Metodologie e tecnologie didattiche per l'informatica - Consegne

Luca Barra

Anno accademico 2023/2024

Indice

Capitolo 1

Prima consegna

1.1 Punti su cui sono d'accordo

1.1.1 Frasi

1. "*Certe attività cognitive non sono più dominio esclusivo dell'umanità: lo vediamo in tutta una serie di giochi da scacchiera (dama, scacchi, go, ...) un tempo unità di misura per l'intelligenza e nei quali ormai il computer batte regolarmente i campioni del mondo.*", **Informatica: la terza rivoluzione "dei rapporti di potere"**;

2. "*Per preparare i cittadini alla società industriale, nei due secoli passati, non sono state date agli studenti competenze operative sui macchinari industriali, ma sono state inserite nelle scuole le discipline scientifiche che ne spiegavano i princìpi scientifici alla base.*", **Informatica e competenze digitali: cosa insegnare?**;

3. "*Le tecnologie digitali dovrebbero essere progettate per promuovere la democrazia e l'inclusione.* ", **Manifesto di Vienna per l'umanesimo digitale**.

1.1.2 Motivazioni

1. Attualmente l'essere umano con l'ELO[1] più elevato è **Magnus Carlsen**[2] che ha raggiunto un picco di 2882 nella variante classica. **Stockfish**, il più forte motore scacchistico attualmente esistente, ha un ELO stimato di circa 3600. Una differenza abisssale. Inoltre è molto interessante osservare le partite tra vari computers in cui la probabilità che Stockfish perda con il bianco è quasi e, inoltre riesce anche a vincere con il nero[3]. Alcune delle mosse che vengono fatte in questi "scontri" sono basati su calcoli molto complessi e per molti esseri umani, inclusi dei granmaestri (**GM**), risultano innaturali. Oggi vengono usati *engine*, dagli stessi GM, per allenarsi poichè un essere umano non giocherà sempre la mossa migliore, un computer sì;

2. Come ricordato nell'articolo sono importanti sia le competenze digitali che quelle informatiche, tuttavia bisogna mettere in evidenza una realtà ineluttabile: le tecnologie si **evolvono continuamente**. Ciò significa che anche gli applicativi cambino nel tempo e, il compito della scuola non è solo quello di preparare i ragazzi al momento corrente, ma di fornire le capacità per **adattarsi** ai nuovi modelli. Ciò può essere fatto insegnando l'informatica perchè, anche se passa il tempo, le strutture fondamentali non ricevono quasi nessun cambiamento[4]. Questo fa si che "insegnare la programmazione" sia molto più efficace che "insegnare un determinato linguaggio" o, per mantenerci più sul generico "insegnare come si usa un editor di testo" sia più efficacie che "insegnare come si usa Word/Office";

3. Tutt@ dovrebbero essere liber@ di esprimersi liberamente nella società della tecnologia. Purtroppo l'aumento della platea dei beneficiari di queste tecnologie può aver contribuito ad aumentare l'odio verso il diverso dando voce a persone la cui opinione, un tempo, sarebbe stata relegata alle quattro mura di un bar il sabato sera. La frase in sè e per sè è corretta, ma per trasformare l'inchiostro in realtà ci vorrà ancora molto tempo e forse non si raggiungerà mai una tecnologia tale da permettere una profonda e sincera "**inclusione**".

[1] Sistema di valutazione della forza relativa di uno scacchista
[2] Campione del mondo dal 2013 al 2023
[3] Statisticamente, ad alti livelli, è molto più difficile vincere con il nero dato che non si ha il bonus dell'iniziativa
[4] Ci possono essere delle "rivoluzioni" estreme nel modo di concepire la base di una tecnologia, ma sono casi estremi e limitati

1.2 Punti su cui non sono d'accordo

1.2.1 Frasi

1. "I professionisti di tutto il mondo dovrebbero riconoscere la loro corresponsabilità nell'impatto sociale delle tecnologie digitali. Devono capire che nessuna tecnologia è neutrale ed essere sensibilizzati a considerare sia i potenziali benefici sia i possibili aspetti negativi.", **Manifesto di Vienna per l'umanesimo digitale**;

2. "Per preparare i cittadini alla società industriale, nei due secoli passati, non sono state date agli studenti competenze operative sui macchinari industriali, ma sono state inserite nelle scuole le discipline scientifiche che ne spiegavano i princìpi scientifici alla base.", **Informatica e competenze digitali: cosa insegnare?**.

1.2.2 Motivazioni

1. Nonostante io concordi con l'esaminare sia i beneficiche gli aspetti negativi non posso essere d'accordo con la parte della frase in cui si afferma che "nessuna tecnologia è neutrale". Questo perchè è l'uso che se ne fa a essere negativo o positivo. Spesso, negli ultimi anni, si sentono studiosi parlare del fatto che la tecnologia non è neutra, ma io lo vedo come un tentativo di deresponsabilizzarsi. Basta immaginarsi un mondo senza esseri umani: in tal mondo la tecnologia non potrebbe essere usata dato che non ci sarebbe nessuno capace di usarla. E se è la presenza dell'uomo a rendere una tecnologia negativa o positiva allora non è forse vero che la tecnologia è neutra?

2. Questa frase è molto interessante perchè, in un certo senso, è vero che non si è insegnato l'utilizzo dei macchinari industriali nella società industriale, ma bisogna tener presente che, all'epoca, la scuola non era ancora così accessibile. Solitamente a scuola andavano persone con un certo livello economico che raramente sarebbero diventate operai, quindi non serviva loro una formazione specifica sull'utilizzo dei macchinari. Questo tipo di formazione veniva invece fatta fare agli operai che, spesso, non avevano nemmeno la licenza elementare.

1.3 Considerazioni sui 3 paradigmi

1.3.1 Analisi dei 3 paradigmi

Definizione 1.3.1: Paradigma matematico

Per il *paradigma matematico* si formalizza un linguaggio che garantisce certe proprietà (il tutto prima dell'esecuzione).

Definizione 1.3.2: Paradigma ingegneristico

Per il *paradigma ingegneristico* si devono fare molti test di affidabilità con diversi input (unit test, a run time).

Definizione 1.3.3: Paradigma scientifico

Per il *paradigma scientifico* si deve validare empiricamente la correttezza di un programma.

Domanda 1.1

Dove si trovano, nell'informatica, questi paradigmi?

- **Paradigma matematico:** questo paradigma trova la sua massima espressione nei linguaggi funzionali. Per esempio, in Haskell[5] ogni cosa è una funzione matematica. Il punto di forza dei linguaggi funzionali (lazy) è che la loro correttezza è vera a priori per cui, se un programma viene eseguito allora è corretto.

[5] Linguaggio funzionale puro

Inoltre esistono linguaggi come Agda, Coq, etc. che servono per dimostrare matematicamente la correttezza formale di alcuni programmi (purtroppo non sono Turing completi);

- **Paradigma ingegneristico:** si basa su un intenso uso di Unit test e testing generici per assicurare una "correttezza" su un ampio insieme di casi (spesso nei casi limite). Questo implica una correttezza a posteriori per cui, il programma deve prima essere eseguito;
- **Paradigma scientifico:** si effettuano delle ipotesi e delle deduzioni. Un esempio di ciò è la logica di Floyd-Hoare in cui vengono poste delle pre-condizioni (ipotesi sui dati) e delle post-condizioni (dati attesi se il programma è corretto).

1.3.2 Conclusioni generali

Tutti e tre i paradigmi mostrano un differente modo di osservare la realtà. É difficile metterli in una classifica o dire quale sia il più corretto perchè ciò dipende in gran parte dal background personale del singolo: un logico probabilmente sarà orientato verso il paradigma matematico, un tecnico verso quello ingegneristico e un chimico o un biologo verso quello scientifico. L'informatica è qualcosa di troppo complesso per essere ridotto a un solo di questi paradigmi. Come mostrato nella sezione precedente ogni paradigma include un pezzo dell'informatica quindi è inutile affermare che appartiene al paradigma X o al paradigma Y[6]

1.4 L'informatica può essere considerata una scienza?

1.4.1 I requisiti

- Organizzati per comprendere, sfruttare e far fronte a un fenomeno pervasivo: esiste un intero campo di applicazione dell'informatica a questo livello ossia la **data science**;
- Comprende i processi naturali e artificiali del fenomeno: l'informatica vuole comprendere e replicare in modo automatico dei fenomeni sia naturali che artificiali. Basti vedere alcune strutture chiaramente ispirate alla natura come gli **alberi**;
- Corpo di conoscenza strutturato e codificato: l'informatica ha un compendio di conoscenze ben identificabile e strutturato in vari argomenti. Per esempio: le **basi di dati**, gli **algoritmi**, etc.;
- Impegno verso metodi sperimentali per la scoperta e la validazione: ci sono parti dell'informatica con carattere prevalentemente sperimentale. Si è parlato nel capitolo precedente del paradigma ingegneristico che ha come oggetto il testing per validare i programmi;
- Riproducibilità dei risultati: i risultati dei programmi sono riproducibili se si possiede un interprete per un dato linguaggio e una macchina su cui eseguirli;
- Falsificabilità di ipotesi e modelli: usando il **moduls tollens**[7] si può falsificare un programma. Per esempio, se un programma non termina allora non è corretto;
- Abilità di fare predizioni: l'informatica è in grado di predire il comportamento di un programma. Per esempio, se un programma è corretto allora termina;

1.4.2 Conclusioni

In conclusione si può ritenere l'informatica una scienza dato che soddisfa tutti i requisiti. Inoltre, come mostrato nella sezione precedente, l'informatica è un campo molto vasto che include molti altri campi. Per esempio, l'informatica è strettamente legata alla matematica e alla logica. Inoltre, l'informatica è strettamente legata alla fisica e alla chimica poichè, per esempio, un computer è un dispositivo fisico che sfrutta la chimica per funzionare. Infine, l'informatica è strettamente legata alla biologia poichè, per esempio, l'informatica è usata per studiare il DNA e per simulare la vita (si veda il Game of Life di Conway, gli automi cellulari e la vita artificiale).

[6] Per usare un termine informatico possiamo dire che l'informatica, così come C++ o Java, è **multiparadigma**

[7] Se A $\rightarrow$ B, $\neg$B $\rightarrow$ $\neg$A

Capitolo 2

Seconda consegna

Note:-

Obiettivi sulle conoscenze (K): lo studente è in grado di ***capire se un determinato pseudo-algoritmo rappresenti un algoritmo reale***. Lo studente è in grado di ***definire cosa è un algoritmo***.
Obiettivi sulle abilità (A): lo studente è in grado di ***discutere con i compagni le proprie idee***.
Obiettivi sulle competenze (C): lo studente è in grado di **riconoscere un algoritmo reale**.

2.1 Proposta di suddivisione in fasi

i. **Fase 1:** lettura e analisi, a coppie, del testo degli "algoritmi".

   (a) **Consegna:** leggere e analizzare gli pseudo-algoritmi proposti nel documento cercando di capire quali rappresentino algoritmi.

   (b) **Svolgimento:** gli alunni vengono divisi in coppie. Ogni coppia legge il testo proposto e cerchi di capire se gli algoritmi siano reali o meno.

   (c) **Discussione:** ciascun elemento della coppia discuta con l'altro quali algoritmi siano reali e quali no.

   (d) **Conclusione:** l'insegnante unisce le coppie, a due a due, e introduce la fase successiva.

ii. **Fase 2:** analisi, a gruppi, degli "algoritmi".

   (a) **Consegna:** a gruppi, confrontare le proprie risposte ottenute nella fase precedente.

   (b) **Svolgimento:** ciascuna coppia, all'interno del gruppo, discute con l'altra coppia le proprie risposte.

   (c) **Discussione:** ogni gruppo decide quali siano le risposte corrette.

   (d) **Conclusione:** l'insegnate introduce la fase successiva.

iii. **Fase 3:** definizione di algoritmo.

   (a) **Consegna:** ogni gruppo definisca che cos'è un algoritmo, partendo dalle proprie risposte.

   (b) **Svolgimento:** ciascun gruppo discute e cerca di definire che cos'è un algoritmo.

   (c) **Discussione:** ogni gruppo propone alla classe la propria definizione e la discute. L'insegnate deve mettere in risalto le differenze e le somiglianze nelle definizioni.

   (d) **Conclusione:** il docente presenta una definizione "consolidata" di algoritmo e la mette a confronto con le definizioni trovate dagli studenti.

2.2 Snodi

i. **Fase 1:**

   (a) Comprendere il testo di uno pseudo-algoritmo.

(b) Analizzare uno pseudo-algoritmo.

(c) Trovare un modo per stabilire se un algoritmo sia reale o meno.

ii. **Fase 2:**

(a) Lavorare in gruppo per trovare una soluzione condivisa.

iii. **Fase 3:**

(a) Capire che cosa si intende con la parola "algoritmo".

(b) Trovare le caratteristiche che definiscono un algoritmo.

2.3 Indicatori

i. **Fase 1:**

(a) Quali sono le caratteristiche di un algoritmo?

(b) Un algoritmo termina sempre?

(c) Un algoritmo deve essere preciso?

ii. **Fase 2:**

(a) Come mai si hanno opinioni diverse?

(b) Si hanno dei punti in comune?

iii. **Fase 3:**

(a) Commentare gli pseudo-algoritmi in modo da evidenziare le caratteristiche che un algoritmo reale deve o non deve avere.

(b) Cosa si intende per "passo" di un algoritmo?

(c) Ci sono algoritmi più efficienti di altri?

(d) Quali sono altre proprietà che un algoritmo deve avere?

Capitolo 3

Terza consegna

3.1 Le sfaccettature dei programmi

I programmi come strumenti: questo punto di vista è, probabilmente, il più comune e il più intuitivo per la maggior parte delle persone. Nel percorso scolastico, infatti, si vedono i programmi come strumenti per risolvere problemi, per esempio, calcolare la media di un voto, oppure per svolgere un compito.

I programmi come opere dell'uomo: una visione di questo tipo sorge spontanea quando si inizia a scrivere codice e ci si trova davanti a un bivio in cui ci sono più possibili implementazioni di un algoritmo. Ci si rende conto che bisogna fare delle scelte e che queste scelte possono presentare vantaggi e svantaggi.

I programmi come oggetti fisici: durante l'ultimo anno del mio percorso scolastico della scuola secondaria di secondo grado, ho potuto osservare alcuni aspetti più fondazionali dell'informatica. Per esempio, dato che un programma ci mette una determinata quantità di tempo e di energia per essere eseguito, è possibile ottimizzarlo? Così mi sono imbattuto nei concetti di complessità temporale e complessità spaziale. Inoltre ho anche potuto constatare che un programma nel "cloud" comunque verrà eseguito su un computer fisico da qualche parte nel mondo. A livello universitario ci sono molti corsi che coprono questa sfaccettatura, come "Architettura degli elaboratori", "Calcolabilità e complessità", "Sistemi operativi", etc...

I programmi come entità astratte: questo aspetto non è emerso in modo esplicito come altri, ma si può facilmente intuire mediante concetti come "varaiabile" e "funzione" che non sono comprensibili direttamente a livello fisico del calcolatore. All'università invece vengono trattati in modo approfondito, in quanto bagaglio fondamentale di ogni informatico.

I programmi come entità eseguibili: il fatto di considerare un programma come un'entità eseguibile è stato molto sottolineato dal mio professore alle superiori. Il linguaggio che ha scelto di insegnarci è stato il C che si presta molto bene a questo tipo di visione. Infatti, il C che integra caratteristiche di basso livello e deve essere compilato bene perchè possa produrre dei file eseguibili.

I programmi come manufatti linguistico-notazionali: essendo uno studente del corso di "Linguaggi e Sistemi" posso guardare, in retrospettiva, a quello che è stato insegnato alle superiori con occhio critico. Per esempio mi è stata solo spiegata la sintassi del C ed è stata liquidata con un "è così" senza spiegazioni approfondite. Un esempio è il confronto (==) che, per la maggior parte del mio percorso scolastico, è stato "oscuro" e arcaico. Quindi posso tranquillamente affermare che tra le sei sfaccettature è stata quella più trascurata.

3.2 Difficoltà nello studio dell'informatica

La difficoltà che si incontra studiando informatica, come qualsiasi altra disciplina, è altamente soggettivo, tuttavia credo che esistano degli *scogli* in cui gli studenti hanno più possibilità di inciampare. Per me, come si è già potuto intuire, fu l'aspetto sintattico e linguistico. Pur non avendo alcuna difficoltà nel risolvere problemi raramente riuscivo a prendere punteggio pieno per colpa di errori sintattici (come, per esempio = al posto di ==). Un'altra

sfaccettatura che può, potenzialmente, creare problemi riguarda la visione dei programmi come entità astratte. Può essere difficile per gli studenti concettualizzare qualcosa che non è tangibile. Si è visto, durante questo corso [1], che spesso gli alunni fraintendono il significato di "variabile" vedendola come una sorta di storico di tutti i valori che ha assunto durante l'esecuzione. Oppure si ha difficoltà a concettualizzare il concetto di "loop". Per quanto riguarda la concezione di un programma come un oggetto fisico: raramente viene affrontata in certi percorsi scolastici in cui ci si concentra più sullo scrivere codice in linguaggi di alto livello in modo scollegato dal dispositivo fisico che lo esegue. Infine, il concetto di programma come opera dell'uomo è molto difficile da affrontare in quanto richiede una certa maturità e una certa esperienza oltre a una propensione a una visione più "umanistica" e "filosofica" rispetto alle altre sfaccettature.

3.3 Informatica alle superiori: modi di vedere un programma

Primo anno: al primo anno, come prima cosa presenterei la visione come strumento, perchè è la più facile da comprendere per chi si affaccia per la prima volta a questo mondo ed è quella con i risvolti più pratici. Per esempio, il telefono che usi per divertirti o il computer che usi per fare i compiti.

Secondo anno: al secondo anno ritengo opportuno mostrare agli studenti che un programma può essere visto come entità astratta e come entità eseguibile. Queste due sfaccettature sono molto legate tra loro e si possono spiegare in modo congiunto. Esse mostrano qualcosa di non tangibile che, può fare un po' paura all'inizio, ma proprio per il fatto che non è legato a una struttura ben determinata è più interessante da studiare e mostra un numero quasi infinito di possibilità.

Terzo anno: a questo punto si potrebbe presentare, finalmente, il calcolatore e il programma come oggetto fisico. In questo modo si mostra agli allievi che tutto ciò che hanno studiato fin'ora ha anche dei risvolti fisici e che non è solo un'astrazione.

Quarto anno: al quarto anno è tempo di fare un'analisi della sintassi di un linguaggio e di come può essere scritto un programma in modo che sia leggibile e comprensibile da altre persone.

Quinto anno: giunti alla fine del percorso di studi si può iniziare a parlare di un argomento molto delicato, ossia il programma come opera dell'uomo. Ciò prevede che gli studenti, negli anni trascorsi abbiano raggiunto un'adeguata consapevolezza per poter discutere in modo critico e costruttivo lo scopo con cui un programma è stato scritto e il fatto che porta con sè le idee e i valori del suo creatore.

[1] MTDI

Capitolo 4

Quarta consegna

4.1 Classificare le misconceptions

Nella seguente tabella vengono riportate alcune misconception con il relativo tipo di difficoltà:

⇒ *sintattica*: relativa allo specifico linguaggio di programmazione che si sceglie di utilizzare (es. la dichiarazione di una variabile);

⇒ *concettuale*: relativa a un particolare concetto ricorrente nella programmazione (es. i cicli for);

⇒ *strategica*: relativa all'applicare i concetti appresi (es. dubbi sull'utilizzare una certa struttura rispetto a un'altra).

Misconception	**Tipo di difficoltà**
L'assegnamento di variabili funziona in entrambe le direzioni o in direzione inversa.	Concettuale.
I tipi primitivi, in Java, hanno dei valori di default.	Sintattica.
Una condizione falsa termina un "IF" se non c'è un "ELSE".	Concettuale.
Un metodo può essere invocato solo una volta.	Strategica.
Una funzione cambia sempre i parametri in input per farli diventare output.	Strategica.
Confusione tra una classe e una sua istanza.	Concettuale.
Per ogni oggetto si alloca sempre la stessa quantità di memoria a prescindere dall'istanza.	Concettuale.
Nella ricorsione si comprende solo l'aspetto attivo, ma non i valori di ritorno (o viceversa).	Concettuale.
Si può definire un metodo, in Java, che aggiunge attributi a una classe.	Sintattica.
I metodi possono fare solo assegnamenti.	Concettuale.
Un oggetto è soltanto un record.	Concettuale.
Difficoltà con array a più dimensioni.	Strategica.
Si confondono tipi statici e dinamici.	Concettuale.
Confusione tra un array e una sua cella.	Concettuale.
I numeri sono solo numeri (int, float, etc.)	Concettuale.
In Java, i tipi possono cambiare "on the fly".	Sintattica.
Difficoltà nel comprendere il costruttore vuoto.	Sintattica.
Due oggetti con lo stesso nome sono lo stesso oggetto.	Concettuale.
Non si possono avere più metodi con lo stesso nome in file diversi.	Concettuale.
Salvare in memoria un oggetto vuol dire memorizzare i parametri del costruttore al momento della sua inizializzazione (questi parametri definiscono l'oggetto in modo non ambiguo).	Concettuale.

4.2 Misconception comuni

In questa sezione ci occuperemo di discutere possibili soluzioni ad alcune delle misconceptions che abbiamo incontrato durante il nostro percorso scolastico/accademico.

Misconception	Analisi e Risoluzione
L'assegnamento di variabili funziona in entrambe le direzioni.	Questa misconception nasce dall'utilizzo del segno uguale (=) che, generalmente, viene associato alle equazioni e quindi funziona in entrambi i versi. Per risolverlo si può sostituire nello pseudocodice il simbolo di ugualianza con simboli analoghi (:= o $\rightarrow$) ma meno equivochi, tuttavia quando si passerà al codice vero e proprio bisognerà rinunciare a questo *scaffolding*[1] .
Un metodo può essere invocato solo una volta.	Questa visione dei metodi come "usa e getta" è relativamente facile da correggere in quanto basta far riflettere lo studente sul perchè si scelga di utilizzare un metodo invece che scrivere tutto il codice all'interno del main.
Per ogni oggetto si alloca sempre la stessa quantità di memoria a prescindere dall'istanza.	Questa misconception compare soprattutto in linguaggi che presentano un Garbage Collector (es. Java), per cui non si deve allocare e deallocare a mano la quantità di memoria che si vuole utilizzare (es. C). Per risolvere si può fare un semplice esempio: se si deve scegliere uno zaino e si sa che devono starci poche cose è inutile prenderne uno troppo grande (spreco di spazio/memoria) ma, viceversa se si devono mettere dentro molte cose uno troppo piccolo non è sufficiente.
I numeri sono solo numeri (int, float, etc.)	Quando si scrive un equazione o una disequzione su carta non ci si preoccupa troppo del tipo che essi devono avere perchè gli interi possono essere scritti come numeri decimali con infiniti zeri, ma i computer hanno una quantità di memoria finita per cui una cosa del genere è infattibile. Questo "salto cognitivo" degli studenti è anche dovuto al processo di *cast*[2]. Sebbene sia utile (soprattutto nella programmazione a oggetti) può causare confusione per via della fluidità con cui si passa da un int a un float. Per risolvere il problema è utile spiegare agli allievi come i dati vengono effettivamente memorizzati dal calcolatore.
Difficoltà con array a più dimensioni.	Un array a più dimensioni può essere complicato da concettualizzare. Una soluzione è quella di concentrarsi sulla rappresentazione a due dimensioni: per spiegarla basta disegnare una tabella e mostrare che a ogni coppia di coordinate corrisponde uno e un solo valore. Da qui si deriva che lo stesso vale per array a n dimensioni in cui si sostituisce il termine "coppia" con "n-upla".

[1] Lo scaffolding è un processo di supporto che aiuta gli studenti a ridurre il carico cognitivo.
[2] Il cast è l'operazione con cui si converte una variabile da un tipo di dato a un altro.

Il nome, in linguaggio naturale, delle variabili influenza il modo in cui verranno usate nel codice e il tipo di valori.	Questa misconception è delicata da trattare: il nome non ha un collegamento diretto con l'utilizzo della variabile (un classico esempio è: posso chiamare una variabile Pippo, Pluto o Paperino), ma bisogna anche porre l'accento sul fatto che si dovrebbero utilizzare dei *nomi significativi* (un contatore lo si chiama count) in modo da rendere più semplice la comprensione e la lettura del codice.
Dopo un "IF" sia la clausola "THEN" che la clausola "ELSE" sono eseguite.	Il fraintendimento nasce da una concezione sbagliata di "IF", la soluzione più efficace è portare lo studente a riflettere sul perchè sia necessaria una condizione booleana.
Difficoltà nel capire l'incremento del contatore in un ciclo "FOR".	La sintassi del ciclo "FOR" può apparire confusionaria all'inizio e il modo più semplice per affrontarla è quello di spezzarla in tre parti distinte: • assegnamento del valore iniziale al contatore, viene eseguita solo all'ingresso nel ciclo; • condizione booleana: come un "IF" che si ripete a ogni iterazione del ciclo; • incremento del contatore: viene eseguito in ogni iterazione successiva alla prima.
Nessun modello di ricorsione.	Per cambiare la percezione sull'impossibilità di risolvere un programma in modo ricorsivo è utile illustrare un problema dove l'utilizzo della ricorsione è "naturale", per esempio la *successione di Fibonacci*.
Due oggetti con lo stesso valore in un campo "nome" sono lo stesso oggetto.	Si può affrontare questo equivoco con un esempio: ci sono più persone che hanno lo stesso nome o lo stesso cognome, ma sono individui distinti. Così come due oggetti che hanno un campo "nome" uguale non sono lo stesso oggetto: "nome" è solo una convenzione che si è scelta per rappresentare una caratteristica di un oggetto ma, come si è visto nella misconception sul linguaggio naturale, non presenta alcun legame intrinseco con l'idea che si ha di "*identità*".
Un costruttore può includere solo assegnamenti per inizializzare gli attributi.	L'incomprensione del concetto di costruttore deriva dal fatto che *solitamente* si utilizza in questo modo, ma nulla vieta di effettuare altre operazioni al suo interno. Il modo con cui si cura questa misconception è mostrare un esempio in cui nel costruttore vengano eseguite operazioni diverse dall'assegnamento.
Un oggetto non può essere il valore di un attributo.	Basta semplicemente fare un esempio in cui si ha questa necessità: un oggetto "banca" può avere un attributo "conto" che è a sua volta un oggetto con attributi "numero", "saldo", etc...
Confusione di stringhe che presentano numberi con numeri veri e propri.	La sbagliata concezione dell stringhe "numeriche" come numeri deriva dal fatto che, nella vita quotidiana, ogni numero è di per sè una stringa (ogni simbolo può essere visto come un carattere, di fatto non c'è distinzione tra 1 e un "1") ma allo stesso tempo un numero (ci si possono effetture dei calcoli). Come per altri problemi simili la soluzione risiede nell'insegnare come un computer rappresenta e veda tutto ciò.

Capitolo 5

Quinta consegna

5.1 PRIMM

Definizione 5.1.1: PRIMM

PRIMM è un framework per la progettazione di attività didattiche per l'insegnamento della programmazione comprendente le seguenti fasi:

- **Predizione (Prediction):** gli studenti devono prevedere il comportamento di un programma;
- **Esecuzione (Run):** gli studenti devono eseguire il programma e verificare la predizione;
- **Investigazione (Investigation):** gli studenti devono correggere la predizione in caso di errore;
- **Modifica (Modify):** gli studenti devono modificare il programma in modo che si comporti in un modo diverso;
- **Risoluzione (Make):** gli studenti devono risolvere un problema usando il programma.

Pro:

⇒ L'esecuzione del codice, passo per passo, favorisce una chiara comprensione di ogni parte dello stesso;

⇒ Fornisce allo studente un riscontro immediato nella fase di esecuzione;

⇒ È molto *flessibile* in quanto non sempre sono necessarie tutte le fasi e si presta a modifiche a seconda del contesto.

Contro:

⇒ Non è applicabile a codici molto complessi e lunghi;

⇒ Nella sue fasi iniziali si fa uso di codice non appartenente allo studente che quindi può risultare difficile da capire.

5.2 POGIL

Definizione 5.2.1: POGIL

POGIL è un framework per la progettazione di attività didattiche per l'insegnamento della programmazione. Durante queste attività gli studenti attraversano un ciclo di esplorazione, concettualizzazione e applicazione. Gli studenti scoprono i concetti chiave e costruiscono la propria conoscenza attraverso l'interazione con i compagni e con il docente.

Pro:

⇒ Si ha uno scambio attivo di idee;

⇒ Ci si concentra sulla formazione di un pensiero critico;

⇒ L'insegnante ha il ruolo di "*facilitatore*", non viene visto come fonte di informazioni;

⇒ L'obiettivo è quello di insegnare a imparare e applicare la conoscenza in nuovi contesti;

⇒ Si vuole sviluppare la *metacognizione*, vista come la riflessione sul proprio io e sul processo stesso di apprendimento (di fatto si impara a imparare).

Contro:

⇒ Non può essere applicato efficaciemente in contesti per cui si richiede uso di specifiche nozioni non derivabili dal ragionamento (perchè frutto di convenzioni);

⇒ Alcuni studenti potrebbero avere difficoltà a lavorare in gruppo e forzarli sarebbe solo controproducente;

⇒ Il fatto che gli studenti siano divisi in piccoli gruppi può limitare la circolazione di idee e di conseguenza l'apprendimento. Ciò è parzialmente risolto dalla presenza del docente, ma si ha comunque la possibilità di errore umano per cui alcuni gruppi riceveranno più attenzioni rispetto ad altri;

⇒ Passando a problemi più pratici, un approccio POGIL richiede tempo, materiali e risorse per essere viabile.

5.3 NLD

Definizione 5.3.1: NDL

NLD (Necessity learning design) è un framework per la progettazione di attività didattiche per l'insegnamento della programmazione. Sostanzialmente si dà un problema risolvibile con un certo costrutto senza introdurlo. Successivamente, prima che lo studente si scoraggi, si introduce il costrutto. Si distinguono tre fasi:

- P!S: si dà il problema senza fornire il costrutto non necessario per risolvere l'esercizio;
- I: si introduce il concetto che serve;
- PS: gli studenti applicano il concetto appreso.

Note:-

La fase I è da svolgersi, preferibilmente *unplugged*, perchè gli studenti, tentando di applicare subito il concetto appreso (senza aspettare la fase PS) potrebbero perdersi dei passaggi fondamentali.

Pro:

⇒ È utile quando si devono apprendere concetti a un *differente livello di astrazione* (es. imparare le strutture dati conoscendo già le variabili);

⇒ È utilizzabile per un insegnamento mirato di un singolo concetto.

Contro:

⇒ Non è applicabile in ogni fase dell'apprendimento;

⇒ Necessità di un'adeguata calibrazione per evitare la perdità di interesse da parte degli studenti;

⇒ Alcuni studenti potrebbero sentirsi subito scoraggiati e annoiarsi invece di tentare di risolvere il problema. L'idea parte dal presupposto che lo studente abbia effettivamente il desiderio di risolvere il problema, ma non è sempre così;

⇒ Non si può abusare di questo approccio perchè gli studenti comprenderebbero l'inutilità di impegnarsi nella fase P!S che quasi sicuramente è al di fuori della loro portata.

5.4 Riflessioni sugli approcci

Dopo aver stilato e visionato i pro e i contro di ogni approccio è evidente che *NLD* viene percepito da noi in luce prevalentemente negativa (inoltre lo riteniamo il meno innovativo tra i tre in quanto è un estensione di productive failure), mentre gli altri due approcci presentano un equilibrio. Tra PRIMM e POGIL quello che preferiremmo sperimentare è il primo, per via della sua già citata flessibilità e della sua facile implementazione. *PRIMM* mette il focus sul capire come funziona ogni riga di codice come una sorta di "*Divide et Impera*" e offre agli studenti un "output" sulla loro comprensione dei meccanismi della programmazione. È ottimo per introdurre nuovi concetti, ma allo stesso tempo fornisce allo studente la possibilità di sperimentare personalmente (fase 4 e, soprattutto, fase 5). Riteniamo *POGIL* un buon approccio in linea teorica: purtroppo la metacognizione è un concetto molto poco esplorato nell'ambito dell'istruzione, almeno in Italia. Però questo approccio necessità di apposita formazione e materiale didattico specifico (per guidare gli studenti), il chè può diventare dispendioso. Inoltre non ha quell'adattabilità a molteplici concetti che abbiamo trovato in PRIMM.