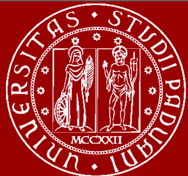
[ITADINFO]

2° CONVEGNO ITALIANO
SULLA DIDATTICA DELL'INFORMATICA

Trovare il lupo: cassetta degli attrezzi per insegnare il debugging a scuola

Gabriele Pozzan
Tullio Vardanega

19 Ottobre 2024, Genova



Università degli Studi di Padova

DIPARTIMENTO MATEMATICA
DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

Trovare il **lupo** = individuare **errori** di programmazione

- Usando una **strategia** di ricerca **sistematica**
- Usando le **informazioni** che l'ambiente di programmazione mette a nostra disposizione
- **Costruendo** programmi dove sia più facile individuare gli errori
- **Costruendo** programmi dove sia più difficile introdurre nuovi errori
- **Individuare** errori + **correggere** errori = **debugging**

Cassetta degli **attrezzi**

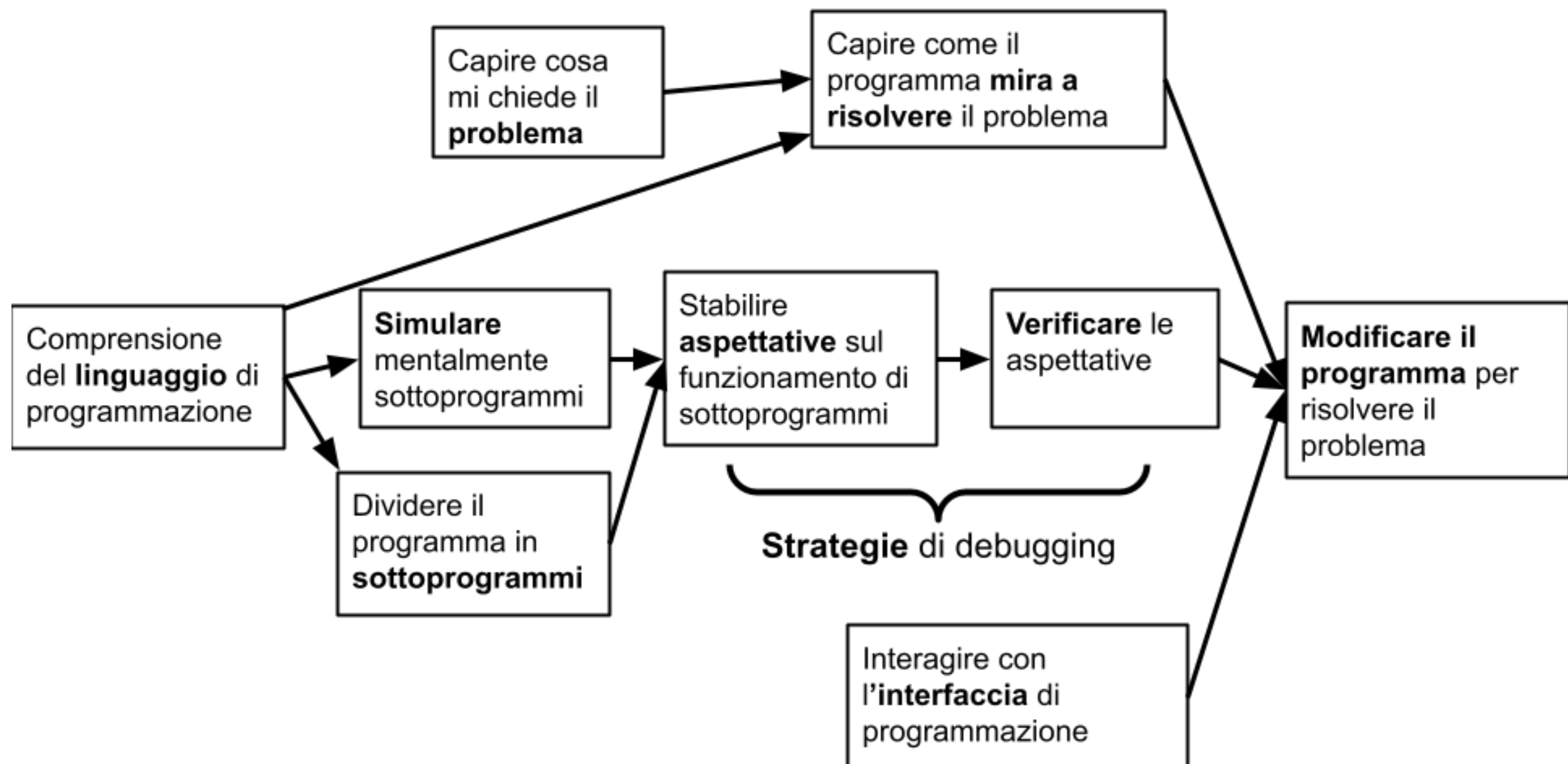
- Vedremo una semplice **strategia** di ricerca sistematica
- Vedremo come può essere **applicata** a diversi ambienti di programmazione
- Faremo degli **esercizi** che possono essere riproposti e usati per costruire altre attività didattiche

Andremo a **cercare il lupo**…

Il **lupo** è nostro **amico!**

Perché affrontare il **debugging esplicitamente**?

- È **garantito** che chi impara a programmare debba imparare a fare debugging
  - Chi programma passa **molto tempo** a fare debugging (intorno al 50% del tempo secondo qualcuno)
  - Chi programma passa molto tempo lavorando (cioè ampliando e/o **correggendo**) su codice scritto **da altre persone**
- Cosa bisogna saper fare per fare debugging in modo efficace?



Attività 1

Alla ricerca del lupo

Abbiamo sperimentato i vantaggi dell'uso di un approccio **sistematico** per la ricerca di errori chiamato **Wolf "Fence" Algorithm**

1) "**Sentire gli ululati del lupo**"
- **Sintomi** precisi di malfunzionamenti
- In qualche modo **verificabili**
- **Aspettative** vs. osservazioni/verifiche

2) "**Costruire barriere**"
- Isolare e verificare sottoprogrammi
- **Irrobustire** i programmi
- Ridurre la quantità di <u>cose a cui pensare contemporaneamente</u>

Cf. Gauss, Edward J. "**The "Wolf Fence" algorithm for debugging.**" *Communications of the ACM* 25.11 (1982): 780.

Attività 2

Un esercizio di Code.org

https://studio.code.org/s/course2/lessons/10/levels/3

Aspettative su questo problema

- L'ape deve fare un **certo percorso**
  - <u>Idea per esercizio</u>: **disegnare** il percorso prima di programmarlo
- L'ape deve **raccogliere 1 nettare sui due fiori**

Come si **ascoltano gli ululati** in questo esercizio?

- L'interfaccia ci mette a disposizione un bottone "*Fai un passo*" che permette di eseguire il programma **un blocco alla volta**
- L'ululato si "sente" quando l'ape fa una azione **diversa** dalla mia **aspettativa**

Clicca su "Fai un passo" per eseguire il prog
Blocchi
vai avanti
gira a sinistra ↺
gira a destra ↻
prendi il nettare
fai il miele
quando si clicca su "Esegui"
vai avanti
prendi il nettare
gira a destra ↻
vai avanti
gira a destra ↻
vai avanti
prendi il nettare
inizia
Fai un passo

Il blocco evidenziato contiene un errore (in questo caso) ed è sicuramente "**vicino**" al bug.

Attenzione! Il collegamento tra le due informazioni (mappa - codice) non è ovvio, soprattutto per principianti!

- L'istinto è spesso "**buttare via e ricominciare**"
  - In Code.org è facilissimo farlo, ma questa cosa succede anche con linguaggi di programmazione "classici"
  - Costruire un programma pezzo per pezzo è come leggere usando il dito per tenere il segno
  - Quando si toglie il dito **non si trova più** il segno

- Per evitare questa cosa si può allenare la capacità di leggere e **simulare mentalmente** il codice (*tracing*)

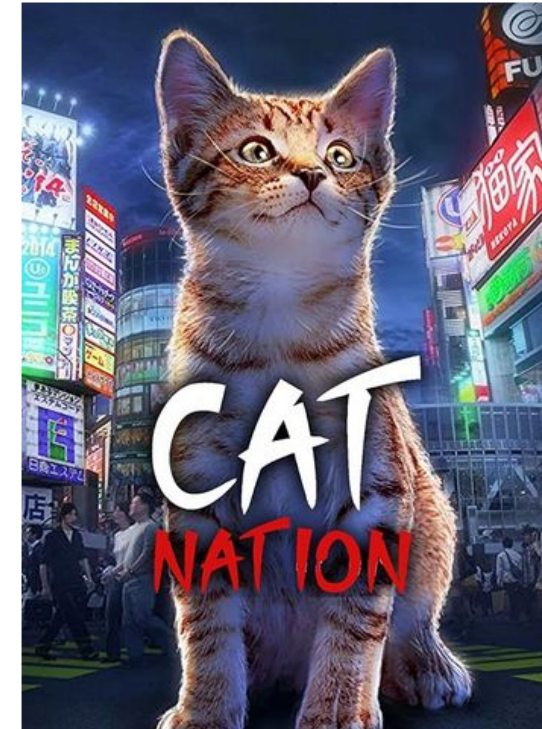
What will be the result of executing these blocks on this map?

```
when run
move forward
get nectar
get nectar
```

Prima di procedere, alcune definizioni…

Quando parliamo di **variabili** intendiamo i valori che vengono manipolati dall'esecuzione del programma

- Sono identificate da un **nome**
- Possono essere numeri, testo, ma anche combinazioni più complesse
- Esempio: *temperatura = 100*, *animale = "gatto"*

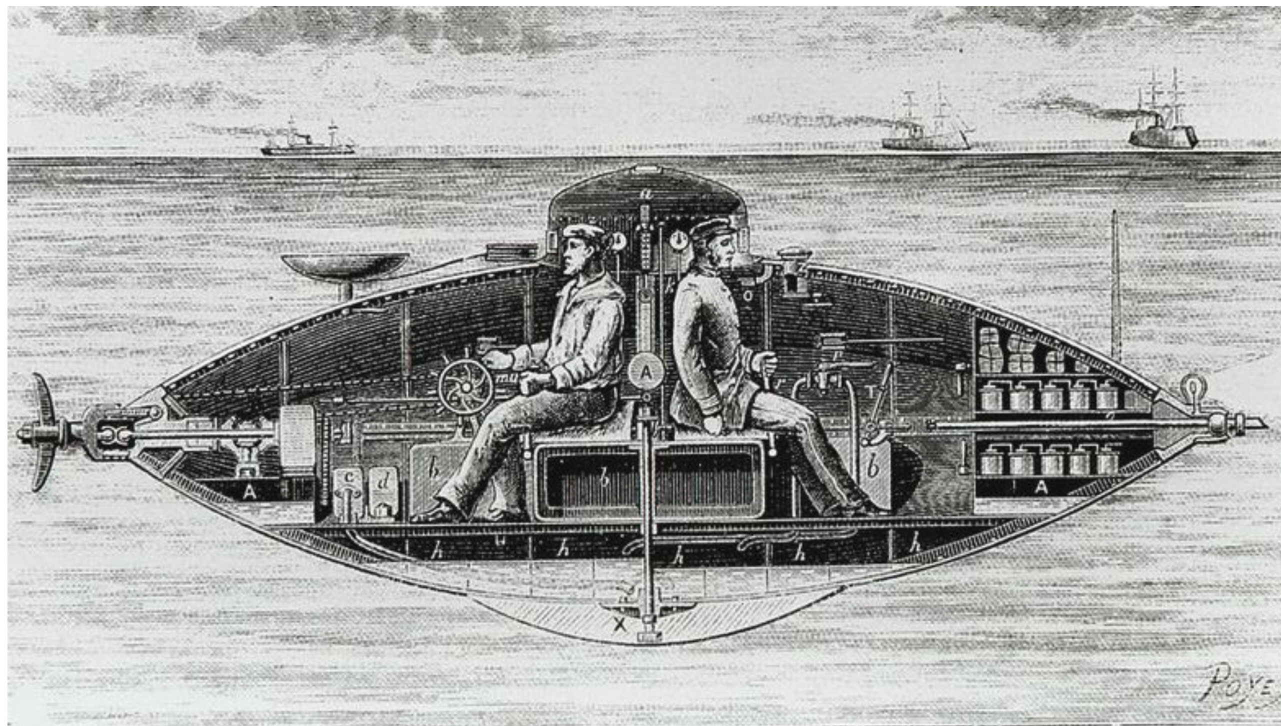


Quando parliamo di **stato dell'esecuzione** intendiamo l'insieme di variabili che vengono manipolate dal programma

- Queste variabili hanno un valore iniziale che è sempre lo stesso all'inizio dell'esecuzione del programma
- Ogni istruzione, potenzialmente, può modificare una o più variabili
- Per esempio in Code.org il movimento dell'ape è ottenuto modificando **posizione** e **direzione**

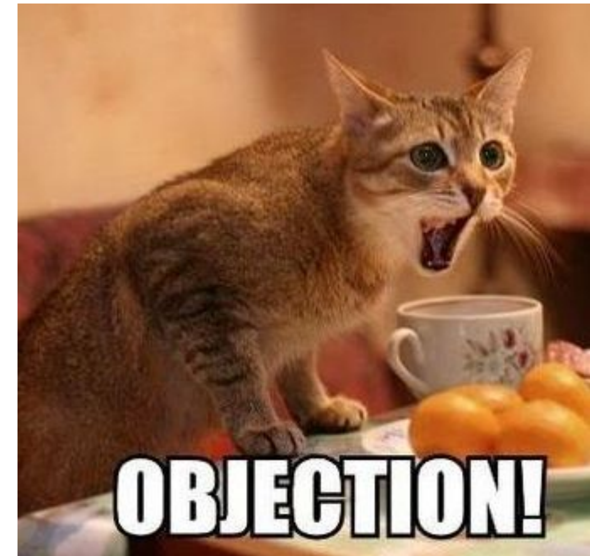
Attività 3

Ascoltare ululati nelle “viscere” di Code.org



- Gli errori non erano nei programmi di superficie ma nell'implementazione dei loro **comandi**

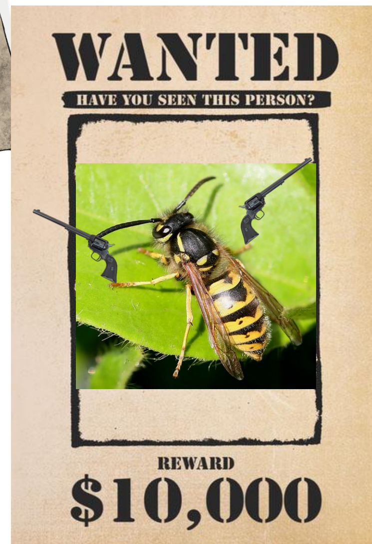
- Le definizioni dei **comandi** ci davano dei **contratti** nel senso di **aspettative misurabili/osservabili**
- Abbiamo usato le **stampe** per fare delle **verifiche su questi contratti**



- Questo non è il metodo più raffinato ma è, superficialmente, il più semplice da usare
  - … e quindi il più usato
- <u>Vantaggi? Svantaggi?</u>

Ascoltare ululati

Bug Detective



- L'obiettivo principale è **dimostrare** la presenza dei bug con prove "schiaccianti"
  - Per farlo bisogna ragionare sullo **stato** e su come si evolve durante l'esecuzione
- Per poterlo fare formuliamo delle **aspettative** sullo **stato** del programma in vari punti
  - Inizialmente lo dobbiamo fare su una versione **corretta** del programma
  - Poi si può iniziare ad avere aspettative anche su programmi "sospetti"
    - Per esempio se un programma fa la divisione di due numeri posso aspettarmi che *il divisore debba essere diverso da 0*
- Possiamo poi verificare il rispetto o mancato rispetto di queste aspettative
  - Usando le stampe o meccanismi più raffinati
  - Manipolando direttamente lo stato
  - Ci deve essere un modo migliore per farlo…

Attività 5

Costruire barriere con **test di unità**

- Insieme di **valori in ingresso** e **risultati attesi**
- I valori in ingresso vanno scelti in modo tale da **mettere alla prova** il programma
  - Li possiamo scegliere in base alle nostre aspettative di **come il programma dovrebbe comportarsi**
  - Li possiamo scegliere in base alle nostre aspettative di **quali errori** potrebbe contenere il programma (*"cosa può andare storto?"*)

Esempio: immaginiamo due possibili programmi per fare la *"divisione tra due numeri"*

- I valori in ingresso sono **due numeri**: **a** è il dividendo **b** è il divisore
- Il risultato atteso è quello che il programma dovrebbe calcolare e "ritornare"

```
Test1
Valori ingresso:
  a = 10
  b = 2
Risultato atteso:
  5
```

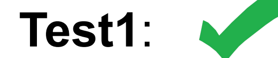
```
Test2
Valori ingresso:
  a = 10
  b = 0
Risultato atteso:
  "non definito"
```

```
dividi(a, b):
  ris = a + b
  ritorna ris
```

Test1:

Test2:

```
dividi(a, b):
  ris = a / b
  ritorna ris
```

Test1: 

Test2:

```
dividi(a, b):
  se b uguale a 0:
    ritorna "non definito"
  altrimenti:
    ris = a / b
    ritorna ris
```

Test1:

Test2: