Imparare il pensiero computazionale, imparare a programmare

Michael Lodi
Dottore Magistrale in Informatica (Università di Bologna)
Mentor di CoderDojo Bologna
LDOMHL@GMAIL.COM

Il concetto di pensiero computazionale è stato risaltato nel 2006 da Jeannette Wing, che ha mostrato come l'informatica abbia portato alla scienza non solo strumenti (computer e linguaggi di programmazione) ma anche innovazioni nel modo di pensare. Viene proposta una definizione di pensiero computazionale originata da una review della letteratura. Grazie all'utilizzo della Psicologia della Programmazione (studi sulle misconcezioni, che si occupano di individuare i concetti che sono compresi male dai programmatori novizi, e studi sul commonsense computing, che cercano di capire come persone che non hanno mai ricevuto nozioni di programmazione esprimano concetti e processi computazionali), si forniscono una serie di consigli per insegnare al meglio il pensiero computazionale (con i linguaggi attuali, con nuovi linguaggi appositamente progettati, con l'aiuto di strumenti ed ambienti ad-hoc o con una serie di attività estranee alla programmazione) al più ampio pubblico possibile.

1. Introduzione

Il concetto di pensiero computazionale, già presente da tempo ma *trend topic* dell'ultimo decennio nella Didattica dell'Informatica, è stato portato all'attenzione della comunità scientifica da Jeannette Wing in un lungimirante lavoro [Wing, 2006] in cui si evidenzia che, in tutta una serie di ambiti scientifici, l'informatica ha portato non solo strumenti (computer e linguaggi di programmazione) ma innovazioni nel modo di pensare. Un esempio illuminante è il sequenziamento del DNA umano, avvenuto nel 2003, in forte anticipo sulle previsioni, grazie all'adozione di un algoritmo di tipo *shotgun* (basato su metodi *random*) per ricombinare i frammenti di DNA analizzato.

Se nelle discipline scientifiche il processo è già in atto, così non è nella vita di tutti i giorni. Viviamo in un mondo in cui i computer sono pervasivamente attorno a noi (*ubiquitous computing*), ma all'esplosione della disponibilità di calcolatori dalle dimensioni ridottissime, con enorme potenza di calcolo - resa potenzialmente infinita dall'accesso a Internet - e all'enorme disponibilità

ISBN 978-88-98091-31-7

d'informazioni - sempre grazie alla Rete - non ha fatto seguito l'aumento della conoscenza e delle abilità connesse all'elaborazione di tale informazione.

Il pensiero computazionale va ben oltre l'uso della tecnologia, ed è indipendente da essa (sebbene la sfrutti intensivamente): non si tratta di ridurre il pensiero umano, creativo e fantasioso, al mondo "meccanico e ripetitivo" di un calcolatore, bensì di far comprendere all'uomo quali sono le reali possibilità di estensione del proprio intelletto attraverso il calcolatore. Si tratta di *risolvere problemi, progettare sistemi, comprendere il comportamento umano basandosi sui concetti fondamentali dell'informatica*. In sostanza, *pensare come un informatico quando si affronta un problema*.

Riconosciuta la sua importanza, il pensiero computazionale è stato proposto da molti come **quarta abilità di base oltre a leggere, scrivere e calcolare**. Vista la posizione rilevante in cui è posto, in questi anni sono stati proposti curriculum, attività, framework, strumenti tecnici hardware e software per insegnare il pensiero computazionale. Il panorama rimane però ancora molto frammentato.

Negli ultimi tre decenni sono stati svolti numerosi studi di Psicologia della Programmazione. Sono lavori a metà tra l'informatica e la psicologia cognitiva, che riguardano il "modo di pensare" dei programmatori, svolti con lo scopo di valutare e migliorare gli strumenti a loro disposizione basandosi sugli effetti cognitivi di tali tool. Volendo rendere l'informatica accessibile a tutti, di particolare interesse sono gli studi sulle **misconcezioni**, che si occupano di trovare i concetti che sono compresi male dai programmatori novizi e gli studi sul **commonsense computing**, che cercano di capire come persone che non hanno mai ricevuto nozioni di programmazione e, più in generale, di informatica, esprimano (in linguaggio naturale) concetti e processi computazionali.

La tesi di laurea dell'autore [Lodi, 2014] tratta in modo più approfondito e dettagliato questi temi.

2. Definire il pensiero computazionale

La necessità di insegnare il pensiero procedurale è stata riconosciuta da vario tempo [Perlis, 1962; Papert, 1980]. L'articolo di Jeannette Wing ha però portato all'attenzione della comunità scientifica il concetto, producendo un vero e proprio movimento per il pensiero computazionale.

2.1 Definizione

Numerose sono le disquisizioni filosofiche originatesi sulla natura del pensiero computazionale. La necessità pratica di una definizione "operativa", per permettere - nel complesso e altamente burocratizzato sistema dell'educazione pre-uniersitaria - la definizione di obiettivi educativi e la loro valutazione, è sentita da molti. Un'analisi della letteratura [Lodi, 2014] ha portato all'individuazione di alcuni concetti chiave, al loro inserimento in una definizione e all'individuazione di competenze e pratiche che attuano tale definizione.

ISBN 978-88-98091-31-7

2.1.1 Concetti chiave

I concetti ricorrenti nelle definizioni di pensiero computazionale sono elencati di seguito, molti riconosciuti in letteratura [ISTE e CSTA, 2011; Google, 2011; Wing, 2010; …] e altri proposti dall'autore: **- Collezione e analisi dei dati. - Rappresentazione dei dati. - Decomposizione dei problemi. - Astrazione. - Generalizzazione e riconoscimento di pattern. - Algoritmi. - Automazione. - Simulazione, test, debug. - Parallelizzazione. - Complessità e calcolabilità.**

2.1.2 Una definizione operativa

Ispirandoci a [ISTE e CSTA, 2011] possiamo dare la seguente.

Definizione (Pensiero Computazionale). *Il* pensiero computazionale *è un* processo di problem-solving *che consiste nel: - formulare problemi in una forma che ci permetta di usare un computer (nel senso più ampio del termine, ovvero una macchina, un essere umano, o una rete di umani e macchine) per risolverli; - organizzare logicamente e analizzare dati; - rappresentare i dati tramite astrazioni, modelli e simulazioni; - automatizzare la risoluzione dei problemi tramite il pensiero algoritmico; - identificare, analizzare, implementare e testare le possibili soluzioni con un'efficace ed efficiente combinazione di passi e risorse (avendo come obiettivo la ricerca della soluzione migliore secondo tali criteri); - generalizzare il processo di problem-solving e trasferirlo ad un ampio spettro di altri problemi.*

Molte sono state le proposte per l'insegnamento del pensiero computazionale (si veda [Lodi, 2014]), quasi tutte accomunate dalla centralità della programmazione. Prima di proseguire, è bene indagare quindi quali difficoltà s'incontrano nell'apprendimento della programmazione.

3. Imparare a programmare è difficile

Programmare è un'attività centrale nel mondo dell'informatica. La letteratura è concorde sul fatto che, nei corsi introduttivi, gli studenti fatichino ad imparare come farlo, e questo non è limitato a una particolare area geografica, istituzione o linguaggio di programmazione utilizzato [Sorva, 2012]. Analizziamo nei prossimi paragrafi alcuni aspetti, pedagogici e psicologici, che si ritiene siano alla base di tali difficoltà.

3.1 Gli obiettivi cognitivi richiesti sono difficili da raggiungere

Alcuni obiettivi di apprendimento sono più difficili da raggiungere rispetto ad altri (ad esempio è più facile elencare un insieme di keyword relative alla programmazione rispetto a valutare l'efficienza di un algoritmo).

Gli obiettivi di un corso di introduzione all'informatica (tipicamente: scrivere un programma che risolve un problema in linguaggio naturale) si collocano ai livelli più alti (*elevata complessità cognitiva*) delle gerarchie di obiettivi di apprendimento classicamente utilizzate (anche da ACM e IEEE), come quella di Bloom [1956].

ISBN 978-88-98091-31-7

3.2 La memoria di lavoro è limitata

Secondo la teoria dei "magazzini di memoria" [Atkinson e Shiffrin, 1968] la nostra **memoria di lavoro**, l'unica di cui siamo coscienti e che utilizziamo per svolgere ogni attività, ha capacità limitata in spazio e tempo (circa sette elementi, decine di secondi).

In essa usiamo gli **schemi**: strutture mentali che contengono conoscenza concettuale generica (che risiedono nella memoria a lungo termine). Un esperto possiede schemi più generali e astratti, ad esempio "calcolare la media degli elementi di un array", che ora occuperà un solo elemento nella sua memoria di lavoro, mentre un novizio dovrà ricordare molti schemi: per inizializzare le variabili, per scorrere gli elementi di un vettore, per un assegnamento, per calcolare una divisione, per usare una funzione di libreria, ecc. [Soloway, 1986]. Oltre che far posto a più schemi per risolvere problemi più complessi, liberare la memoria di lavoro (ridurre cioè il carico cognitivo) lascia spazio ad *attività di riflessione non strettamente collegate all'obiettivo ma indispensabili per apprendere*.

Secondo la teoria del **carico cognitivo**, cercare di risolvere problemi non è il modo migliore per imparare a farlo, in quanto richiederebbe un carico cognitivo troppo elevato. Risultati sono stati ottenuti invece presentando *esempi risolti o semirisolti e commentati* [Linn e Clancy, 1992].

3.3 Si possono formare misconcezioni

Oltre agli schemi, in memoria di lavoro sono presenti i **modelli mentali**: riguardano sistemi e oggetti con cui entriamo in contatto, servono per prevedere come tali aspetti dell'ambiente si comporteranno. Sono *eseguibili*: possiamo fare simulazioni mentali su di essi.

I modelli mentali si formano sulla base degli aspetti visibili di un sistema e la loro formazione è inevitabile [Ben-Ari, 2001], pertanto è bene insegnare esplicitamente i modelli che si ritengono corretti. Nella programmazione l'unico aspetto visibile è la *sintassi*. Questa però, specialmente ad alto livello, nasconde molti dettagli del modello di esecutore sottostante.

Quando si comprende male qualcosa, si formano schemi e modelli inadeguati: si parla di **misconcezioni**. Le misconcezioni nella programmazione [Clancy, 2004] sono dovute in particolare a: *sintassi e notazione* (es. assegnamento visto come swap o incremento come equazione impossibile), *lingua inglese* (parole come *while*, *and*, *if* con significati multipli in inglese), *attribuzione di intenzionalità* (computer con intelligenza nascosta che comprende le intenzioni del programmatore, computer con visione generale sull'intero codice), *sovrageneralizzazione di esempi* (non ho mai visto funzioni con variabili non numeriche, teorizzo che le funzioni prendano in input solo numeri).

4. Imparare a programmare non è sempre intuitivo

Avendo lo scopo di estendere le capacità programmazione al più ampio pubblico possibile, è naturale chiedersi quale sia **il modo naturale di**

esprimere una computazione e quali sono le **concezioni che i soggetti posseggono** prima di approcciarsi alla programmazione.

4.1 Il modo naturale di esprimere le istruzioni

Alcuni studi [Pane et al, 2001; Good et al, 2010] su bambini e adulti che non hanno mai programmato mostrano quale sia il modo naturale con cui essi esprimono processi computazionali (chiedendo loro di risolvere esercizi di programmazione in linguaggio naturale). **Stile di programmazione**: le istruzioni sono espresse con regole di produzione o "event based" e iniziano con *if* o *when*; si notano soluzioni più vicine ai sistemi reattivi, con scarsa attenzione per il flusso di controllo globale; lo stile imperativo è utilizzato per descrivere il flusso di controllo locale, mentre quello dichiarativo per "fare il setup dello scenario". **Prospettiva**: si tende ad assumere quella dell'utente o di una terza persona; quasi mai quella del programma. **Operazioni su oggetti multipli e cicli**: invece di iterare su una collezione, si usano insiemi e sottoinsiemi (es. *foreach*), oppure esprimono le istruzioni al plurale ("aggiorna gli elementi"); questo elimina molte situazioni in cui si usano i cicli; le rare volte in cui sono usati, assumono la forma *repeat…until*. **Condizionali complessi e not**: invece di usare condizionali complessi, si usano una serie di regole semplici mutuamente esclusive o nella forma "se A allora fai qualcosa a meno che B". **Operazioni matematiche**: i bambini esprimono le operazioni matematiche in linguaggio naturale; solo pochi adulti usano una notazione matematica (i+3), nessuno quella informatica (i=i+3); alcune espressioni matematiche non vedono esplicitato su quale valore operare o l'ammontare dell'operazione. **Ricordare lo stato**: non sono usate variabili per ricordarsi il progresso in un task, piuttosto espressioni quali "quando tutti / quando nessuno... allora il task è finito"; se si vuole usare un'informazione precedente per una decisione attuale, o un'informazione attuale per una decisione futura, si parla al passato o al futuro per riferirsi alle informazioni necessarie. **And, or, but**: la parola *and* viene usata come operatore di sequenza piuttosto che booleano, a volte anche quando servirebbe or ("90 and above"); *or e but* appaiono poco. **Then ed else**. Il termine *then* viene usato per dire "dopo", quindi come operatore di sequenza, e non per dire "allora"; la clausola *else* non è mai presente. **Operazioni di inserimento, cancellazione, ordinamento**: non si considerano problemi di memorizzazione degli array, evidenziando un modello di tipo "lista"; l'ordinamento è considerato operazione elementare già disponibile. **Object oriented**: sono usati alcuni aspetti dell'O.O. (entità aventi uno stato e rispondenti a richieste di azioni), altri no (ereditarietà e polimorfismo). **Uso di disegni**: si fa largo uso di diagrammi per esprimere le soluzioni. **Errori**: nonostante l'assenza dei problemi legati alla sintassi, solo una piccola parte delle risposte è completa, corretta e non ambigua; gli errori di "omissione" (parti di regole necessarie non esplicitate) sono il triplo degli errori "commessi" (parti di regole presenti ma sbagliate); le regole sono accurate ma non complete.

4.2 Commonsense computing

Una serie di studi [Simon et al, 2006 e seguenti], raggruppati sotto il titolo emblematico di **Commonsense Computing**, sono stati svolti per investigare quali concetti legati alla programmazione o all'informatica in generale, siano posseduti dagli studenti prima di ricevere un'istruzione formale. Gli studi riguardano studenti all'inizio del loro primo anno di Università (iscritti a Informatica o ad altri corsi quali Economia). - I partecipanti hanno evidenziato buone capacità di problem solving, che paradossalmente peggiorano dopo alcune settimane di studio di un linguaggio di programmazione. - Il modello di esecutore non è coerente con quello normalmente insegnato (es. si pensa ai numeri naturali come memorizzati sotto forma di stringhe di caratteri). - Alcune operazioni come l'ordinamento sono considerate primitive. - I partecipanti sono stati capaci di riconoscere un problema di concorrenza, fornendo in molti casi una soluzione ragionevole al problema posto. - I partecipanti usano tecniche di debug ingenue (es. ripara e testa, riprovare più volte la stessa cosa) ma non le tecniche più avanzate e utili (testare per acquisire informazioni, provare più di una soluzione, avere un'operazione di fallback, uso dettagliato della conoscenza di dominio per ottenere informazioni dettagliate, annullare un tentativo di riparazione fallita). - I partecipanti non usano la metrica del "caso pessimo" per valutare l'efficienza di un algoritmo. - Chiedere di fare tracing o di usare esempi ha migliorato i risultati nell'individuazione del migliore tra due algoritmi (persino quando il tracing era errato). - C'è stata difficoltà ad astrarre da dati inutili presenti nei problemi. - Gli studenti non hanno compreso una regola logica espressa nella forma "if and only if", ma hanno compreso la stessa parafrasata nella forma "either both … and… or neither … nor…".

5. Spianare la strada al pensiero computazionale

Il pensiero computazionale ha una forte componente linguistica e di programmazione: si basa sul pensiero algoritmico e il suo obiettivo principale è la descrizione procedimenti effettivi per la risoluzione dei problemi.

Almeno attualmente, il modo più diffuso, studiato e maturo per favorirne l'acquisizione è, banalmente, insegnare a programmare. Chi impara a programmare bene infatti sa analizzare un problema, scomporlo nelle sue parti essenziali, astrarre, automatizzarlo, testarlo e correggere errori, riutilizzare codice e ottimizzarlo. Non è però pensabile prendere semplicemente un corso universitario di programmazione e mapparlo sui gradi più bassi d'istruzione, per ragioni facilmente intuibili (età, interesse, difficoltà riscontrate dai novizi, necessità di una visione più ampia, scopo generalista e non specialistico).

Nel breve periodo, proponiamo di seguire l'appello di Guzdial [2008], che chiede di utilizzare le scoperte fatte dalla ricerca in campo educativo e informatico per rendere lo studio dei linguaggi di programmazione e la comprensione del concetto di computazione più facile e accessibile a tutti.

Contemporaneamente però la ricerca deve chiedersi quali siano le caratteristiche intrinseche del pensiero computazionale indipendenti dallo strumento "linguaggio di programmazione", quali siano le età e le metodologie giuste per introdurlo.

ISBN 978-88-98091-31-7

Si forniscono di seguito alcuni consigli, sulla base degli studi esposti.

5.1 Insegnare meglio a programmare

Conoscendo quali aspetti sono più difficili da imparare, fonti di misconcezioni e controintuitivi, bisogna porre particolare attenzione su di essi, spiegandoli e rimarcandoli esplicitamente.

Bisogna favorire la creazione di schemi: partire da soluzioni di problemi semplici e ricorrenti (es. “scorrere tutti gli elementi di un array”), dapprima presentati come esercizi da risolvere ed esplicitando poi che si tratta di un pattern da utilizzare in seguito quando viene riconosciuto.

Così come ci viene insegnato a leggere prima che a scrivere, così dovrebbe essere nella programmazione (l’autore ha assistito a vari novizi - bambini o universitari - bloccati su un bug “introvabile”, illuminarsi alla richiesta di “spiegare cosa fa il programma, riga per riga”). Vanno progettati esercizi che impongano di fare tracing (tecnica utilissima nel debug, ma scarsamente utilizzata dagli studenti), utile anche per insegnare agli studenti a ragionare dal punto di vista del programma - cosa che non fanno intuitivamente. Va favorito l’uso di debugger e vanno spiegati esplicitamente aspetti non intuitivi di debug (inutilità di una ricompilazione sullo stesso codice, disfare ciò che si è fatto per ritornare a uno stato noto, capire qual è l’errore prima di correggerlo, possibilità di cause molteplici per uno stesso errore).

Fare tracing significa “eseguire un modello mentale”. L'importanza dei modelli è evidenziata dalle numerose misconcezioni adducibili a un modello “errato” e ai vantaggi che invece porta conoscerlo esplicitamente. Seguendo i consigli di Ben-Ari [2001], si sottolinea l'importanza di insegnare esplicitamente un modello concettuale di una macchina astratta ai novizi, di modo che essi si formino un modello mentale adeguato all'apprendimento della programmazione.

5.2 Rendere più naturali i costrutti

Alcuni [Pane et al., 2001] suggeriscono di pensare alla programmazione come “il processo di trasformazione un piano mentale espresso in termini familiari in uno comprensibile dal computer”. Più il linguaggio di programmazione è vicino a questo piano mentale, più sarà facile il processo di trasformazione del piano in un programma per calcolatore. Andare verso una programmazione in linguaggio naturale farebbe riemergere alcune misconcezioni (inglese, intenzionalità), ma rendere i linguaggi più naturali vorrebbe di certo dire facilitarne la comprensione. Sulla base degli studi citati, emergono alcuni consigli per la progettazione dei nuovi linguaggi.

L'iterazione su collezioni dovrebbe essere più astratta, con costrutti come il *foreach* (*for* su oggetti *Iterable* in Java) oppure operazioni ad alto livello, quali mapping e filtri. È meglio avere e utilizzare cicli con test posticipato della condizione e con le parole chiave *repeat..until*, salvo poi introdurre il *while* (parola chiave non intuitiva, da riconsiderare), per mostrare come sia più generale. Molto naturale è la scrittura di strutture del tipo *when* <*condizione*>, *while* <*condizione*>, *if* <*condizione*>, tutte pensate come “aspetta che la condizione sia vera e poi fai qualcosa” (istruzione bloccante) o “non appena la

condizione è falsa smetti di fare qualcosa" (controllo continuo). La parola *then* viene intesa come costrutto di sequenza ("dopo"), da evitare quindi nell'*if*. Il ramo *else* è poco utilizzato perché si considera implicito ciò che accade quando la condizione è falsa; il programma però non può dedurre dal contesto cosa fare in tali casi, dunque va pensata una struttura che ne forzi l'uso. Le variabili sono fonti di numerose misconcezioni: meglio evitare il simbolo = per esprimere l'assegnamento (e in generale simbologie che possono essere confuse con quelle altri ambiti). Molti errori e omissioni sono dovuti alla difficoltà di esprimere condizionali complessi. C'è la tendenza a mettere il caso generale all'inizio e poi elencare le eccezioni alla fine, in positivo, con la parole chiave *unless*; utile dunque un costrutto in stile *try...catch*. Molti errori sono generati dalle parole chiave *and*, *or*, *not*, ambigue nel linguaggio naturale; vanno pensate sintassi alternative o metodi visuali, quali tabelle o colonne in cui elencare le condizioni da soddisfare e quelle da non soddisfare (spesso infatti i condizionali complessi vengono spezzati in un elenco di condizioni - a volte mutamente esclusive). Come già visto, essendo gli errori soprattutto di omissione, sarebbe utile fornire template che richiedono il completamento.

Discussione complessa e con radici antiche è quella sul paradigma da adottare. I linguaggi funzionali e dichiarativi sono più astratti e dunque più vicini alla rappresentazione mentale del programma; esprimono più il "cosa fare" piuttosto che il "come farlo" e potrebbero indebolire l'apprendimento del pensiero computazionale, che si basa proprio sulla necessità di esprimere procedimenti algoritmici. L'orientamento agli oggetti introduce una quantità non indifferente di concetti aggiuntivi da apprendere (classi, oggetti, puntatori, ereditarietà e polimorfismo, allocazione e deallocazione implicita della memoria...), portandosi poi dietro tutte le misconcezioni già presenti nel paradigma imperativo. Gli studi mostrano come il paradigma sia intuitivo solo in parte: se è vero che sono stati riscontrati alcuni aspetti del paradigma ad oggetti, è altresì vero che raramente questo era visto dalla prospettiva dell'entità stessa. Nessuna evidenza invece ha mostrato i partecipanti usare categorie di entità (classi), ereditarietà o polimorfismo.

Il mix di stili utilizzati in diverse situazioni suggerisce di non limitare il linguaggio a un singolo paradigma, ma lasciare la libertà di usarne di più (un esempio commerciale: The Wolfram Language, implementato in Mathematica).

5.3 Usare strumenti di visualizzazione

La visualizzazione degli algoritmi e dei programmi non è nuova in ambito informatico (vedi: diagrammi di flusso). Per il pensiero computazionale si propone l'uso di alcuni strumenti. **Linguaggi di programmazione visuale**: permettono di creare i programmi manipolando elementi grafici; la notazione può essere costituita da diagrammi o da "blocchi" da unire tra loro, eliminando le difficoltà legate alla sintassi. **Animazione dei programmi**: forma di visualizzazione in cui l'ambiente mostra esplicitamente alcuni aspetti dell'esecuzione di un programma (es. stato delle variabili, stack delle chiamate, oggetti, …), rendendo esplicito il modello dell'esecutore e riducendo così il carico cognitivo. **Simulazione visuale di programmi**: allo studente si chiede di leggere il codice, comprendere come procederà il flusso di controllo ed eseguire

un tracing modificando correttamente gli elementi dello stato di esecuzione che sono messi in evidenza. Questo è coerente con la necessità di mettersi nei panni del programma e aiuta a superare le misconcezioni.

5.4 Usare un ambiente e una metodologia specifici

Da quanto visto finora, elenchiamo alcune caratteristiche che dovrebbe avere un buon linguaggio adatto all'insegnamento del pensiero computazionale. Parliamo più correttamente di **ambiente di sviluppo e collaborazione**, in quanto il linguaggio è, in questo contesto, molto legato all'ambiente attraverso il quale si interagisce con esso. Abbiamo in particolare in mente un linguaggio che: sia *multiparadigma*, cioè supporti parti imperative, regole di produzione, agenti ed eventi; sia *visuale o organizzato per template*; preveda una forma *grafica* di rappresentazione del codice o suggerimenti su cosa rappresentino in un dato momento variabili, parametri, funzioni; preveda una forma di *rappresentazione grafica dello stato della macchina* a *compile time* e a *runtime*; preveda strumenti di tracing e debug completi e facili; sia eventualmente inserito in una piattaforma di collaborazione (in cui sia possibile leggere il codice prodotto da altri, modificarlo in copia, o collaborare su uno stesso progetto); preveda l'accesso selettivo a funzionalità o la loro complicazione a seconda dell'età o del livello dell'utente; preveda incentivi per coinvolgere in modo attivo chi apprende.

Un ambiente con le caratteristiche proposte permette l'uso di una metodologia per favorire il coinvolgimento dei ragazzi nel pensiero computazionale: è la cosiddetta **progressione usa-modifica-crea** (*use-modify-create*, es. in [Lee et al, 2011]). Questa metodologia prevede di far interagire lo studente in modo crescente con l'ambiente di sviluppo. Nella fase **usa**, lo studente è consumatore delle creazioni di altri (serve per imparare ad utilizzare gli strumenti, prendere confidenza con l'ambiente ed essere coinvolti). Nella fase **modifica**, lo studente vorrà modificare il programma, scrivendo nuovi pezzi di codice, per i quali avrà bisogno di comprenderne i meccanismi. Nella fase **crea** lo studente verrà incoraggiato a realizzare un proprio progetto, applicando le conoscenze e le competenze apprese. Questo approccio favorisce il *coinvolgimento* dello studente in quanto gli propone sfide la cui difficoltà cresce man mano che crescono le sue abilità, senza dunque l'ansia iniziale di un compito troppo difficile ma senza annoiarlo una volta che sarà diventato più esperto [Lee et al, 2011]. Quest'approccio è coerente con le soluzioni proposte per ridurre il carico cognitivo (dà la possibilità di studiare "esercizi" risolti o di modificarne o completarne di parzialmente risolti, risolvendo il problema della noiosità degli stessi) e con i consigli per favorire la creazione di schemi (poiché spinge a leggere e comprendere il codice per poterlo modificare). Questo approccio inoltre sposa la filosofia open source, che è ritenuta essere un buon esempio di *collaborazione in un contesto autentico* e di *situated learning*, concetti molto cari ai sociocostruttivisti [Sorva, 2012].

5.5 Insegnare il pensiero computazionale senza la programmazione

Lu e Fletcher [2009] evidenziano come sarebbe necessario porre le basi del pensiero computazionale molto tempo prima che gli studenti imparino a programmare. Così come le dimostrazioni in Matematica e la letteratura in Italiano vengono introdotte solo dopo anni in cui si insegna a leggere, scrivere, calcolare, così dovrebbe essere per la programmazione. L'enfasi dovrebbe andare sulla comprensione e l'abilità di eseguire manualmente processi computazionali, acquisendo familiarità con il flusso del controllo, imparando ad astrarre e a rappresentare informazioni, a valutare proprietà di processi.

Le attività per insegnare questo tipo di conoscenze e competenze sono accomunate dal fatto cruciale che in esse **gli studenti sono gli agenti computazionali** e sono in linea con gli studi (assunzione del punto di vista del computer, forma di tracing in prima persona, comprensione dei limiti della macchina) e affrontano il problema del *transfer*: per far sì che gli studenti imparino ad usare il pensiero computazionale in diversi contesti, questo va insegnato in tutti quei contesti, identificandolo ed esplicitandolo ogni volta che si usa. Molte attività sono state proposte in questo senso, con diverse modalità (si veda [Lodi, 2014]).

La prima modalità consiste nel **mettere in evidenza** gli aspetti del pensiero computazionale quando si utilizzano nelle altre discipline. Introducendo la moltiplicazione, si può far notare come essa sia una somma ripetuta (iterazione) e come, sebbene valga la proprietà commutativa, in termini di efficienza è meglio sommare tre volte sei (6x3) piuttosto che sei volte tre (3x6).

Possono poi essere progettate **attività multidisciplinari** con obiettivi specifici nelle varie materie (es. ricerca su un periodo storico) che possono o meno comprendere aspetti del pensiero computazionale (es. identificare gli eventi chiave - astrazione - e proporre un algoritmo che descrive come la storia avrebbe potuto evolversi se gli eventi fossero andati diversamente - facendo quindi largo uso del costrutto di selezione).

Si possono sfruttare i **lavori di gruppo** per favorire una *meta riflessione sul parallelismo*. Le interazioni per lo scambio di dati sono ottimi momenti per introdurre il concetto *d'interfaccia*. Scrivere il report in modo collaborativo può permettere di sperimentare i problemi e le strategie della *comunicazione sincrona e asincrona*, del *locking* e del *message passing*.

Si possono infine progettare attività esplicitamente pensate per far comprendere ai ragazzi i concetti informatici, senza però usare il computer, secondo l'esempio guida di "Computer Science Unplugged". In molte declinazioni diverse, un'attività particolarmente suggerita consiste nel far eseguire istruzioni (es. camminare da un punto ad un altro, disegnare, …) ad uno studente sulla base delle indicazioni del compagno. L'esecutore dovrà attenersi esattamente a quanto gli è stato detto ed essere limitato ad un set di istruzioni predeterminate, il programmatore dovrà essere il più preciso possibile. Le istruzioni potranno essere date dal vivo (*interpretazione*) o prima scritte e poi eseguite (*compilazione*).

Escludere completamente la programmazione ha comunque i suoi limiti. Il problema del *transfer* può essere infatti posto anche nell'altra direzione: se non

si applicano mai i principi del pensiero computazionale alla programmazione, poi non sarà automatico il loro riconoscimento e utilizzo in essa.

6. Conclusioni

Il pensiero computazionale è riconosciuto come un concetto centrale da imparare per vivere nella nostra società e lavorare con profitto nelle scienze (matematiche, naturali, umanistiche e sociali). Insegnarlo programmando può essere una strada, a patto che si tenga conto delle difficoltà del programmare e si sfruttino le scoperte in ambito psicologico ed educativo per superare queste difficoltà. Bisogna poi che la ricerca informatica si coordini con quella psico-pedagogica per capire quali aspetti intrinseci del pensiero computazionale sono necessari per tutti, e quali siano i tempi, i metodi, i concetti e le tecnologie migliori per insegnarlo.

Bibliografia

[Atkinson e Shiffrin, 1968] Atkinson, R. C., & Shiffrin, R. M., Human memory: A proposed system and its control processes. The psychology of learning and motivation, 2, 1968, 89-195.

[Ben-Ari, 2001] Ben-Ari, M., Constructivism in computer science education. Journal of Computers in Mathematics and Science Teaching, 20, 1, 2001, 45-73.

[Bloom 1956] Bloom, B. S., Taxonomy of educational objectives. Handbook I: Cognitive Domain. David McKay Company, New York, 1956.

[Clancy, 2004] Clancy, M., Misconceptions and attitudes that interfere with learning to program. Computer science education research, 2004, 85-100.

[Good et al, 2010] Good, J., Howland, K., & Nicholson, K., Young people's descriptions of computational rules in role-playing games: an empirical study. IEEE Symp. on VL/HCC, 2010, 67-74.

[Google, 2011] Google, Exploring Computational Thinking. http://www.google.com/edu/computational-thinking/.

[Guzdial, 2008] Guzdial, M., Education Paving the way for computational thinking. Comm. ACM, 51,8, 2008, 25-27.

[Lee et al, 2011] Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J. & Werner, L.,Computational thinking for youth in practice. ACM Inroads, 2, 1, 2011, 32-37.

[Linn e Clancy, 1992] Linn, M. C., & Clancy, M. J., The case for case studies of programming problems. Comm. ACM, 35, 3, 1992, 121-132.

[Lodi, 2014] Imparare il pensiero computazionale, imparare a programmare. Tesi di Laurea Magistrale in Informatica, Universtià di Bologna, Marzo 2014. http://amslaurea.unibo.it/6730/

[Lu e Fletcher, 2009] Lu, J. J., & Fletcher, G. H., Thinking about computational thinking. In ACM SIGCSE Bull., 41, 1, 2009, 260-264).

[Pane et al, 2001] Pane, J. F., Ratanamahatana, C., & Myers, B. A., Studying the language and structure in non-programmers' solutions to programming problems. Int. J. Hum.-Comput. Stud., 54, 2, 2001, 237-264.

[Papert, 1980] Papert, S., Mindstorms: Children, computers, and powerful ideas. Basic Books, 1980.

[Perlis, 1962] Perlis, A., The computer in the university. Computers and the World of the Future, MIT Press, 1962, 180-219.

[Simon et al, 2006] Simon, B., Chen, T. Y., Lewandowski, G., McCartney, R., & Sanders, K., Commonsense computing: what students know before we teach (episode 1: sorting). In proc. ICER '06, ACM, New York, 2006, 29-40.

[Soloway, 1986] Soloway, E., Learning to program = learning to construct mechanisms and explanations. Comm. ACM, 29, 9, 1986, 850-858.

[Sorva, 2012] Sorva, J., Visual program simulation in introductory programming education. Ph.D. thesis, Aalto Univ., 2012.

[Wing, 2006] Wing, J. M., Computational thinking. Comm. ACM, 49, 3, 2006, 33-35.

[Wing, 2010] Wing, J. M., Computational thinking: What and Why? Link Magazine, 2010.