DAL MESTIERE DELL'INFORMATICO AL PENSIERO COMPUTAZIONALE

FROM DIGITAL COMPUTING TO COMPUTATIONAL THINKING

Giorgio Olimpo
Istituto per le Tecnologie Didattiche, Consiglio Nazionale delle Ricerche, Genova, Italy, olimpo@itd.cnr.it

How to cite: Olimpo, G. (2017). Dal mestiere dell'informatico al pensiero computazionale. *Italian Journal of Educational Technology, 25(2),* X-X. doi:10.17471/2499-4324/918

Sommario Nel seguito viene proposta una breve analisi delle fasi in cui si articola lo sviluppo del software, dei metodi comunemente utilizzati dagli informatici in ciascuna di queste fasi e del possibile significato che queste modalità di pensiero possono assumere nei processi educativi. L'obiettivo è quello di offrire un quadro delle potenzialità educative del pensiero computazionale più ampio ed approfondito di quello che, in molti casi, è il modo di pensare corrente. Il pensiero computazionale si configura così come una abilità di portata generale che ha a che fare con l'astrazione, la rappresentazione, l'espressione e la comunicazione, e la capacità di indagine; e non viene confinato nella sola dimensione, importante ma restrittiva, della codifica in uno specifico linguaggio di programmazione.

PAROLE CHIAVE Pensiero computazionale, Sviluppo del software, Abilità cognitive, Codifica.

Abstract This paper proposes a short analysis of the main phases of software development, the approaches and conceptual tools that professionals typically adopt to tackle the complexity of those phases, and the possible meanings that this way of thinking may assume in educational processes. The main purpose is to offer a point of view on the educational potential of computational thinking that is deeper and more comprehensive than the coding oriented attitude widely assumed by stakeholders. The perspective on computational thinking that is provided involves cognitive skills of general value such as abstraction, representation, expression and communication and inquiry, and is not confined to the important but restrictive area of coding.

KEYWORDS Computational thinking, Software development, Cognitive skills, Coding.

1. INTRODUZIONE

«*Il pensiero computazionale rappresenta un atteggiamento ed un complesso di abilità che sono universalmente applicabili e che chiunque, non soltanto gli informatici, dovrebbe essere desideroso di apprendere e di utilizzare*». Questa famosa affermazione di Wing (2006) è stata ripresa da molti ed è diventata uno dei pilastri motivazionali per l'introduzione del pensiero computazionale nell'educazione. La Wing è un'informatica *di rango* (all'epoca direttore del Dipartimento di Computer Science alla Carnagie Mellon University) che certamente ha ben interiorizzato il modo di pensare degli informatici. Proprio per questo la definizione di pensiero computazionale da lei proposta non è strettamente focalizzata sulla programmazione dei computer. Particolarmente significative sono alcune sue affermazioni sul pensiero computazionale.

- Si riferisce a concettualizzare e non a programmare.
- Pensare come un informatico significa molto più che esser capaci a programmare il computer e richiede soprattutto di saper pensare a livelli multipli di astrazione.
- Si riferisce ad abilità fondamentali, non a capacità meccaniche di basso livello.
- Non è un tentativo di fare in modo che gli uomini pensino come i computer; i computer sono inintelligenti e noiosi mentre gli uomini hanno intelligenza e fantasia.

Nonostante queste posizioni di partenza, in una prima fase della ancor breve storia del pensiero computazionale, l'enfasi sulla programmazione intesa come capacità tecnica è diventata molto forte se non prevalente (Manilla, 2014). Significativa è anche la terminologia adottata: il termine *coding*, diventato così popolare, mette in modo deciso l'accento sulla fase finale del processo di sviluppo del software, la scrittura del codice in un dato linguaggio di programmazione. Oggi tuttavia sta sempre più maturando la consapevolezza che il pensiero computazionale «*…coinvolge un ambito molto più ampio della sola programmazione. Per esempio i processi di analisi e di decomposizione dei problemi precedono l'attività di codifica vera e propria*» (Bocconi, Chioccariello, Dettori, Ferrari, & Engelhardt, 2016). Alcuni autori poi sostengono come il pensiero computazionale includa l'esercizio di atteggiamenti e abilità cognitive molto lontane dalle tecnicalità del coding, per esempio la capacità di gestire la complessità o di affrontare problemi aperti o non ancora formalizzati (Barr, Harrison & Conery, 2011; Weintrop et al., 2015).

Nel seguito di questa nota si cercherà di offrire una prospettiva di concretezza su alcune delle possibili componenti concettuali del pensiero computazionale descrivendo il percorso seguito dagli informatici impegnati nello sviluppo di software ed evidenziando gli approcci adottati, le abilità messe in gioco e gli atteggiamenti richiesti. Quando possibile, si cercherà di suggerire il significato che queste modalità di pensiero possono assumere nei processi educativi. Si parlerà sia di significati che sono già stati portati

nella pratica educativa o che sono stati sperimentati in ambito di ricerca, sia di ipotesi plausibili, ma ancora da coniugare con i processi di apprendimento.

Il punto di vista adottato è che l'educazione, per quanto riguarda il pensiero computazionale, non dovrebbe perseguire soltanto una finalità di preparazione all'esercizio di una professione come ancora oggi emerge in modo più o meno esplicito dalle dichiarazioni o dai documenti di natura politica. Per es. in (European Commission, 2015) si legge che «l'acquisizione di competenze digitali, incluso il coding, è considerata essenziale per sostenere lo sviluppo e la competitività»**.** Analogamente in (European Commission, 2016) si mette il fuoco sulla necessità di sviluppare competenze digitali per il mercato del lavoro. L'educazione dovrebbe invece soffermarsi su capacità generali che hanno a che fare con l'analisi di *situazioni problematiche*, l'identificazione e la *rappresentazione* dei problemi e delle relative soluzioni, la *comunicazione* e la condivisione in relazione al problema e ai suoi possibili processi risolutivi, tutte cose che sono certamente rilevanti nell'esercizio delle professioni, ma che hanno anche la dimensione culturale di strumenti per rapportarsi con una società in continua trasformazione, interpretarne i mutamenti, condividere le idee.

2. QUAL È IL PROBLEMA

Solo raramente un informatico conosce a priori il problema che dovrà affrontare e che dovrà risolvere attraverso la costruzione di un programma. Quindi, all'inizio, il problema non c'è ancora, o meglio, c'è, ma non si sa ancora bene quale sia. Il primo passo richiede quindi di affrontare lo studio del contesto in cui il problema si manifesta e le specifiche esigenze dell'utente, o degli utenti, che cercano una soluzione a quel problema. Spesso l'utente percepisce le proprie esigenze, ma non è sempre in grado di tradurle in un problema ben formalizzato. Tocca all'informatico far emergere i requisiti del software da sviluppare – arrivare cioè a una prima formalizzazione del problema – a partire da una situazione spesso complessa, in cui almeno una parte dei saperi e delle variabili di contesto rilevanti sono inizialmente impliciti. La comprensione e la definizione del problema non è quindi per l'informatico il punto di partenza, ma un primo punto di arrivo.

Questo percorso contrasta con una pratica del problem solving ancora fortemente presente nei nostri sistemi educativi. Spesso i problemi che sono proposti agli allievi sono già ben formalizzati, sterilizzati da tutte le complessità, i saperi impliciti e le ambiguità di un contesto che, di solito, non fa neppure parte dell'enunciato del problema. Si perde così una parte del problem solving molto preziosa e ricca dal punto di vista cognitivo: il percorso che porta da una situazione problematica all'identificazione di uno specifico problema sufficientemente formalizzato e tale da poter essere risolto. D'Amore e Fandiño Pinilla (2006) definiscono «*un vero e proprio problema scolastico quello in cui tutti gli ingredienti sono al posto giusto … dati numerici, situazione fittizia ma comprensibile*

ed immaginabile, un quasi - suggerimento semantico delle operazioni necessarie…». In questo caso il modo di pensare degli informatici può suggerire all'educazione un tipo di percorso in cui, quando è opportuno e possibile, si parte da una esigenza o da un obiettivo per arrivare, attraverso una fase di esplorazione e di analisi, ad un problema specifico a cui si deve trovare una soluzione. Ai docenti è richiesto un vero e proprio percorso di invenzione didattica che deve naturalmente tener conto dell'età, della capacità di astrazione raggiunta dagli allievi e dello specifico ambito contenutistico/disciplinare in cui ci si muove. È importante osservare che il modo di procedere che gli informatici utilizzano per "capire qual è il problema" è applicabile in una molteplicità di situazioni anche quando l'obiettivo finale non è la costruzione di un programma. In altri termini "capire qual è il problema" fa parte del modo di pensare dell'informatico, ma non implica necessariamente il rapporto con una attività di *programmazione*.

3. COSA DEVE FARE UN PROGRAMMA?

Una volta capito qual è il problema, l'informatico deve descrivere il comportamento di un programma capace di risolverlo. Si parla di definizione delle *specifiche* del programma (come il programma deve comportarsi) o anche della sua *architettura funzionale*, cioè il complesso di funzioni che il programma deve svolgere insieme con le loro reciproche interrelazioni. Mentre la fase precedente ha soprattutto a che fare con la domanda "*perché* si deve realizzare un certo programma", questa seconda fase si riferisce alla domanda "*cosa* deve fare quel programma". Come vedremo successivamente, la terza domanda a cui rispondere sarà "*come* si deve costruire il programma" perché svolga correttamente le funzioni previste dalle specifiche. Il *perché* prima del *cosa* e il *cosa* prima del *come*. Va detto chiaramente che non si tratta di una regola rigida e implacabile. Ci sono casi in cui, per definire il comportamento di un programma (cosa) si deve tener conto per esempio degli ambienti software disponibili o di parti di software esistenti da riutilizzare, tutte cose che rientrano nell'ambito del come. Così come per definire i bisogni dell'utente e i requisiti di un programma (*perché*) si dovrà, per esempio, tener conto di architetture già esistenti. Nonostante questi tre momenti non siano sempre del tutto indipendenti fra loro, è comunque importante disporre della capacità di distinguere chiaramente i tre livelli del perché, del cosa e del come. Importante per l'informatico impegnato nello sviluppo di software, ma anche importante in ambito educativo in quanto fattore strategico per leggere la realtà e per operare su di essa.

L'informatico che deve progettare l'architettura funzionale di un programma sa bene che non esiste una sola soluzione, ce ne sono molte, di più o meno efficienti, di più o meno resistenti al cambiamento, di più o meno eleganti. Spesso si parte con un impianto iniziale e poi si torna indietro quando ci si imbatte in un ostacolo imprevisto o ci si rende contro che si può fare di meglio. È il cosiddetto

bactracking (= tornare sui propri passi) molto praticato dagli informatici sia nel percorso progettuale, che non è quasi mai del tutto lineare, sia come tecnica algoritmica all'interno dei loro programmi. Anche se qualcosa sta iniziando a cambiare, nei nostri sistemi educativi è ancora presente una visione secondo la quale i problemi vengono spesso proposti come qualcosa che ha una soluzione unica ed un unico percorso risolutivo. Già verso la metà del secolo scorso era stata evidenziata l'importanza educativa di adottare un atteggiamento differente: «*…dovreste adottare un modo di pensare flessibile e chiedervi se c'è un altro modo più semplice di risolvere il problema. Di solito esiste più di un percorso risolutivo corretto*» (Polya, 1945). Uno dei valori impliciti nel prendere in considerazione soluzioni diverse per metodo e/o per risultato è quello di attivare nelle classi un processo di discussione-argomentazione sulla natura e sulle caratteristiche delle diverse soluzioni individuate favorendo così lo sviluppo del pensiero critico (Halpern, 2003).Tuttavia nella pratica didattica, questo non è sempre agevole da mettere in atto sia per vincoli temporali legati al curriculum, sia per le competenze di cui i docenti devono disporre. Integrare nel curriculum uno spazio dedicato al pensiero computazionale, come è avvenuto o sta avvenendo in molti paesi (Bocconi et al., 2016), può rappresentare una specie di *cavallo di Troia* per espugnare la visione unicistica della soluzione dei problemi, ed aprire di conseguenza l'educazione ad un processo di problem solving più vicino alla realtà, più motivante per gli allievi e più ricco di significati educativi. Per quanto riguarda le competenze docenti, D'Amore (2014) afferma «*Ci vuole coraggio, ci vuole prontezza, ci vuole professionalità per accettare soluzioni inattese…Se tu hai già in mente una risposta, farai molta fatica a metterti nei panni altrui per capire quel che ha in mente l'altro. Ma è il confronto fra la tua soluzione e le proposte degli altri a creare sviluppo cognitivo vero, a creare professionalità, come avviene nella ricerca scientifica quando più scienziati si confrontano* ...». Si tratta quindi di mettere in atto un percorso di formazione dei docenti non finalizzato alla semplice acquisizione di competenze teoriche. In Bocconi et al. (2016) si afferma: «esiste un ampio consenso fra gli esperti e gli operatori della formazione sul fatto che l'introduzione del pensiero computazionale nei curricula scolastici sta creando una forte domanda di sviluppo professionale in servizio su larga scala. Le attività di aggiornamento sono spesso progettate per essere *hands-on* in modo che gli insegnanti possano più facilmente trasferire le loro competenze nelle classi.».

4. PROGETTARE LA STRUTTURA DI UN PROGRAMMA

I programmi possono essere entità complesse e di grandi dimensioni e, normalmente vengono organizzati in *moduli*, ossia componenti che, interagendo fra loro, danno attuazione alle funzioni identificate nelle specifiche. La suddivisione di un programma in moduli potrebbe apparire soprattutto come un'esigenza tecnica legata alla necessità di frammentare un programma di grandi dimensioni in

componenti di dimensione e complessità contenuta. Ma esistono altre ragione per suddividere un programma in moduli. Una delle più importanti è l'esigenza di incapsulare quelle parti di programma prevedibilmente soggette a modifica, per esempio, a causa del cambiamento di fattori esterni di cui il programma deve tenere conto. Concentrando in un solo modulo tutti gli aspetti relativi a quei fattori, tutte le volte che ci sarà un cambiamento, sarà sufficiente sostituire quel modulo lasciando tutto il resto del programma invariato.

In realtà, da un punto di vista concettuale, la suddivisione in moduli è la controparte strutturale del processo di astrazione che l'informatico ha messo in atto nella costruzione delle specifiche. Un modulo altro non è che una entità concreta che ha il compito di attuare una funzione astratta prevista dalle specifiche. David Parnas (1972), uno dei padri dell'Ingegneria del Software, scrisse che il compito principale di un modulo è nascondere un segreto, o, in altri termini, un dettaglio che a un certo livello di astrazione non è opportuno conoscere o rendere visibile. Questo perché mescolare livelli di dettaglio differenti rende i programmi meno leggibili e favorisce gli errori. Nascondere un segreto ha anche un altro significato: introdurre, nella struttura del programma, una separazione netta fra il *cosa* e il *come*. Definire un modulo significa definire con precisione la sua interfaccia, ma non il suo contenuto (di cui ci si occuperà nella successiva fase di *codifica*). Mentre l'interfaccia e la funzione svolta da un modulo sono pubbliche, il suo contenuto è strettamente privato, ignoto a tutti gli altri moduli. L'architettura di un programma può così essere pensata, e anche già parzialmente realizzata, come una collezione di moduli molto spesso organizzati in forma gerarchica. Ciascun modulo svolge una specifica funzione più o meno complessa e la comunicazione fra i moduli avviene attraverso le loro interfacce.

L'organizzazione di un programma in moduli può essere un processo ricco e significativo anche dal punto di vista dell'educazione al pensiero computazionale. Definire quali sono i moduli di un programma implica l'analisi di vari fattori e quindi porsi domande che aiutino a riflettere prima di decidere. A titolo di esempio, citiamo alcune domande che rappresentano possibili direzioni di riflessione. Qual è la ripartizione in moduli che meglio evidenzia le funzioni svolte dal programma? Quali sono le variazioni del problema (e quindi del programma) a cui potremo dovere/volere fare fronte? E quale ripartizione in moduli faciliterà maggiormente le possibili evoluzioni del programma? Come definire l'interfaccia di un modulo per dare evidenza alla logica del programma e favorirne la correttezza? Come utilizzate la ripartizione in moduli per favorire la collaborazione fra gruppi? E, non meno importante da un punto di vista concettuale, quale nome dare ai moduli per rendere evidente *a colpo d'occhio* la funzione svolta?

5. L'INFORMATICO: UN PARTICOLARE TIPO DI PROGETTISTA

Molti degli approcci concettuali che sono stati messi in luce fino ad ora non si riferiscono in modo esclusivo al modo di pensare degli informatici. Analizzare a fondo il contesto e i bisogni dell'utente, definire i requisiti di una soluzione, descrivere e comunicare la soluzione concettuale ideata, dare forma concreta alle componenti del progetto, distinguere chiaramente i tre momenti del *perché*, del *cosa* e del *come* sono tutti fattori che in larga misura caratterizzano qualunque tipo di attività progettuale. Quello che può cambiare sono i linguaggi di rappresentazione che i diversi progettisti applicano nelle diverse fasi della loro attività. Per esempio la progettazione in ambito meccanico richiederà strumenti concettuali e linguistici completamente diversi da quella in ambito elettronico o civile. Quindi, il modo di pensare degli informatici ha molto in comune con quello di molti altri progettisti. Vi è tuttavia una differenza significativa che non si riferisce soltanto alla natura dei sistemi progettati e ai linguaggi di progetto utilizzati. Mentre gli altri progettisti hanno un loro ambito di azione ben definito – la meccanica o l'elettronica o le costruzioni civili, solo per citare qualche esempio - l'informatico non ha un proprio settore di azione specifico. La vocazione dell'informatica è proprio quella di dialogare con le situazioni problematiche più diverse per trovare soluzioni attraverso lo sviluppo di software. Questo non significa che un informatico non possa o non debba avere un suo settore di specializzazione, ma significa che l'informatica dispone di strumenti concettuali di valore generale applicabili a una molteplicità di situazioni. Forse è proprio questa una delle ragioni più importanti per l'introduzione del pensiero computazionale in ambito educativo: la sua potenziale *declinabilità* con una grande varietà di ambiti disciplinari e tematici. L'informatica mette a disposizione delle cosiddette *fasi alte* dello sviluppo del software – quelle fasi cioè in cui non si pensa ancora al codice, ma a modellare e rappresentare i problemi e le soluzioni – una varietà di strumenti concettuali, veri e propri linguaggi che non hanno natura algoritmica, ma sono strumenti di rappresentazione e di specifica (Olimpo, 2011). Essi potrebbero agevolmente essere utilizzati nell'educazione, eventualmente in forma semplificata, con diverse finalità: rappresentare situazioni e problemi sostenendo il processo di comprensione e scoperta, rappresentare soluzioni garantendo la loro intrinseca consistenza, agevolare la collaborazione e aiutare la comunicazione. Senza voler entrare nel dettaglio, si possono citare alcuni esempi di linguaggi che, per la loro semplicità concettuale e per la loro natura grafica facilmente interpretabile anche dai non esperti, si prestano naturalmente ad una utilizzazione in ambito educativo: le Reti Semantiche, le Reti di Petri, gli schemi Entità Relazione, le Ontologie e, perché no, anche le mappe concettuali, ben note in ambito educativo e utilizzate con maggiore rigore e disciplina in ambito professionale, per esempio, per far emergere e rappresentare saperi impliciti. Questi linguaggi, fino ad oggi, hanno ricevuto scarsa attenzione in ambito educativo. Pochissime le eccezioni come p. es. Demo (2013) o Stroedter (2012).

Probabilmente la ragione è che questi linguaggi si riferiscono ad aree dell'informatica meno note, ritenute astratte, complesse e scarsamente operative, che non offrono cioè un riscontro di concretezza paragonabile all'esecuzione di un programma. L'ipotesi che qui si suggerisce è che proporre in ambito educativo, sia pure in forma semplificata, alcuni dei linguaggi/strumenti di rappresentazione utilizzati dagli informatici sia non solo possibile, ma anche ricco di significati in relazione ai processi di costruzione del sapere. Va detto che si ratta di un percorso in gran parte da costruire. Secondo Wing (2017), «*Ci sono delle interessanti domande di ricerca a cui gli informatici dovrebbero trovare una risposta in collaborazione con le comunità delle scienze cognitive e delle scienze dell'apprendimento. In primo luogo, quali concetti dell'informatica dovrebbero essere insegnati, a che livello scolare e con quale metodo*». È necessario formulare ipotesi realistiche sui linguaggi e sulle forme di astrazione che si possono proporre ai diversi livelli scolari. È necessario far convergere competenze informatiche disciplinari, didattiche e cognitive per esplorare come i diversi linguaggi possono arricchire la didattica dei diversi ambiti disciplinari e tematici e, nello stesso tempo, contribuire a forgiare capacità di astrazione, collaborazione, comunicazione e costruzione del sapere. Ed è anche necessario disporre di adeguati strumenti software orientati al mondo dell'educazione. Oggi è disponibile una varietà di strumenti professionali per supportare le *fasi alte* dello sviluppo del software, ma non ne esistono di specificamente orientati al mondo dell'educazione in termini di semplicità d'uso, qualità delle interfacce e funzionalità offerte. Solo per citare qualche esempio, fra i molti ambienti per il progetto di Reti di Petri, Woped[1] è uno dei più semplici da utilizzare, ma la sua interfaccia è troppo povera, non sufficientemente motivante e scarsamente adatta a dare evidenza agli aspetti semanticamente rilevanti di una Rete; esistono strumenti per tracciare schemi Entità-Relazione come SmartDraw[2] che hanno una buona qualità grafica, ma non vanno oltre al semplice tracciamento dello schema e non consentono di fare operazioni strutturali o semantiche su di esso.

Infine, val la pena di accennare come, da poco più di un decennio, si stiano diffondendo sempre più metodologie di tipo *agile* (Fowler & Highsmith, 2001) che si contrappongono al tradizionale processo di sviluppo del software proponendo un approccio meno strutturato, basato sempre meno su documenti descrittivi e sempre più sulla realizzazione rapida di una successione di prototipi funzionanti che costituiscono approssimazioni via via migliori e più complete del sistema che si vuole realizzare. In questo modo la comunicazione avviene sempre meno attraverso documenti e sempre più attraverso prototipi funzionanti, anche solo parzialmente. Naturalmente anche con questo tipo di approcci, l'utilizzazione di linguaggi di rappresentazione e di specifica mantiene comunque un ruolo

---

1 http://woped.dhbw-karlsruhe.de/woped/

2 https://www.smartdraw.com/

importante perché consente di disporre di una base concettuale di riferimento, sia pure in costante evoluzione, allo sviluppo dei prototipi. I successivi prototipi diventano così uno strumento agevole di comunicazione con chi non ha il tempo o la competenza per seguire un approccio più astratto e formalizzato e fanno convergere in modo rapido e sicuro verso il prodotto finale. Dal punto di vista educativo, la metodologia *agile* potrebbe essere interessante da esplorare. È infatti un approccio facilmente utilizzabile da chi e con chi non ha ancora sufficienti capacità di astrazione per utilizzare linguaggi formali; e introduce una nuova modalità di comunicazione attraverso artefatti, che consente di trasferire agevolmente e concretamente un'idea complessa.

6. PARLIAMO DI CODING

«*Informatica non equivale a programmazione. I giornali inglesi hanno pubblicato molti articoli con titoli del tipo "Perché dobbiamo insegnare il coding ai nostri bambini?" o "Coding, il nuovo Latino". Il pericolo è che insegnanti e politici, decidano di attivare, per esempio, dei corsi del linguaggio Java per i quattordicenni e dichiarino il problema risolto. Nella scienza c'è un'analogia che può essere di aiuto. La fisica, senza il lavoro di laboratorio sarebbe soltanto un'ombra di sé stessa, proprio come l'informatica senza programmazione sarebbe quasi un ossimoro. Ma nessuno si sognerebbe di far frequentare agli studenti un laboratorio di fisica senza insegare loro la fisica*» (Peyton Jones, Mitchell, & Humphreys, 2013). Un tempo la scrittura del codice che concludeva il percorso di progettazione del software, era considerata l'attività meno pregiata, lasciata ai cosiddetti *programmatori*, coloro che dovevano semplicemente scrivere parti di codice che si comportassero esattamente come qualcun altro aveva deciso. Semplici esecutori dunque. Questa concezione era nata soprattutto in connessione con lo sviluppo di applicazioni gestionali considerate all'epoca dagli informatici le più routinarie e meno complesse da un punto di vista concettuale. Col tempo, l'attività di codifica, non importa in quale linguaggio di programmazione, si è, per così dire, rivalutata.

Dal punto di vista educativo, saper programmare è certamente un valore. Significa aver capito cosa è un algoritmo e saper costruire algoritmi per risolvere problemi, non importa se in una dimensione reale o di gioco; significa saper comunicare in modo rigoroso con il computer che non ammette forme di comunicazione ambigua o approssimativa; e implica un certo patrimonio di conoscenza tecnologica per interagire con un computer e con un ambiente di programmazione. Qui non ci occuperemo dei meccanismi concettuali più significativi che la programmazione mette in gioco dal momento che numerosi autori lo hanno già fatto, p.es. (Brennan & Resnick, 2012) e (Olimpo, 2015). Si ritiene, a ragione, che la prima proprietà che un programma o un modulo di programma deve possedere sia la correttezza. Questo significa che il programma deve fornire risultati corretti per tutte le possibili combinazioni di dati di ingresso valide per quel tipo di programma. È un concetto semplice

da capire, ma molto difficile da mettere in pratica per il fatto che spesso le combinazioni possibili di dati di ingresso sono in numero così elevato che non è pensabile verificare il comportamento del programma in ogni possibile situazione. Non è sufficiente far *girare* qualche volta un programma con successo per dichiararlo corretto. Particolarmente critici sono alcuni settori come la programmazione parallela in cui gli errori diventano, per così dire, sfuggenti: a parità di dati di ingresso possono manifestarsi o non manifestarsi in base a come il computer porta avanti l'esecuzione del programma. Senza entrare nei temi classici del testing, delle dimostrazioni formali di correttezza o della ricerca degli errori nella programmazione (Ammann & Offut, 2017), ci limitiamo a osservare che scrivere il codice di un programma o di un modulo di programma comporta affrontare il formidabile problema concettuale della sua correttezza. E questo vale per il programma nella sua interezza e per ogni singolo modulo del programma. Se ci riferiamo all'educazione il problema della correttezza richiede non soltanto una buona conoscenza del processo algoritmico e dei tecnicismi dei linguaggi di programmazione, ma richiede di penetrare il comportamento del programma a un certo livello di profondità sia sul piano tecnico che su quello concettuale. Rappresenta un ottimo addestramento della capacità di indagine, un po' come capire chi è il colpevole di un crimine sulla base di indizi, o come capire il perché di un comportamento anomalo o inatteso in un esperimento scientifico. Già negli anni ottanta del secolo scorso Klahr e McCoy Carver (1988) avevano condotto interessanti esperimenti in ambito educativo sul tema della ricerca degli errori nei programmi. Questa ricerca era volta a identificare quali abilità fossero trasferibili e in quali contesti lo fossero con l'obiettivo di strutturare le attività di insegnamento/apprendimento in modo tale da favorire il transfer.

Tuttavia la correttezza /on è la sola proprietà interessante del codice. Programmare è un po' come scrivere. Si può scrivere bene o male. Capire il significato di un testo può essere facile o difficile. Leggere un testo può essere interessante o noioso. La stessa cosa avviene con i programmi. I buoni informatici sanno che nella programmazione possiamo trovare anche una dimensione estetica. Sanno che la chiarezza, la sinteticità e l'eleganza, oltre naturalmente alla correttezza, sono proprietà fondamentali dei programmi: «*…considero la programmazione un atto creativo che necessariamente coinvolge l'estetica. Alcuni considerano l'estetica nemica del pragmatismo e sostengono che non si dovrebbe perder tempo a scrivere codice elegante quando invece si può usare il tempo per scrivere codice efficiente. Tuttavia ritengo che il senso estetico sia il miglior servitore del pragmatismo perché porta a programmi più concisi e di più facile manutenzione…*» (Verborgh, 2013). Perché chiarezza ed eleganza sono proprietà importanti di un programma? La risposta è che produrre codice non è semplicemente una attività di comunicazione fra l'uomo e il computer, ma è anche, in egual misura, una attività di comunicazione fra persone. Un programma deve poter esser letto agevolmente da altri per una varietà di ragioni: collaborare a costruirlo, modificarlo in un momento successivo, correggerlo

agevolmente quando durante la vita del programma si manifestano errori, cercare soluzioni utili per la costruzione di un altro programma analogo, o anche soltanto capire come il programma funziona per imparare da un esempio di buona programmazione.

Programmare in modo semplice, chiaro ed elegante richiede capacità logiche e, in particolare, una forte capacità di astrazione: "buona programmazione" (Olimpo, 2015), non significa soltanto costruire buoni algoritmi, ma anche – o forse soprattutto - saper affrontare la complessità attraverso l'astrazione. Ciò che è complesso deve essere espresso in termini di entità più elementari, entità astratte perché ancora non esistono e che il programmatore deve saper inventare. Si tratta di un'invenzione che ha a che fare nello stesso tempo con la logica e con l'arte. Come nell'organizzazione di un discorso. È necessario trovare la struttura giusta, ogni frase deve avere un suo significato necessario, ogni parola il suo valore logico ed evocativo. Ne segue che quando si introduce il coding nell'educazione non ci si dovrebbe dare come unico obiettivo la comprensione di tutti gli elementi costitutivi del concetto di algoritmo e la capacità di scrittura di programmi corretti in questo o quel linguaggio di programmazione. Si dovrebbe anche puntare a una concezione dell'algoritmo come atto comunicativo interpersonale dotato di semplicità, efficacia ed eleganza. Non è un percorso semplice. Si possono forse trovare buoni esempi, ma regole da seguire per ottenere con certezza questi risultati non ce ne sono. In realtà, regole ce ne sono molte per esempio, la dimensione contenuta dei moduli, la scelta di nomi evocativi, l'esclusione di variabili di controllo nelle interfacce, ma da sole non sono sufficienti. Bisogna esplorare strade alternative e, gradualmente, forgiare la propria capacità di scrittura avendo a disposizione una guida competente. A quanto ci risulta, non esistono ricerche che abbiano dimostrato le possibilità di *transfer* ad altri ambiti comunicativi. È tuttavia ragionevole ipotizzare che la capacità di costruire programmi eleganti e ben strutturati, in quanto attività concettuale che mette in gioco meccanismi di astrazione, di gestione della complessità e di organizzazione della comunicazione possa favorire la formazione di un atteggiamento comunque utile sia nella comunicazione sia nella costruzione del proprio sapere.

7. CONCLUSIONI

Sono state tratteggiate le fasi in cui si articola l'attività dell'informatico impegnato nello sviluppo di software, e sono stati delineati, a grandi linee, i relativi strumenti e atteggiamenti di pensiero e alcuni loro possibili significati in ambito educativo. Nella maggior parte sono significati ancora da esplorare. Nella sua più recente definizione, Wing (2017) dice che «il pensiero computazionale è il processo di pensiero messo in gioco nel *formulare* un problema e nell'*esprimere* le sue soluzioni in modo tale che un agente computazionale – uomo o macchina – possa efficacemente portare a termine il processo risolutivo». E ancora «Il termine *esprimere* significa creare una rappresentazione linguistica con

l'obiettivo di comunicare una soluzione ad altre persone o a una macchina». Tuttavia, nella ricerca educativa e nella pratica didattica resta ancora da approfondire sia il processo di formulazione del problema, (quello che gli informatici affrontano nella prima fase dello sviluppo del software) sia l'aspetto della *espressione* delle soluzioni (quello che gli informatici affrontano nelle successive fasi di specifica funzionale, strutturazione e codifica del programma). Quelli che sono stati approfonditi in maniera privilegiata sono soprattutto i valori che il coding può portare all'educazione. E questo è avvenuto in modo particolare per i livelli scolari più bassi dove sarebbe stato difficile proporre altri aspetti del pensiero computazionale a causa dei limiti legati alle capacità di astrazione degli allievi. In questo ambito, i riferimenti non mancano, basti pensare, solo per citare gli esempi più famosi, al lavoro fondazionale di Papert (1980) e, in tempi più recenti, di altri ricercatori come Mitchel Resnik (Brennan and Resnick, 2012). Ma per i livelli scolari più elevati, anche il coding è stato scarsamente esplorato nelle sue dimensioni non strettamente tecniche, soprattutto in relazione alla dimensione di strumento per la comunicazione con altre persone. Con rare eccezioni, come Hazzan, Lapidot e Ragonis (2014), è molto difficile trovare riferimenti organici alla didattica dell'informatica per la fascia della scuola secondaria.

Rimane quindi ancora un ampio spazio di ricerca per estendere la portata del pensiero computazionale in ambito educativo. La direzione è quella di arrivare ad abbracciare tutto l'arco scolastico cercando di declinare i diversi approcci di pensiero adottati dagli informatici a cui si è fatto cenno con i diversi livelli scolari e con i diversi ambiti disciplinari e tematici. Questo processo presenta difficoltà oggettive e passa necessariamente da una ricerca educativa i cui obiettivi generali indicati da Wing (2017) sono già stati menzionati. Non è solo una questione di curriculum. In alcune discipline come la Matematica e la Fisica, la ricerca educativa ha fatto molto lavoro sulla didattica disciplinare e in particolare sulla cosiddetta conoscenza pedagogico contenutistica (PCK) (Abell, 2008). Per quanto riguarda l'informatica, «la conoscenza pedagogico contenutistica dei docenti, le loro convinzioni e i loro orientamenti svolgono un ruolo di grande importanza ai fini di un insegnamento efficace. Fino ad ora, questo ambito di ricerca è stato affrontato soltanto in modo occasionale e non esiste un modello consistente delle competenze necessarie per l'insegnamento dell'informatica.» (Bender *et al*., 2015). Il quadro si fa ulteriormente complesso se si tiene conto del fatto che gli aspetti relativi alla PCK non hanno una dimensione univoca ma sono fortemente dipendenti dalle scelte di sistema relative a come il pensiero computazionale viene integrato nel curriculum (materia a se stante e, se si, a partire da quale livello scolare, inserito all'interno di una disciplina scientifica, argomento cross-curricolare…). In tutti i casi esiste una forte esigenza di raccordo con e fra le discipline legata al fatto che il pensiero computazionale è per sua natura trasversale e non sussiste in astratto, ma si esercita nell'ambito di problematiche reali disciplinari o comunque tematiche come alcune definizioni alternative a quella

proposta da Janette Wing suggeriscono in modo esplicito: «Il pensiero computazionale è il processo di riconoscere aspetti di computazione nel mondo che ci circonda e di applicare strumenti e tecniche dell'Informatica per capire sistemi e processi sia naturali che artificiali e ragionare su di essi.» (The Royal Society, 2012).

Oltre agli aspetti di ricerca, per portare nella scuola una visione allargata del pensiero computazionale che davvero rifletta i valori concettuali presenti nel modo di pensare degli informatici, esistono formidabili problemi di formazione dei docenti a cui si è già fatto cenno che rendono comunque necessario un approccio graduale per l'introduzione del pensiero computazionale nella scuola. I problemi non riguardano soltanto il numero di docenti da formare, ma anche, o forse soprattutto, la natura della formazione richiesta. Per avvicinare i metodi degli informatici all'educazione bisogna avere una conoscenza sufficientemente approfondita di quei metodi, conoscenza di cui gli insegnanti di solito non dispongono. I concetti della buona programmazione e più in generale del pensiero computazionale cui si è fatto cenno sono apparentemente facili da capire, ma difficili da praticare. Richiedono, in qualche misura, di ri-forgiare il proprio modo di organizzare il pensiero, di affrontare i problemi, di rappresentare le soluzioni e di strutturare la comunicazione. È un compito che richiede tempo, impegno personale, guida e orientamento. Non è semplice e non coincide, se non in piccola parte, con l'apprendimento di un linguaggio di programmazione.

Secondo The Royal Society (2012), «*c'è bisogno di migliorare la comprensione della natura e delle implicazioni dell'introduzione del pensiero computazionale nella scuola*». L'esistenza di domande ancora aperte non significa che si debba aspettare di avere tutte le risposte. In molti paesi, in modo più o meno pragmatico, è stato introdotto o si sta introducendo, con differenti modalità, il pensiero computazionale nei curricula ai diversi livelli scolari. Si tratta di un processo «*molto promettente che può aiutare a preparare i bambini alle sfide future di una società sempre più digitalizzata…Ma, nelle prossime decadi, saranno i risultati che dovranno dimostrare che il pensiero computazionale ha una reale influenza sull'apprendimento e sulle abilità cognitive dei bambini. Via via che emergeranno risultati tangibili relativamente alle specifiche implementazioni e alle scelte pedagogiche fatte nei diversi paesi, lo scambio di esperienze e di lezioni imparate a livello europeo e internazionale diventerà un fattore di importanza cruciale*» (Bocconi et al., 2016). Si potrà così arrivare ad una visione del pensiero computazionale fondata anche su ipotesi confermate dall'esperienza.

8. REFERENCES

Abell, S. K. (2008). Twenty years later: Does pedagogical content knowledge remain a useful idea?. *International Journal of Science Education, 30*(10), 1405-1416. doi:10.1080/09500690802187041

Ammann, D. P., & Offut, J. (2017). *Introduction to Software Testing*. Cambridge, UK: Cambridge University Press.

Barr, D., Harrison, J., & Conery, L. (2011). Computational Thinking: A Digital Age Skill for Everyone. *Learning & Leading with Technology, 38*(6), 20–23. Retrieved from http://www.iste.org/docs/learning-and-leading-docs/march-2011-computational-thinking-ll386.pdf

Bender, E., Hubwieser, P., Schaper, N., Margaritis, M., Berges, M., Ohrndorf, L., ... & Schubert, S. (2015). Towards a competency model for teaching computer science. *Peabody Journal of Education, 90*(4), 519-532. doi:10.1080/0161956X.2015.1068082

Bocconi, S., Chioccariello, A., Dettori, G., Ferrari, A., & Engelhardt, K. (2016). *Developing computational thinking in compulsory education – Implications for policy and practice* (JRC Working Papers JRC104188). Seville, ES: Joint Research Centre. Retrieved from http://publications.jrc.ec.europa.eu/repository/handle/JRC104188

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the Congress of the American Educational Research Association* (pp. 1-25). Vancouver, CA: American Educational Research Association. Retrieved from http://web.media.mit.edu/~kbrennan/files/Brennan_Resnick_AERA2012_CT.pdf

D'Amore B., & Fandiño Pinilla, M. I. (2006). Che problema i problemi! *L'insegnamento della matematica e delle scienze integrate, 29*(6). 645-664. Retrieved from http://www.dm.unibo.it/rsddm/it/articoli/damore/588%20%20Problemi.pdf

D'Amore B. (2014). *Il problema di matematica nella pratica didattica*. Modena, IT: Digital Index Editore.

Demo, B. G. (2013, July). *Reading data schemas and knowing a db query interface in non technical secondary schools.* Paper presented at X World Conference on Computers in Education, Toruń, Poland. Retrieved from http://wcce2013.umk.pl/publications/v2/V2.14_196-Demo-fullN.pdf

European Commission (2015). *Relazione congiunta del Consiglio e della Commissione sull'attuazione del quadro strategico per la cooperazione europea nel settore dell'istruzione e della formazione (ET 2020) — Nuove priorità per la cooperazione europea nel settore dell'istruzione e della formazione.* Gazzetta Ufficiale dell'Unione Europea. Retrieved from http://eur-lex.europa.eu/legal-content/IT/TXT/PDF/?uri=CELEX:52015XG1215(02)&from=IT

European Commission. (2016). *A new skills agenda for Europe. Working together to strengthen human capital, employability and competitiveness*. Gazzetta Ufficiale dell'Unione Europea. Retrieved from https://ec.europa.eu/transparency/regdoc/rep/1/2016/EN/1-2016-381-EN-F1-1.PDF

Fowler M. and Highsmith J. (2001). The Agile Manifesto. Retrieved from http://agilemanifesto.org/

Halpern, D. (2003). *Thought and knowledge: An introduction to critical thinking (4th edition).* Mahwah, NJ: Earlbaum.

O. Hazzan, T. Lapidot, N. Ragonis (2014). *A Guide to Teaching Computer Science*. London, UK: Springer-Verlag

Klahr, D., & Carver, S. M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology, 20*(3), 362-404. doi:10.1016/0010-0285(88)90004-7

Manilla L., Dagiene, V., Demo, B., Grgurina, N., Mirolo, C., Rolandsson, L., & Settle, A. (2014). Computational Thinking in K-9 Education. In M. Goldweber (Ed.), *ITiCSE-WGR '14 Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference* (pp. 1-29). Uppsala, SE: Uppsala Universitet. doi:10.1145/2713609.2713610

Olimpo G. (2011). Information flows and graphic knowledge representations. Trentin G. (Eds.), *Technology and knowledge flows: the power of network*. Oxford, UK: Chandos Publishing.

Olimpo, G. (2015). Pensiero computazionale = buona programmazione e non solo. In V. Midoro (Ed.), *La scuola ai tempi del digitale.* Milano, IT: Franco Angeli Editore.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY: Basic Books.

Parnas D. (1972). *On the criteria to be used in decomposing systems into modules*. *Communications of the ACM, 14*(1), 1053-1058. Retrieved from https://www.cs.umd.edu/class/spring2003/cmsc838p/Design/criteria.pdf

Polya G. (1945). *How to Solve It.* Princeton, NJ: Princeton University Press (trad. it. *Come risolvere i problemi di matematica*, UTET Università, Torino, 2016).

Peyton Jones, S., Mitchell, B., & Humphreys, S. (2013, April). Computing at school in the UK. *Microsoft Research Papers.* Retrieved from https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/ComputingAtSchoolCACM.pdf

Strödter, C. (2012, November). Data modeling and database systems as part of general education in CSE. In *Proceedings of the 7th Workshop in Primary and Secondary Computing Education* (pp. 137-140). New York, NY: ACM. doi:10.1145/2481449.2481481

The Royal Society. (2012). *Shut down or restart? The way forward for computing in UK*

Schools. London, UK: The Royal Society. Retrieved from https://royalsociety.org/~/media/education/computing-in-schools/2012-01-12-computing-in-schools.pdf

Verborgh, R. (2013). Programming is an art [blog post]. Retrieved from http://ruben.verborgh.org/blog/2013/02/21/programming-is-an-art/

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology, 25*(1), 127-147. doi:10.1007/s10956-015-9581-5

Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33-35. doi:10.1145/1118178.1118215

Wing J. (2017), *Computational Thinking's Influence on Research and Education for All. International Journal of Educational Technology, 25*(2)