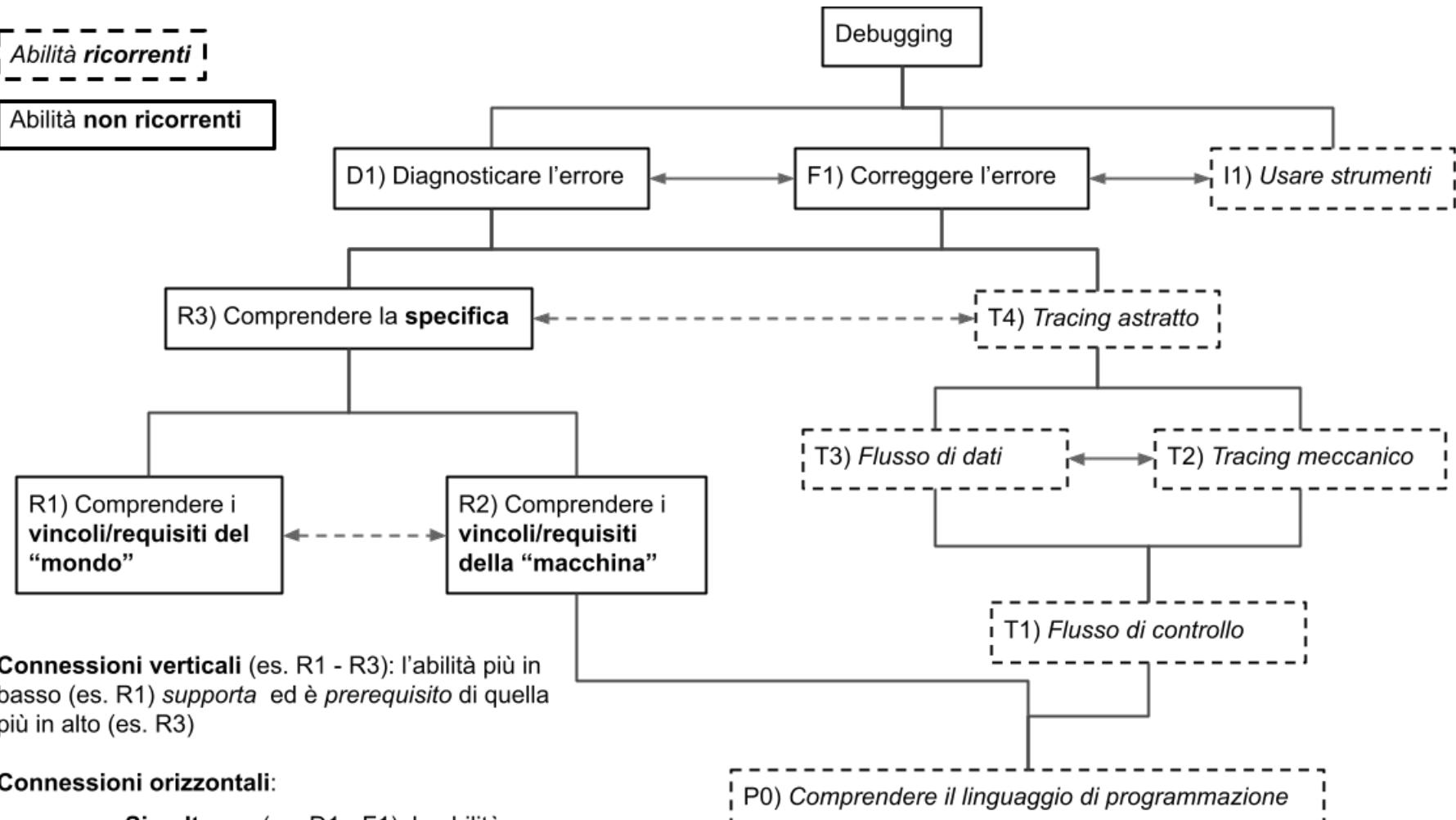
- Lento
- Conscio
- Flessibile
- “Costoso”

- Richiede **varietà** nelle esercitazioni, con l’obiettivo di **generalizzare** le abilità
- Le esercitazioni sono supportate da **informazioni di contesto**
- Enfasi sulla **riflessione** prima, durante e dopo gli esercizi (**metacognizione**)
- Esempio: esercizi di “*problem solving*”

Il debugging, scomposto

Abilità ricorrenti

Abilità non ricorrenti



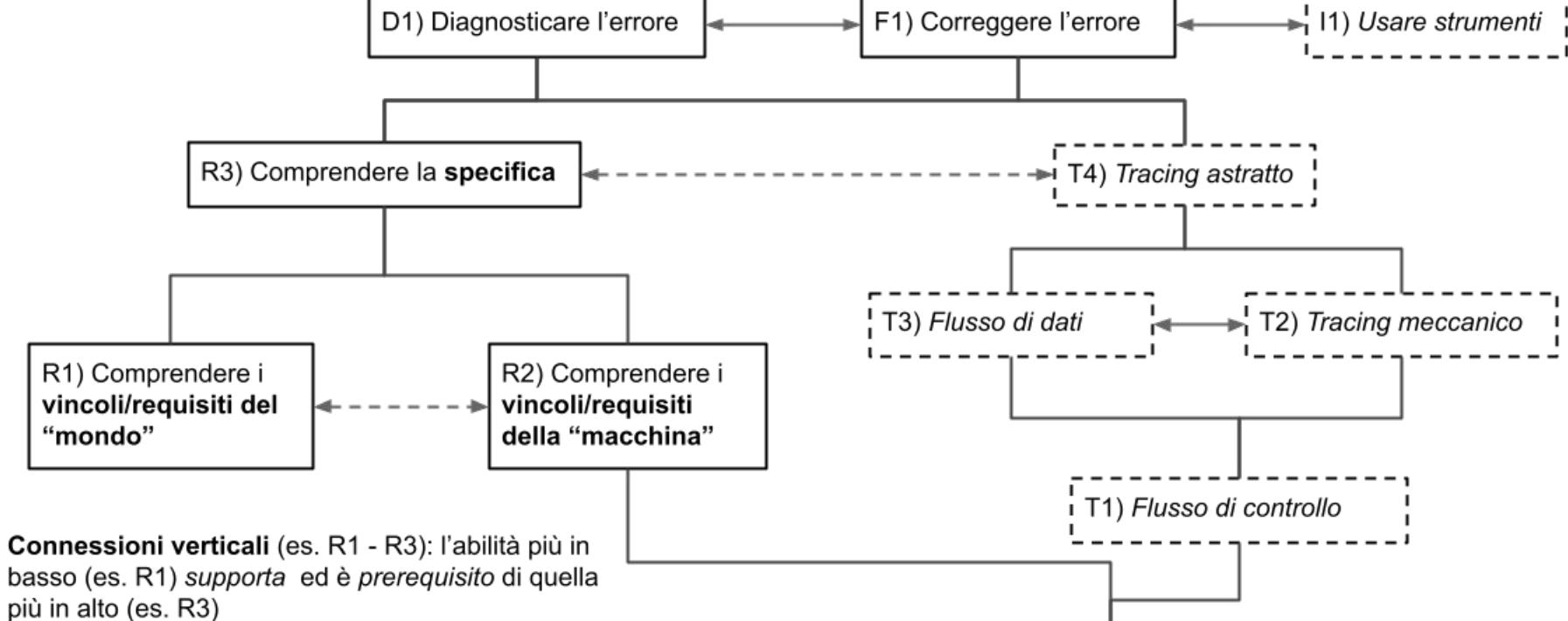
Connessioni verticali (es. R1 - R3): l'abilità più in basso (es. R1) *supporta* ed è *prerequisito* di quella più in alto (es. R3)

Connessioni orizzontali:

↔ **Simultanee** (es. D1 - F1): le abilità sono applicate simultaneamente (in comunicazione reciproca)
 ↔ **Trasponibili** (es. R1 - R2): le abilità possono essere applicate in qualsiasi ordine, anche simultaneamente

Abilità ricorrenti !

Abilità non ricorrenti

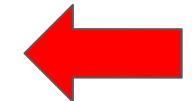


Connessioni verticali (es. R1 - R3): l'abilità più in basso (es. R1) *supporta* ed è *prerequisito* di quella più in alto (es. R3)

Connessioni orizzontali:

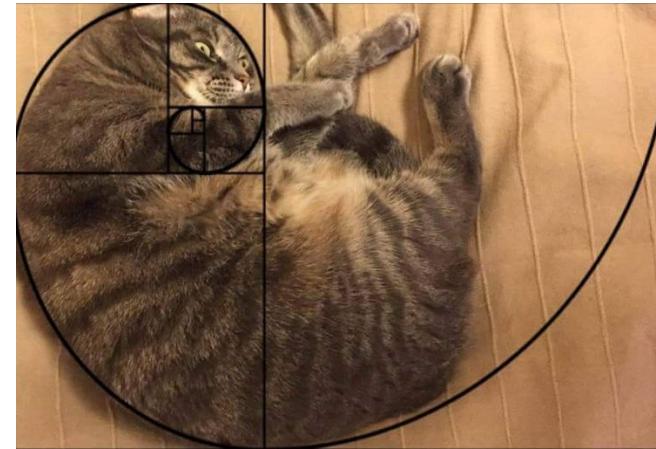
↔ **Simultanee** (es. D1 - F1): le abilità sono applicate simultaneamente (in comunicazione reciproca)

↔ **Trasponibili** (es. R1 - R2): le abilità possono essere applicate in qualsiasi ordine, anche simultaneamente



P0) Comprendere il linguaggio di programmazione

Nota: usare sempre un approccio a **spirale**



Comprendere include:

- **Sintassi:** regole di scrittura
- **Semantica:** significato dei costrutti
- **Pragmatica:** effetti dell'esecuzione

Esercizio 1

Alcune di queste istruzioni (Python) causano errori quando eseguite, quali?

1. `print("ciao, mondo')`
2. `print("ciao, mondo")`
3. `print(ciao, mondo)`
4. `print('ciao, mondo')`
5. `print("ciao, mondo");`
6. `prnt("ciao, mondo")`
7. `print("cioa, mndo")`
8. `print("ciao", "mondo")`
9. `print("ciao" "mondo")`
10. `print("ciao" + "mondo")`

Esercizio 1

Alcune di queste istruzioni (Python) causano errori quando eseguite, quali?

1. `print("ciao, mondo')`
2. `print("ciao, mondo")`
3. `print(ciao, mondo)`
4. `print('ciao, mondo')`
5. `print("ciao, mondo");`
6. `prnt("ciao, mondo")`
7. `print("cioa, mndo")`
8. `print("ciao", "mondo")`
9. `print("ciao" "mondo")`
10. `print("ciao" + "mondo")`

Gli apici a inizio e fine stringa
devono essere dello stesso tipo

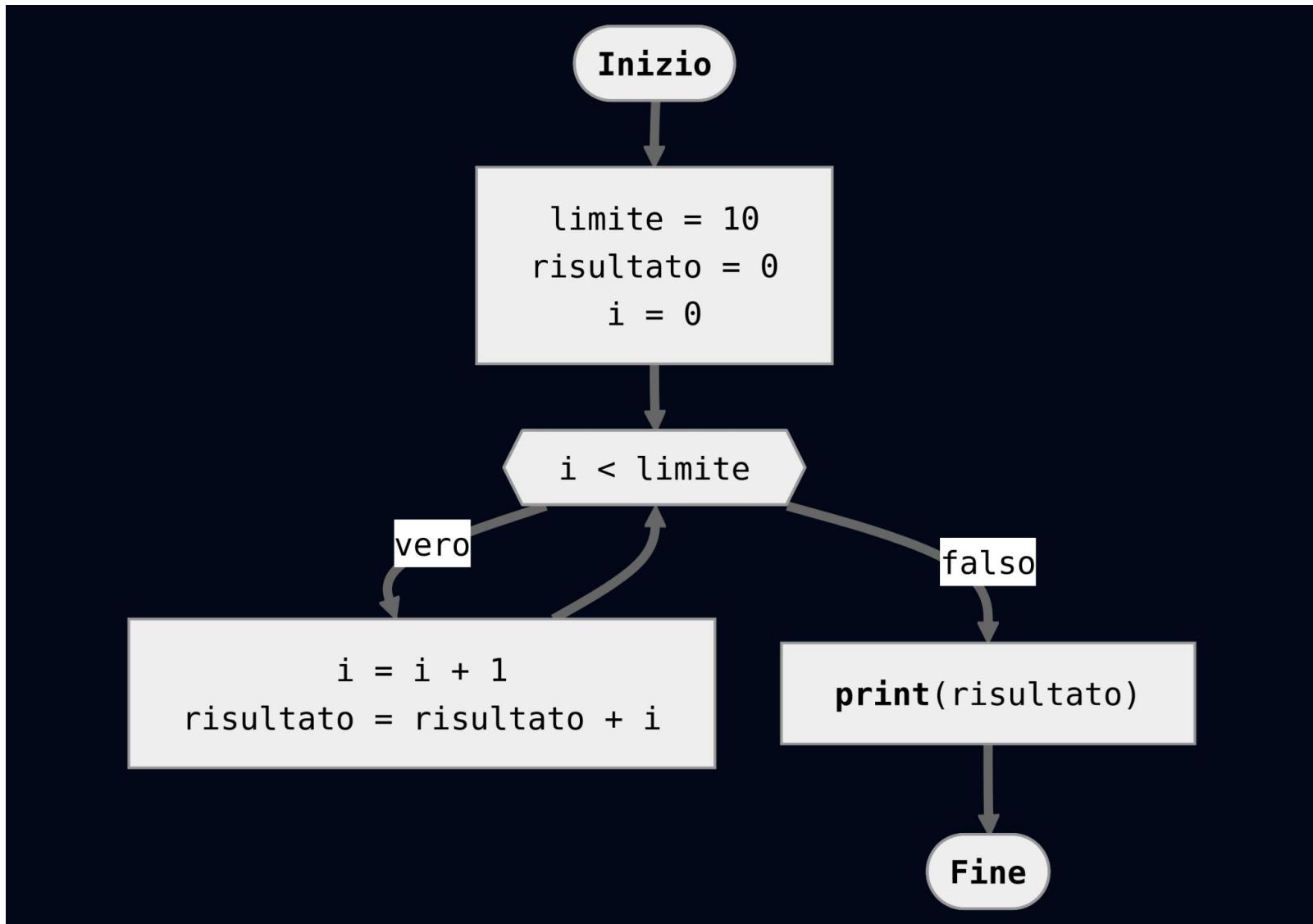
Dipende (!)

L'interprete Python non ha un
“correttore automatico”

Questo l'ho scoperto
provandolo per questo
esercizio (!!!)

Esercizio 2

Traduci questo diagramma in un programma eseguibile



Esercizio 3

Date le variabili:

a = 13

b = 12

Qual'è il valore di a dopo:

1. a = b
2. b = a
3. a = a + 1
4. a = 42
5. b = a - 1
6. a = "42"
7. a + b

Abilità ricorrenti

Abilità non ricorrenti



R1) Comprendere i vincoli/requisiti del "mondo"

R2) Comprendere i vincoli/requisiti della "macchina"

R3) Comprendere la specifica

D1) Diagnosticare l'errore

Debugging

F1) Correggere l'errore

I1) Usare strumenti

T4) Tracing astratto

T3) Flusso di dati

T2) Tracing meccanico

T1) Flusso di controllo

P0) Comprendere il linguaggio di programmazione

Connessioni verticali (es. R1 - R3): l'abilità più in basso (es. R1) *supporta* ed è *prerequisito* di quella più in alto (es. R3)

Connessioni orizzontali:

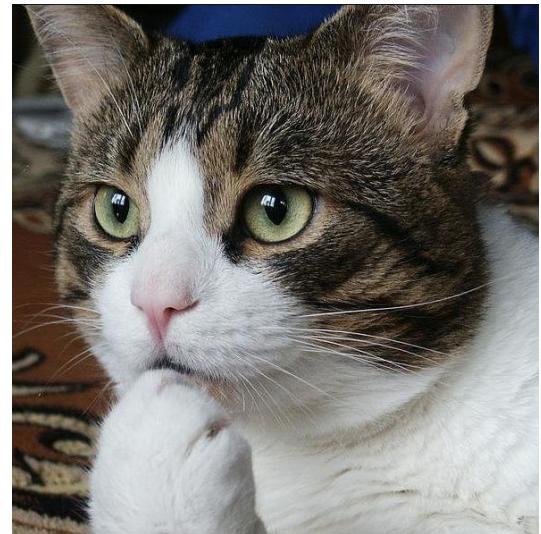
↔ **Simultanee** (es. D1 - F1): le abilità sono applicate simultaneamente (in comunicazione reciproca)
 ↔ → **Trasponibili** (es. R1 - R2): le abilità possono essere applicate in qualsiasi ordine, anche simultaneamente

R1) Comprendere i vincoli/requisiti del “mondo”

In un certo senso rispondere alla domanda:

“Perché esiste questo programma?”

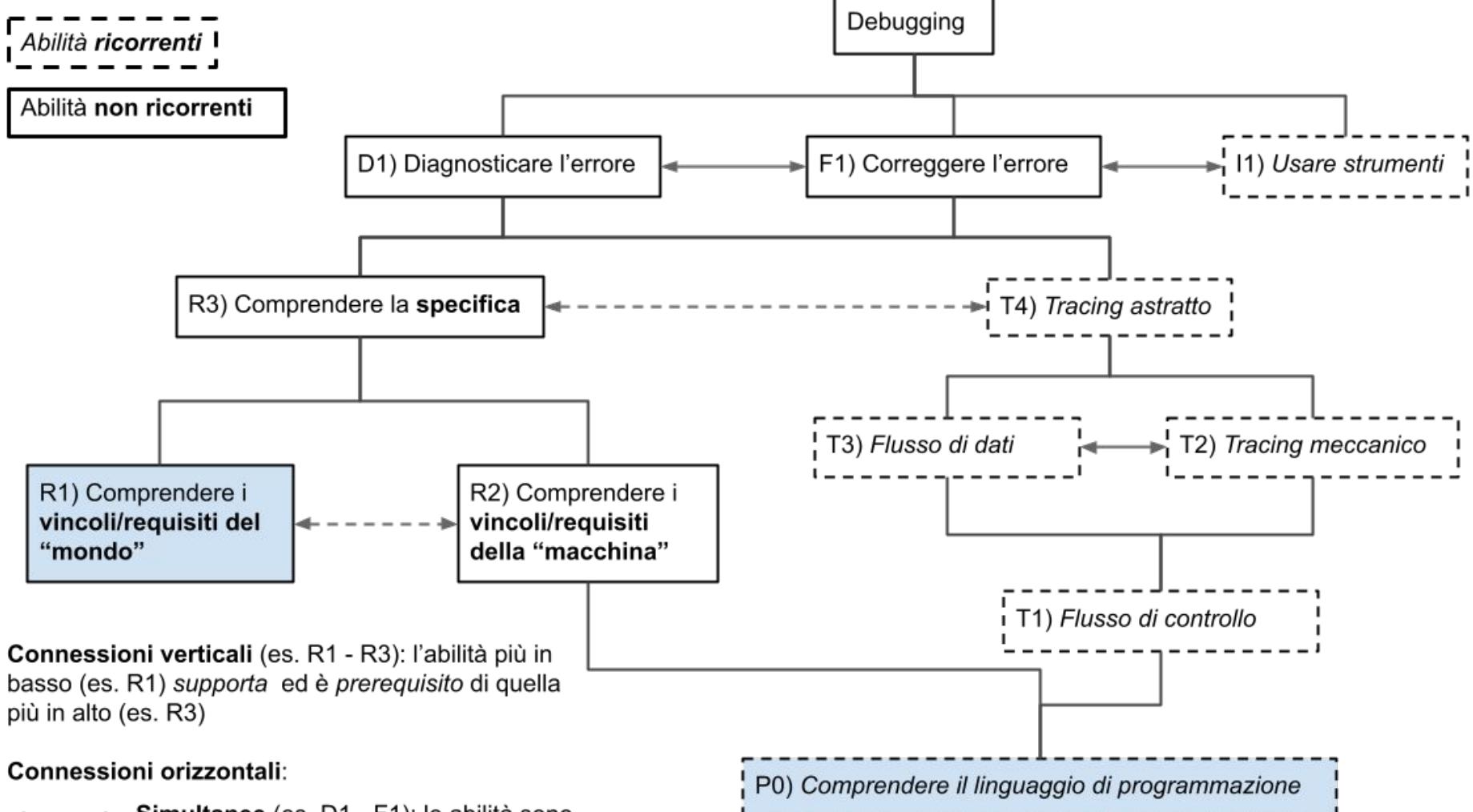
- Come deve essere usato?
- Quali sono le **aspettative** (nel senso di “desideri”)? Cosa stiamo **assumendo**? Cosa stiamo dando per scontato?
- Qual’è il “**problema**” che questo programma sta cercando di risolvere?
 - Conosciamo il **dominio** del problema da risolvere?



Esercizio 4

- “Calcolare l’elemento di valore massimo in un array di 5 interi.”
 - Input: **[2,6,4,5,1]**
 - Output?
- “Data una costante intera SOGLIA, stampare il più piccolo numero maggiore di SOGLIA che non sia divisibile per i seguenti numeri: 2, 3, 5.”
 - Input: **SOGLIA = 4**
 - Output?
- “Dati due interi n1 e n2 tali che $n1 > 0$, $n2 > 0$ e $n1 \leq n2$, stampare le tabelline di tutti gli interi n tali che $n1 \leq n \leq n2$. Per ogni valore n stampare quindi “ $n*1\ n*2\ n*3\ …\ n*10$ ”, separando ogni valore con uno spazio, e poi andare a capo.”
 - Input: **n1 = 3, n2 = 5**
 - Output?

Fonte: Prather, J., Pettit, R., Becker, B. A., Denny, P., Loksa, D., Peters, A., ... & Masci, K. (2019, February). **First things first: Providing metacognitive scaffolding for interpreting problem prompts.** In Proceedings of the 50th ACM technical symposium on computer science education (pp. 531-537).



Connessioni verticali (es. R1 - R3): l'abilità più in basso (es. R1) *supporta* ed è *prerequisito* di quella più in alto (es. R3)

Connessioni orizzontali:

↔ **Simultanee** (es. D1 - F1): le abilità sono applicate simultaneamente (in comunicazione reciproca)

↔ **Trasponibili** (es. R1 - R2): le abilità possono essere applicate in qualsiasi ordine, anche simultaneamente

R2) Comprendere i vincoli/requisiti della “macchina”

La **macchina** in questo contesto unisce

- **L'agente meccanico** che effettivamente esegue il programma (computer, tablet, persona, ecc.)
 - **Limitato** in: capacità (memoria), velocità (processore), comunicazione (rete?), ecc.
- **Il linguaggio di programmazione**
 - Inteso per applicazioni specifiche
 - Rispetta una serie di regole, convenzioni, assunzioni che possono non essere universali
 - Esempio: per i linguaggi imperativi è fondamentale il concetto di stato di esecuzione
 - Questo non vale per i linguaggi funzionali

Esempio: in Python questa espressione è vera (True) o falsa (False)? (*)

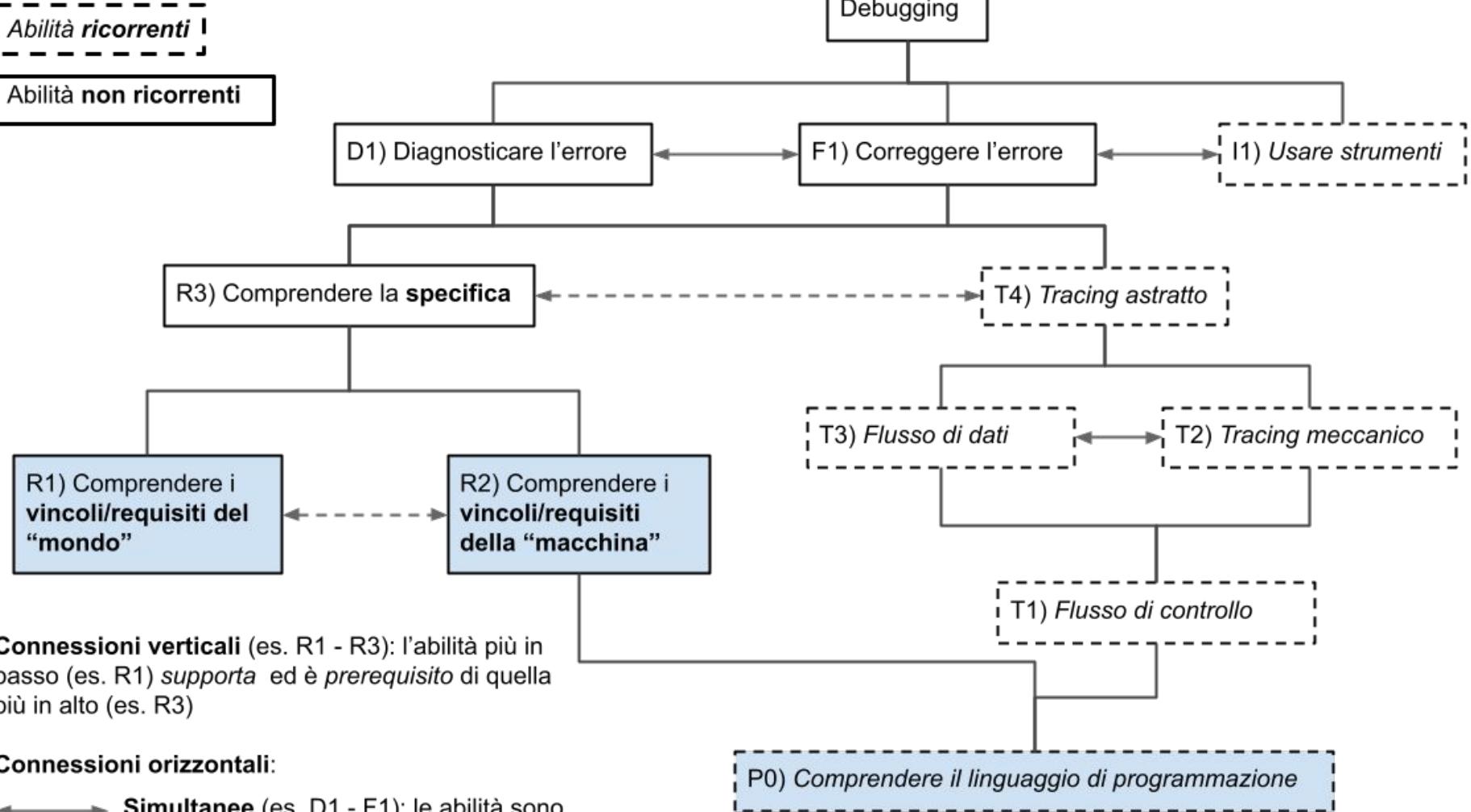
$$0.1 + 0.1 + 0.1 == 0.3$$



(*) Spiegazione approfondita: <https://docs.python.org/3/tutorial/floatingpoint.html>

Abilità ricorrenti

Abilità non ricorrenti



Connessioni verticali (es. R1 - R3): l'abilità più in basso (es. R1) *supporta* ed è *prerequisito* di quella più in alto (es. R3)

Connessioni orizzontali:

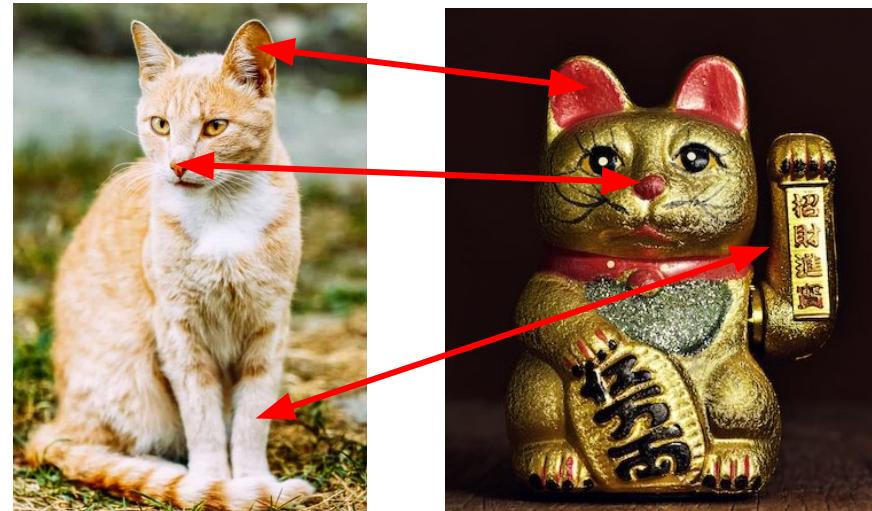
↔ **Simultanee** (es. D1 - F1): le abilità sono applicate simultaneamente (in comunicazione reciproca)

↔ → **Trasponibili** (es. R1 - R2): le abilità possono essere applicate in qualsiasi ordine, anche simultaneamente

R3) Comprendere la specifica

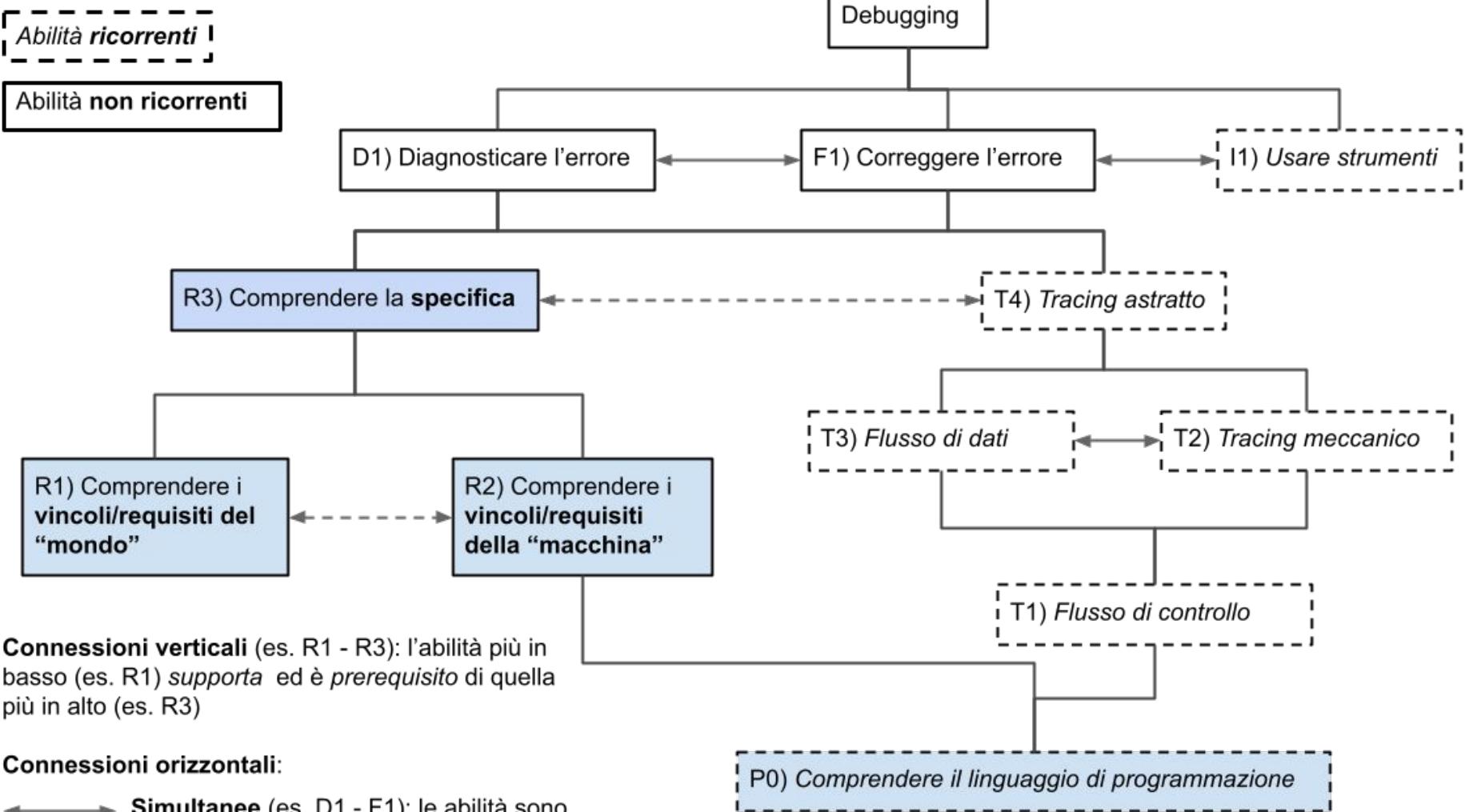
La **specificità** sta nell'intersezione tra i requisiti del “mondo” e i requisiti della “macchina”

In un certo senso è una **mappa** tra i pezzi del “mondo” e i pezzi della “macchina”



Esempio:

- Voglio avere notifiche in tempo reale su una applicazione web
 - Questo è un requisito del “mondo”
- Gli strumenti che ho a disposizione mi permettono solo di fare richieste dal client (l'applicazione che eseguo sul mio browser) al server (dove avviene l’"evento" di cui voglio avere notifica)
 - Questo è un vincolo della “macchina”
- Posso ottenere lo stesso effetto di una comunicazione dal server al client facendo “polling”, ovvero dicendo al client di fare richieste regolari al server
 - Questa è la specifica che spiega come soddisfiamo i requisiti del “mondo” entro i vincoli della “macchina”



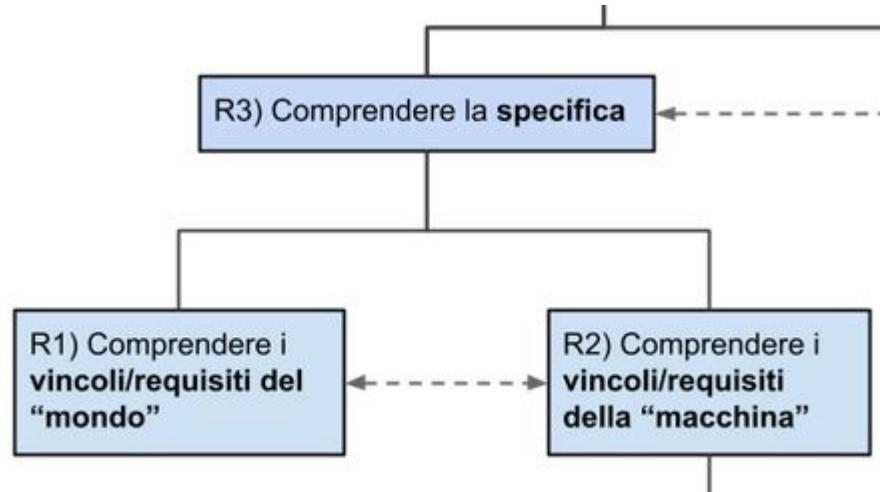
Connessioni verticali (es. R1 - R3): l'abilità più in basso (es. R1) *supporta* ed è *prerequisito* di quella più in alto (es. R3)

Connessioni orizzontali:

- ↔ **Simultanee** (es. D1 - F1): le abilità sono applicate simultaneamente (in comunicazione reciproca)
- - - → **Trasponibili** (es. R1 - R2): le abilità possono essere applicate in qualsiasi ordine, anche simultaneamente

R1..3) Il programma “atteso”

Si tratta sostanzialmente di capire cosa il programma **sta “cercando” di fare**

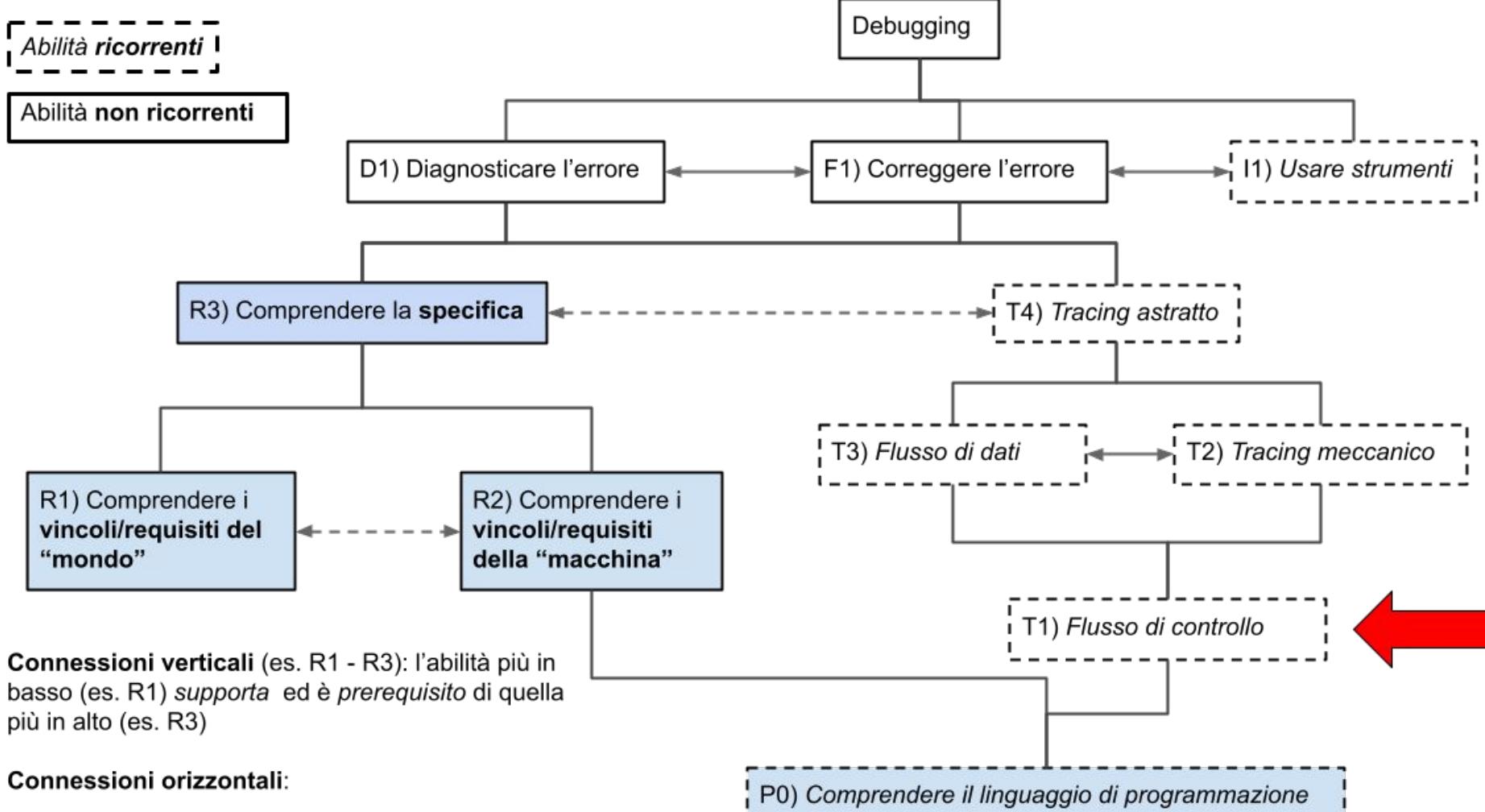


Se siamo noi ad aver creato il programma potrebbe sembrare ovvio, ma non è detto

- Separare la “soluzione algoritmica” dalla “soluzione eseguibile”?
- Es: disegna un diagramma di flusso che riassuma la tua soluzione prima di implementare il programma

Se il programma lo ha creato un’altra persona è un mix di comprensione del programma “effettivo” e di conoscenze pregresse

- I “problemi” Informatici spesso hanno soluzioni canoniche: **design pattern**
- I linguaggi di programmazione hanno spesso uno stile cosiddetto **idiomatico**
- Altri parametri che possono variare il livello di difficoltà: documentazione, convenzioni per assegnare nomi a variabili, ...



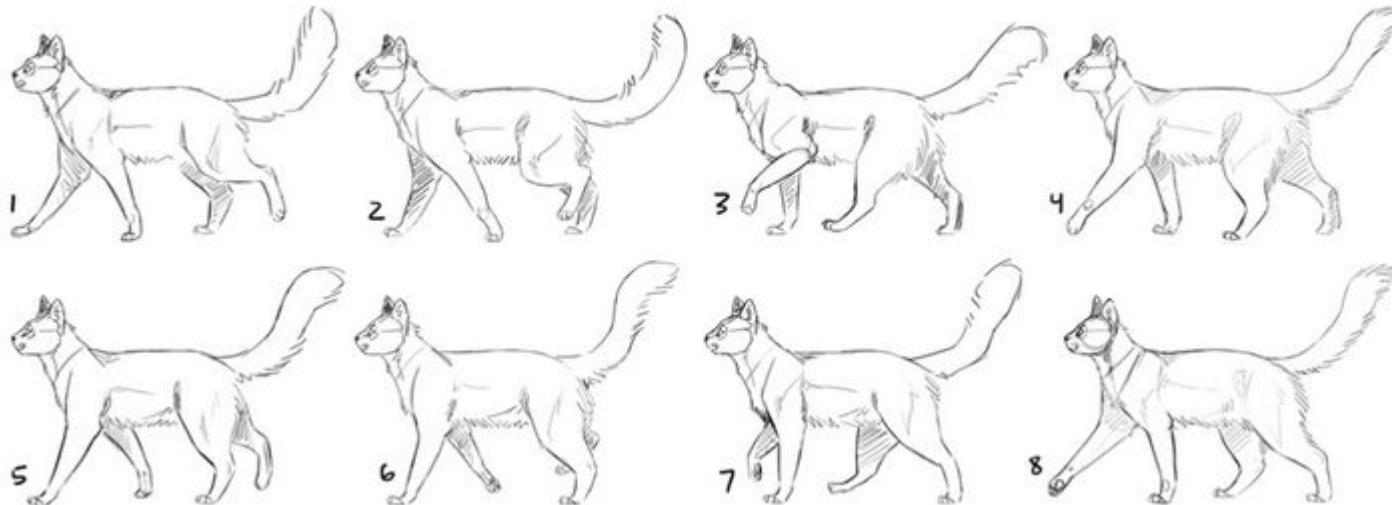
Connessioni verticali (es. R1 - R3): l'abilità più in basso (es. R1) *supporta* ed è *prerequisito* di quella più in alto (es. R3)

Connessioni orizzontali:

↔ **Simultanee** (es. D1 - F1): le abilità sono applicate simultaneamente (in comunicazione reciproca)

- - - → **Trasponibili** (es. R1 - R2): le abilità possono essere applicate in qualsiasi ordine, anche simultaneamente

T1) Flusso di controllo



Ha a che fare con la **sequenza di istruzioni** eseguite dal programma

- Quante istruzioni?
- Quali istruzioni?
- In che ordine vengono eseguite (a partire da un insieme di variabili predefinite)?
- Quali sono le regole che invece valgono per qualsiasi valore delle variabili?

Esercizio 5 (1)

Etichettare (A, B, C, ...) le istruzioni di questo programma e poi scrivere in che sequenza vengono eseguite.

```
1| var1 = 13
2| var2 = 12
3| var1 = var1 * 100
4| var1 = var1 + var2
```

Esercizio 5 (2)

Etichettare (A, B, C, ...) le istruzioni di questo programma e poi scrivere in che sequenza vengono eseguite.

```
1| var1 = 12
2| var2 = 13
3| i = 0
4| while i < 2:
5|     var2 = var2 * 10
6|     i = i + 1
7| var2 = var2 + var1
```

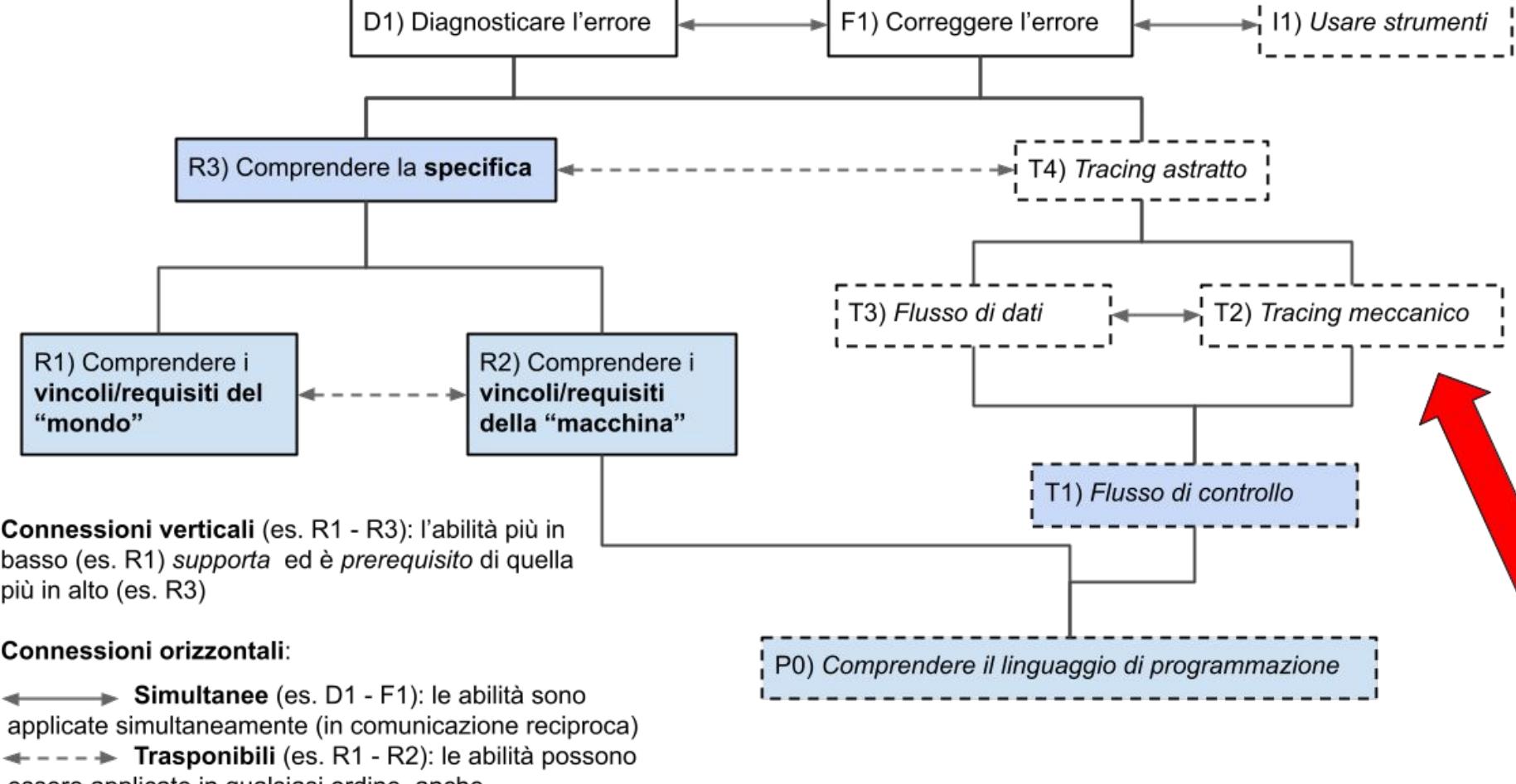
Esercizio 5 (3)

Etichettare (A, B, C, ...) le istruzioni di questo programma e poi scrivere in che sequenza vengono eseguite.

```
1 |     var0 = 0
2 |
3 |     def somma (a, b):
4 |         global var0
5 |         var0 = var0 + 1
6 |         return a + b
7 |
8 |     var1 = 40
9 |     var2 = 2
10|    var3 = somma(var1, var2) + var0
```

Abilità ricorrenti

Abilità non ricorrenti

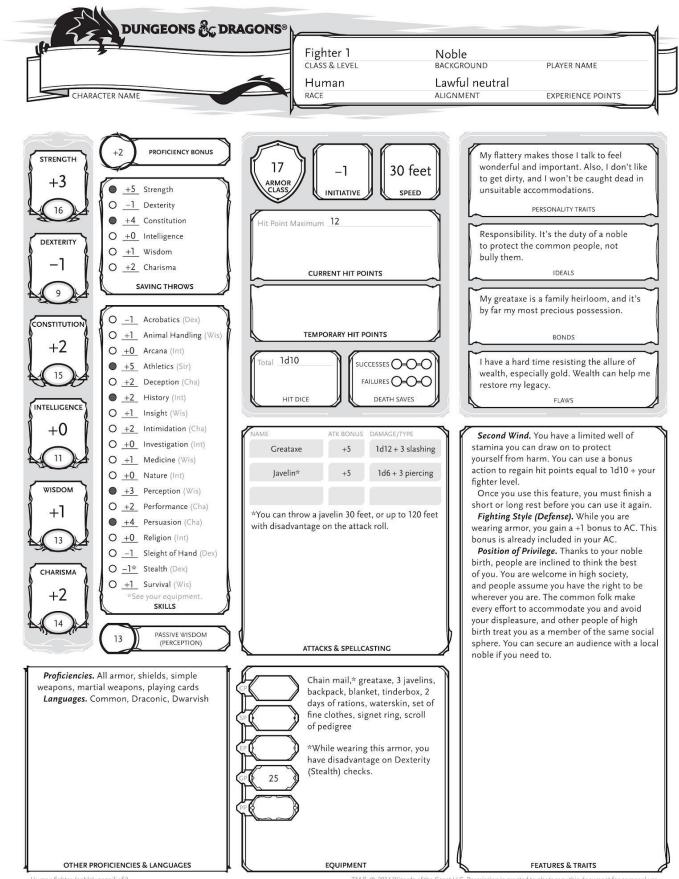


T2) Tracing meccanico

Fare **tracing** vuol dire **tenere traccia** dell'evoluzione dello **stato** di esecuzione mano a mano che un programma viene eseguito.

Lo **stato** è:

- L'insieme dei valori conservati in memoria che vengono manipolati dal programma (per esempio le variabili)
- Il puntatore alla prossima istruzione da eseguire



Il **tracing meccanico** è la sequenza di “fotogrammi” che rappresentano l'evoluzione dello stato passo per passo, a partire da situazione iniziale “fissa”.

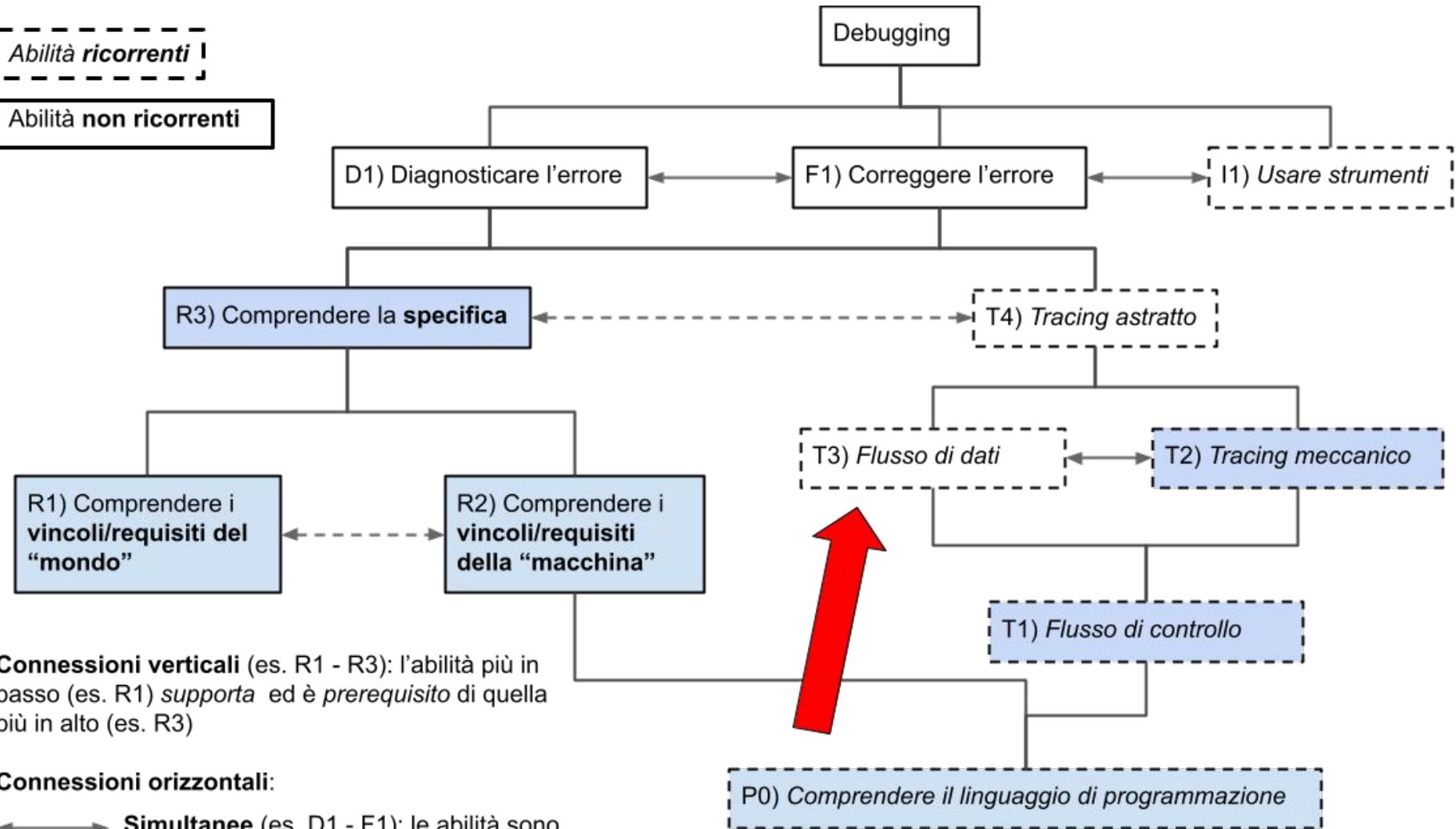
Esercizio 6

Costruire la tabella di tracing per il programma seguente.

```
1| limite = 7
2| precedente = 0
3| corrente = 1
4| prossimo = 1
5| while prossimo < limite:
6|     precedente = corrente
7|     corrente = prossimo
8|     prossimo = precedente + corrente
```

Abilità ricorrenti !

Abilità non ricorrenti



Connessioni verticali (es. R1 - R3): l'abilità più in basso (es. R1) *supporta* ed è *prerequisto* di quella più in alto (es. R3)

Connessioni orizzontali:

↔ **Simultanee** (es. D1 - F1): le abilità sono applicate simultaneamente (in comunicazione reciproca)

↔ - - - → **Trasponibili** (es. R1 - R2): le abilità possono essere applicate in qualsiasi ordine, anche simultaneamente

Nell'**esercizio 6** abbiamo visto che, generalmente, un'**istruzione** manipola (in lettura o scrittura) un insieme **limitato** di **variabili**.

Ragionare sul flusso di dati vuol dire costruire una mappa di quali parti di un programma “hanno a che fare” con specifiche variabili.

Questo permette di fare alcune operazioni avanzate e utili:

- **Tracing a ritroso:** è una strategia più avanzata per individuare bug e problemi, di solito si fa a partire da una variabile il cui valore è evidentemente “sbagliato” a un certo momento dell’esecuzione, seguendo a ritroso le istruzioni che la scrivono
- Permette di escludere “a priori” parti del programma che non hanno a che fare con una variabile “problematica”: questo è ottimo dal punto di vista del carico cognitivo
 - La strategia di debug “Wolf fence” fa questo



Analizziamo questo programma che risolve l'esercizio:

“Scrivi un programma che, data una **lista di numeri interi**, calcoli sia la **somma** dei valori contenuti nella lista sia il **valore massimo** contenuto nella lista”

Ripasso R1

Input

```
lista = [ 5, 2, 7, 1, 1, 6 ]
```

Output

“somma: ???, max: ???”

Esercizio 7 (2)

Input

```
lista = [5, 2, 7, 1, 1, 6]
```

Output

```
"somma: 22 max: 7"
```

```
1 | lista = [5, 2, 7, 1, 1, 6]
2 | i = 0
3 | max_val = -1
4 | somma = 0
5 | while i < len(lista):
6 |     somma = somma + lista[i]
7 |     if lista[i] > max_val:
8 |         max_val = i
9 |     i = i + 1
10| print("somma:", somma, "max:", max_val)
```

Volevamo questo

Input

```
lista = [5, 2, 7, 1, 1, 6]
```

Output

```
"somma: 22 max: 5"
```

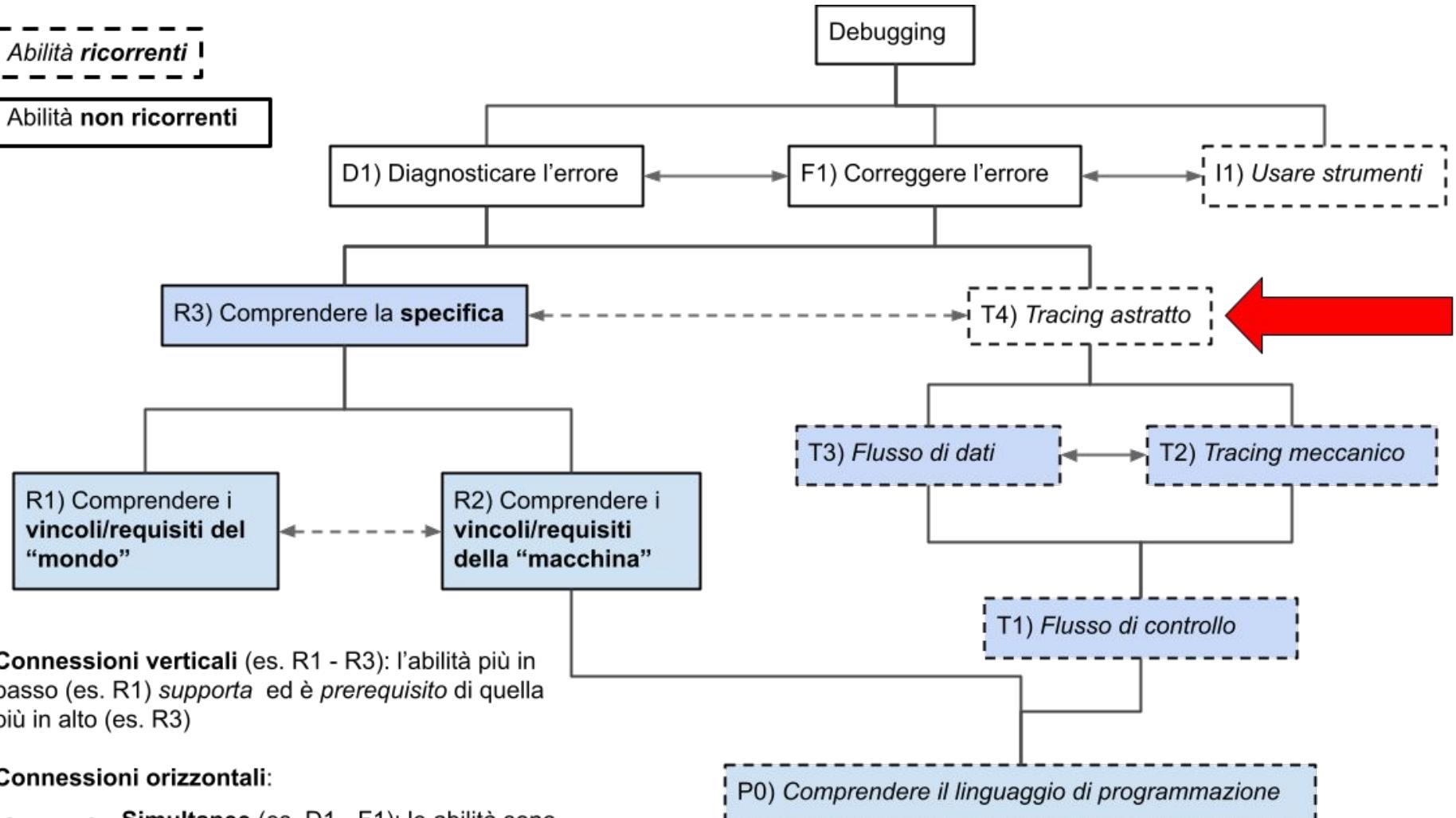
E invece abbiamo
questo...

Esercizio 7 (3)

```
1 | lista = [5, 2, 7, 1, 1, 6]
2 | i = 0
3 | max_val = -1
4 | somma = 0
5 | while i < len(lista):
6 |     somma = somma + lista[i]
7 |     if lista[i] > max_val:
8 |         max_val = i
9 |     i = i + 1
10| print("somma:", somma, "max:", max_val)
```

Abilità ricorrenti

Abilità non ricorrenti



Connessioni verticali (es. R1 - R3): l'abilità più in basso (es. R1) *supporta* ed è *prerequisito* di quella più in alto (es. R3)

Connessioni orizzontali:

↔ **Simultanee** (es. D1 - F1): le abilità sono applicate simultaneamente (in comunicazione reciproca)

↔ ↔ **Trasponibili** (es. R1 - R2): le abilità possono essere applicate in qualsiasi ordine, anche simultaneamente

T4) Tracing astratto

Mettendo insieme la capacità di capire il **flusso di controllo**, il **flusso di dati**, il modo in cui lo **stato è manipolato** dalle **istruzioni** si arriva a fare **tracing astratto**

A questo livello il programma è compreso come

- Una serie di “**pezzi**”
- Le **relazioni** tra questi pezzi
- Una serie di **aspettative** (nel senso di “contratti”) sul comportamento dei pezzi
 - **Precondizioni**: cose vere a monte dell’esecuzione di un pezzo di codice (per esempio: tutti i valori in ingresso a questo funzione sono positivi)
 - **Postcondizioni**: cose vere a valle dell’esecuzione di un pezzo di codice (per esempio: questa funzione restituisce solo valori interi)
 - **Invarianti**: cose sempre vere (per esempio: all’iterazione n-esima di un ciclo while, la variabile indice i avrà valore n-1)

```
1| i = 0
2| while i < 10:
3|     print(i)
4|     i = i + 1
```

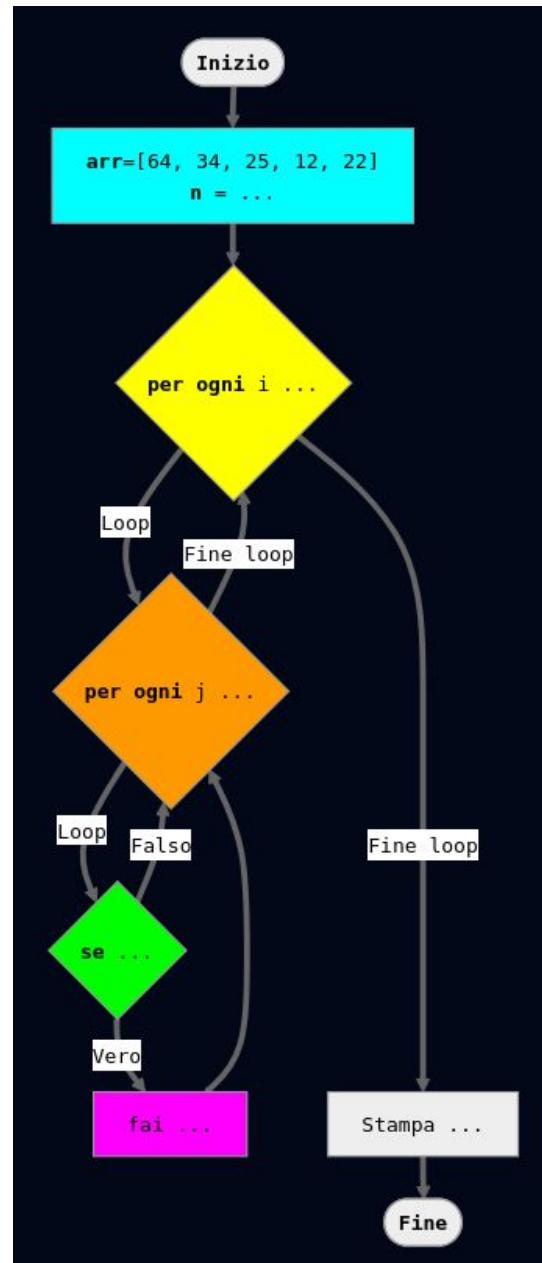
Esercizio 8

Traduciamo da **programma a diagramma** (prima abbiamo fatto l'opposto), con numero **fisso** di nodi.

```

1 | arr = [64, 34, 25, 12, 22]
2 | n_elementi = len(arr)
3 |
4 | i = 0
5 | while i < n_elementi:
6 |
7 |     j = 0
8 |     while j < (n_elementi - i - 1):
9 |
10|         if arr[j] > arr[j+1]:
11|             temp = arr[j]
12|             arr[j] = arr[j+1]
13|             arr[j+1] = temp
14|
15|         j = j + 1
16|
17|     i = i + 1
18|
19| print(arr)

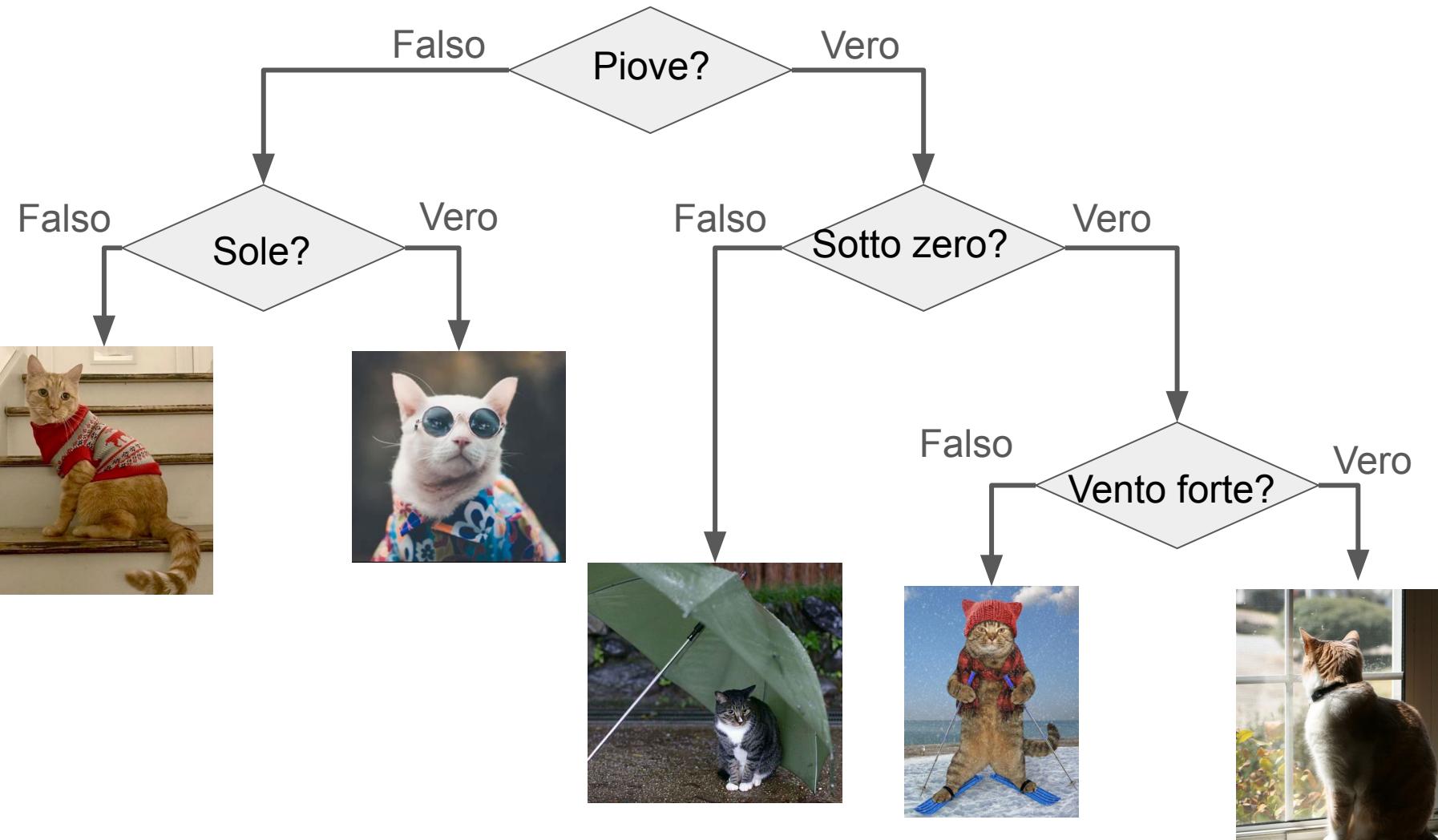
```



Esercizio 9

Proviamo a ragionare sulle **aspettative** che possiamo fissare in un semplice algoritmo.

Questo algoritmo sceglie i vestiti per uscire di casa.



Esercizio 10

In generale ci stiamo muovendo verso la **descrizione “a livello generale” di programmi**, un tipo di esercizio che viene chiamato **explain in plain English** e tradotto potrebbe essere: spiega il programma a parole tue

Variante: per ogni opzione che non descrive il programma, fornisci un esempio di input-output che dimostra perché

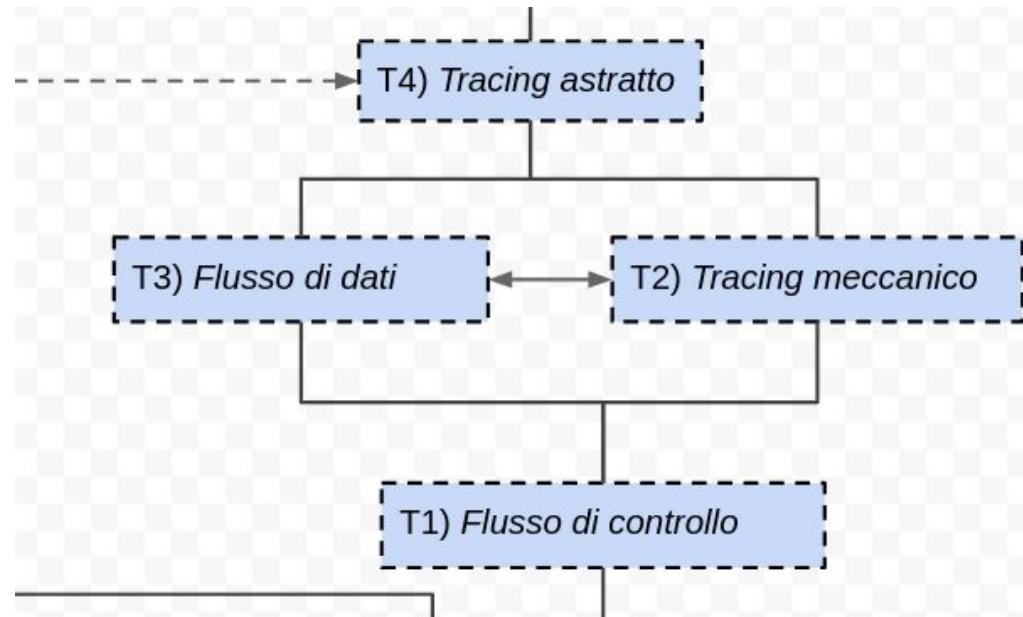
```
1| def funzione (b, e):  
2|     r = 1  
3|     while e > 0:  
4|         r = r * b  
5|         e = e - 1  
6|     return r
```

- A. Restituisce il valore della variabile **b** moltiplicato per **e**
- B. Restituisce il valore della variabile **b** elevato a potenza **e**
- C. Restituisce la somma delle variabili **b** e **e**
- D. Restituisce **b * e * (e - 1) * (e - 2) * ... * 1**

Fonte: Kumar, V. (2021, August). **Refute: an alternative to ‘explain in plain English’ questions.** In Proceedings of the 17th ACM Conference on International Computing Education Research (pp. 438-440).

T1..4) Il programma “effettivo”

Studiare il programma per capire cosa fa effettivamente.



Questo tipo di abilità è considerato in certi ambienti di ricerca come un importante **precursore** a qualsiasi produzione di programmi “nuovi”.

Vedere ad esempio: Lister, R. (2020, October). **On the cognitive development of the novice programmer: and the development of a computing education researcher.** In Proceedings of the 9th computer science education research conference (pp. 1-15).

Un'altra tecnica didattica interessante che fa molta leva sulla comprensione del **programma effettivo** è **PRIMM (Predict, Run, Investigate, Modify, Make)**

Vedere: Sentance, S., Waite, J., & Kallia, M. (2019). **Teaching computer programming with PRIMM: a sociocultural perspective**. Computer Science Education, 29(2-3), 136-176.

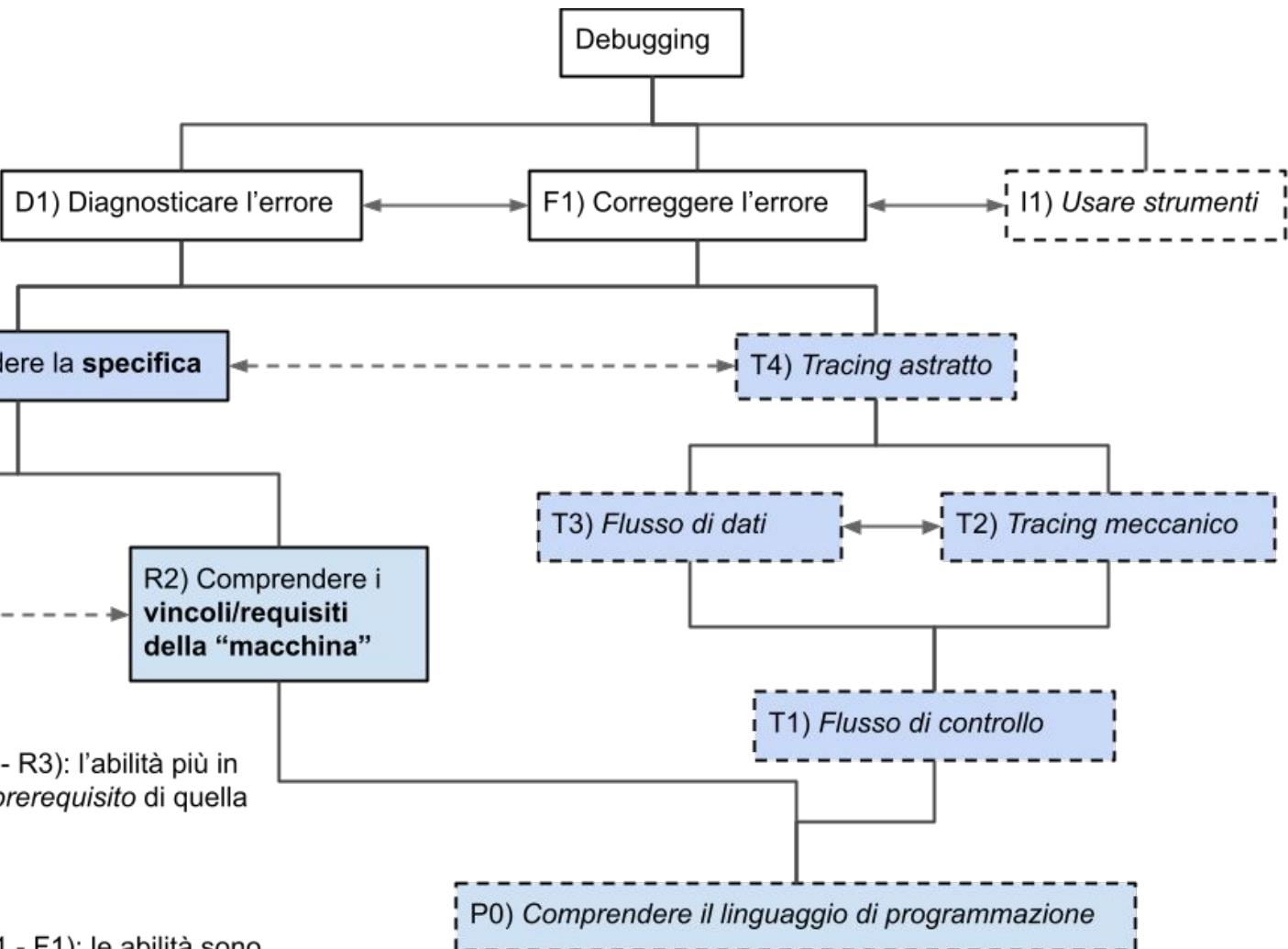
Attività didattiche “lunghe” durante le quali, dato un **programma** scelto dall’insegnante, si seguono queste fasi in cui studentesse e studenti:

- **Predict**: leggono il programma e predicono l'esito dell'esecuzione (T2)
- **Run**: eseguono il programma e verificano la risposta data al punto precedente
- **Investigate**: rispondono a una serie di domande sul programma, per esempio compilando tabelle di tracing (T2), compilando tabelle di data flow (T3), estraendo precondizioni e postcondizioni (T4) ecc.
- **Modify**: estendono il programma con nuove funzionalità oppure correggendo errori
- **Make**: creano un nuovo programma simile a quello studiato durante l’attività

Questi principi possono essere applicati anche al **debugging**

Abilità ricorrenti

Abilità non ricorrenti



Connessioni verticali (es. R1 - R3): l'abilità più in basso (es. R1) *supporta* ed è *prerequisito* di quella più in alto (es. R3)

Connessioni orizzontali:

↔ **Simultanee** (es. D1 - F1): le abilità sono applicate simultaneamente (in comunicazione reciproca)

↔ ↔ **Trasponibili** (es. R1 - R2): le abilità possono essere applicate in qualsiasi ordine, anche simultaneamente

D1) Diagnosticare l'errore

Si tratta di applicare le abilità viste fino ad ora per **dimostrare** la presenza di un errore

- Mostrando che il flusso di controllo non arriva dove si desidera
- Mostrando che una variabile assume un valore non desiderato
- Mostrando che una variabile viene scritta/letta da una istruzione non desiderata
- Mostrando che una precondizione, postcondizione ecc. viene violata



L'enfasi dovrebbe essere sul **produrre prove** dell'esistenza dell'errore



Bug Detective



F1) Correggere l'errore

Dopo aver fatto **tutto il resto** questo dovrebbe essere un passaggio “semplice”

Anche qui vale principalmente un ragionamento sul **carico cognitivo**

- Sarebbe meglio correggere l'errore come “seconda fase” dopo la diagnosi
- Potrebbe aver senso fare qualche esercizio in cui viene esplicitamente indicata l'istruzione che contiene l'errore (semplificando di molto la diagnosi)

I1) Usare strumenti

In generale, imparare a usare gli strumenti **costa** in termini di carico cognitivo

Se si vogliono usare sarebbe utile pensare a delle attività (esercizi) specificamente mirate ad insegnare l'uso degli strumenti (senza altri obiettivi di apprendimento, almeno all'inizio).

