Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

IMPARARE IL PENSIERO COMPUTAZIONALE, IMPARARE A PROGRAMMARE

Tesi di Laurea in Didattica dell'Informatica

Relatore:
Chiar.mo Prof.
SIMONE MARTINI

Presentata da:
MICHAEL LODI

Sessione III
Anno Accademico 2012-13

Se vuoi imparare davvero qualcosa, insegnala!

Introduzione

Computer Science is no more about computers than astronomy is about telescopes.
– E. W. Dijkstra.

Insegnare e spiegare sono attività che mi sono trovato a svolgere, più o meno volontariamente, molte volte nella vita. Provando[1] a far capire un concetto cerco sempre di ricordare le difficoltà che ho avuto *io* nell'apprenderlo, e le cose che avrei voluto sentirmi dire per capirlo bene.

Visti i miei studi, il mio interesse si è rivolto nell'ultimo periodo alla Didattica dell'Informatica. Si tratta di un ambito che, a parer mio, è rimasto sempre un po' in ombra all'interno della nostra disciplina, al contrario di quanto avviene, tanto per fare un esempio, nella matematica.

Nell'ultimo decennio il *trend topic* nell'ambito dell'insegnamento dell'informatica è certamente il **pensiero computazionale**. Sebbene fosse un concetto già presente da decenni, in diverse declinazioni e sotto varie nomenclature, è stato portato all'attenzione della comunità scientifica nel 2006 da Jeannette Wing, direttrice del Dipartimento di Informatica della Carnegie Mellon University, tramite un lungimirante e appassionato articolo in cui mostra come, in tutta una serie di ambiti scientifici, l'informatica abbia portato non solo strumenti (computer e linguaggi di programmazione) ma

[1] Di solito riuscendoci, almeno stando ai commenti dei miei "studenti".

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* (chiamato anche *hill climbing con riavvio casuale* in Intelligenza Artificiale) per ricombinare i frammenti di DNA analizzato.

Se nelle discipline scientifiche il processo di diffusione del pensiero computazionale è già in atto, non possiamo dire lo stesso per quanto riguarda la vita di tutti i giorni. Viviamo in un mondo in cui i computer sono sempre più pervasivamente attorno a noi (*ubiquitous computing*). All'esplosione della disponibilità di calcolatori dalle dimensioni ridottissime, con enorme potenza di calcolo - resa praticamente infinita dall'accesso a Internet - e alla enorme disponibilità di informazioni - sempre grazie alla Rete - non ha fatto però seguito l'aumento della conoscenza e delle abilità connesse all'elaborazione di tale informazione.

Sembra quasi paradossale che, in un mondo in cui si creano continuamente nuove figure professionali legate ai computer e i vecchi lavori richiedono sempre maggiori competenze e abilità legate alla scienza e tecnologia informatica, essa abbia una scarsissima rilevanza nell'istruzione.

In Italia, e non solo, i non addetti ai lavori cadono spesso vittima della scorrettissima equazione "INFORMATICA = USO DEL COMPUTER". Questo ha portato inevitabilmente alla creazione, negli scorsi decenni, di corsi di "informatica" in cui si insegnava l'uso (spesso mnemonico) di un elaboratore di testo o di un foglio di calcolo.

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[2] *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. Ponendolo in una posizione così rilevante, è naturale preoccuparsi che tale approccio alla soluzione dei problemi venga insegnato a tutti gli studenti di tutti i livelli di istruzione.

Così come non si insegna *Analisi 1* ai bambini di prima elementare, allo stesso modo non possiamo pensare di insegnare loro *Programmazione 1*. Abbiamo bisogno di una scienza, da cui trarre i principi, i concetti, le tecniche con cui organizzare la conoscenza da trasmettere durante gli anni della scuola. Questa scienza non può che essere l'informatica (che, non a caso, forma i pensatori computazionali presi da esempio).

In questi anni gli insegnanti di informatica e gli scienziati dell'informazione hanno proposto curriculum, attività, framework, strumenti tecnici hardware e software per insegnare il pensiero computazionale. Il panorama rimane però ancora molto frammentato, e i principali lavori sono opera di informatici che si sono improvvisati esperti di educazione.

In realtà gli informatici si erano già occupati, soprattutto negli anni Ottanta e Novanta, di apprendimento. Sono infatti di quegli anni i maggiori studi della cosiddetta Psicologia della Programmazione. Si tratta di studi rigorosi, che si collocano a metà tra l'informatica e la psicologia cognitiva, che riguardano il "modo di pensare" dei programmatori, iniziati con lo scopo di valutare e migliorare gli strumenti a loro disposizione basandosi sugli effetti cognitivi che tali tool hanno sull'essere umano. Volendo rendere l'informatica accessibile a tutti, di particolare interesse sono gli studi sulle *misconcezioni*, che si occupano di individuare 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 - o più in generale di informatica - esprimano (in linguaggio naturale, ovviamente) concetti e processi computazionali.

[2]Forse ***computer scientist thinking*** avrebbe reso meglio l'idea.

Sebbene la programmazione sia solo uno degli aspetti della questione, è sicuramente l'aspetto centrale. I linguaggi di programmazione sono "il modo con cui gli informatici esprimono i concetti" e, non a caso, tanto gli studi degli anni Ottanta quanto quelli attuali si sono concentrati quasi esclusivamente su di essa o sulle attività ad essa collegate (algoritmi, debug, complessità, ecc.). Visto che, però, non possiamo partire da essa (almeno, non dall'insegnamento dei linguaggi *general purpose*), dovremo allora capire quali aspetti sono "intrinsechi" del pensiero computazionale e trovare modi di insegnarli che siano indipendenti dal linguaggio. Ricordando comunque che la vera potenza e originalità dell'informatica sta proprio nel poter eseguire le proprie astrazioni (in un certo senso dando loro vita), e dunque la necessità di un formalismo con cui esprimere procedimenti effettivi permane (almeno per il prossimo futuro).

Il piano è ambizioso, troppo per una tesi di laurea. Ci siamo perciò dati un obiettivo più limitato ma che può produrre effetti immediati. Dopo aver riesumato il background di studi sulle difficoltà da un lato, e sulla programmazione intuitiva dall'altro, ci proponiamo di fornire una serie di consigli per insegnare al meglio la programmazione (con i linguaggi attuali, con nuovi linguaggi appositamente progettati, con l'aiuto di strumenti ed ambienti ad-hoc) e per ideare e valutare linguaggi, metodologie e attività didattiche da inserire in altre discipline o in altri contesti, sempre volti all'insegnamento del pensiero computazionale.

Michael Lodi

Struttura della tesi

La struttura di questa tesi ricalca il ragionamento esposto nell'introduzione.

Il Capitolo 1, dopo un'analisi iniziale delle necessità e una breve digressione "filosofica", presenta l'idea di pensiero computazionale - prima e dopo l'ar-

ticolo di Jeannette Wing. La sezione 1.5 contiene un primo aspetto *originale* di questo lavoro: una definizione "operativa" di pensiero computazionale, basata su dieci *concetti chiave* individuati nella variegata letteratura che ha cercato di definirlo. Le sezioni seguenti mostrano le altre definizioni proposte, riconoscendo puntualmente in esse i concetti chiave presentati. Si elencano poi una serie di vantaggi che il pensiero computazionale può portare, e una veloce istantanea[3] dei progetti e degli strumenti proposti per l'insegnamento del computational thinking. Si concluderà con l'analisi dei limiti della ricerca allo stato attuale e delle critiche ad essa mosse.

Il Capitolo 2 ci porterà dapprima ad esplorare le principali *teorie dell'apprendimento* proposte dalla psicologia e pedagogia moderne (cognitivismo e costruttivismo). I concetti appresi saranno poi contestualizzati e utilizzati per mettere in evidenza le cause (psicologiche, pedagogiche, linguistiche, intrinseche...) di difficoltà nell'apprendimento dei linguaggi di programmazione. Verranno presentati infine i risultati di molti studi che evidenziano come la computazione e i linguaggi di programmazione vengono percepiti dai novizi.

Il Capitolo 3 contiene infine i *consigli dell'autore*. Dopo aver analizzato il rapporto tra informatica, pensiero computazionale e programmazione, si propongono due strade. La prima consiste in una serie di suggerimenti per migliorare l'insegnamento della programmazione (ponendo attenzione agli aspetti difficili, rendendo più naturali i costrutti, usando strumenti di visualizzazione o programmando in linguaggio naturale). La seconda strada invece pone le basi per rendere il pensiero computazionale "autonomo" rispetto alla programmazione (con consigli su come insegnarlo senza - ma anche con - l'ausilio di un formalismo).

[3] Quando invece sarebbe stato necessario un modello tridimensionale per presentare la mole di articoli visionata.

Indice

Capitolo 1

Il Pensiero Computazionale ovunque e per chiunque

> *All the high-school students will be taught the fundamentals of computer technology, will become proficient in binary arithmetic and will be trained to perfection in the use of the computer languages that will have developed out of those like the contemporary Fortran (from formula translation).*
>
> – Isaac Asimov (1964)
> *Previsioni sul 2014.*

1.1 Necessità

Lo sviluppo della scienza e della tecnologia informatica ha come una delle motivazioni fondanti l'estensione dell'intelletto umano.

Il processo, iniziato negli anni Quaranta del XX secolo, ha portato oggi a una miriade di applicazioni, concetti e tecniche disponibili per aumentare la produttività personale e non solo, in pressoché ogni campo della vita umana.

La tecnologia è sempre più diffusa: si va dall'uso quotidiano di apparecchi digitali a complesse applicazioni scientifiche e tecnologiche, passando per i più disparati settori come agricoltura, economia, giornalismo, scienze sociali.

Per fare un uso efficace dell'informatica nella vita, nel proprio campo di studi o nel lavoro, una persona necessita di alcune skill [114]:

- la prima è la capacità di utilizzare programmi applicativi di base (editor, browser, file system...), che chiameremo **alfabetizzazione informatica** (computer literacy);
- la seconda è una comprensione generale del funzionamento di un sistema informatico, chiamata **padronanza informatica** (computer fluency);
- la terza è l'insieme di strumenti intellettuali e critici che un professionista ha bisogno di padroneggiare per poter utilizzare le metodologie o le applicazioni informatiche per affrontare i problemi della propria disciplina (scienze fisiche, biologiche, sociali, materie umanistiche e arte), che potremmo chiamare **pensiero computazionale** (computational thinking).

Successi in campo accademico e professionale sono sempre più legati all'abilità di usare la tecnologia e la scienza informatica, la cui presenza e centralità sta permeando ogni disciplina. Nonostante ciò, molti studenti non hanno gli strumenti adeguati per superare questa sfida. Peggio: l'interesse degli studenti e l'iscrizione a corsi di Informatica è in declino.

- Il report "Running on Empty" dell'Association for Computing Machinery [144] sottolinea questo declino, individuando numerosi problemi nell'istruzione informatica negli USA: focus sulle tecnologie e sull'apprendimento di skill "sull'uso del computer" (alfabetizzazione) piuttosto che sui principi fondativi dell'informatica (padronanza o pensiero); mancanza di adozione di standard nell'educazione secondaria superiore; non obbligatorietà dei corsi di Informatica; mancanza di formazione e aggiornamento per gli insegnanti.

- Il report della Royal Society, "Shut Down or Restart?" [57] analizza lo stato dell'educazione informatica nel Regno Unito, giungendo a conclusioni analoghe (sono presenti curriculum per l'insegnamento dell'ICT potenzialmente validi nelle scuole, ma i ragazzi non sono ispirati da ciò che apprendono, assimilano solo skill di base) e individuando le possibili cause (il curriculum è troppo generico e difficilmente interpretabile da insegnanti non specialisti del settore, gli insegnati sono poco preparati e non c'è aggiornamento professionale, le infrastrutture non sono adeguate, la scuola non riconosce l'importanza dell'informatica come disciplina...).

- Il recentissimo report ACM "Rebooting the Pathway to Success" [79] sottolinea la crescita vertiginosa dei posti di lavoro legati all'informatica (circa 150.000 nuovi posti all'anno da qui al 2020, solo negli USA), con paghe elevate e con ampliamento a settori sempre nuovi. Il numero dei laureati dovrebbe tenere il passo con questi tassi di crescita, ma purtroppo non è così. Il report, inoltre, sottolinea la scarsa diversità di genere ed etnia dei lavoratori americani in questo settore: solo un quarto di donne e un decimo di afro-americani.

Una direzione per la risoluzione di questi problemi può essere quella ispirata dal dibattito apertosi pochi anni fa attorno al pensiero computazionale. Prima di analizzarlo nel dettaglio, facciamo un passo indietro.

1.2 Informatica e scienza

Il dibattito sulla scientificità e sul peso dell'informatica nella scienza è ampio.

Alcuni ritengono la "scienza computazionale" come terzo pilastro del metodo scientifico [121], e c'è chi addirittura ne aggiunge un quarto: l'uso della computazione per esplorare enormi quantità di dati. Forse è prematuro e dettato dall'entusiasmo del momento aggiungere addirittura due gambe

"computazionali" al metodo scientifico, ma va ovviamente riconosciuto come la computazione permei oggi teoria e sperimentazione [139].

Altri, ad esempio Rosenbloom [125], assegnano alla computazione il ruolo di *quarto grande dominio della scienza*, dopo le scienze **fisiche**, **biologiche** e **sociali**. Ognuno di questi domini studia fenomeni diversi (fenomeni fisici, esseri viventi, comportamento umano). La computazione non sembra rientrare in queste categorie: la computazione si realizza in tutti questi domini (meccanica quantistica, trascrizione del DNA, reti sociali) ed ognuno di essi usa metodi computazionali in modo estensivo, ma nessuno studia la computazione *di per sé.*

Possiamo allora assegnarle un dominio tutto suo? Sì, secondo Rosenbloom, poiché la computazione ha tutte le caratteristiche di un dominio: ha un suo focus (computazione e informazione), ha dei campi che la costituiscono (informatica, information technology, ingegneria informatica, ingegneria del software, sistemi informativi...), le sue strutture e i suoi processi sono in costante interazione, ha un'influenza pervasiva e notevole nella vita delle persone.

C'è però un'altra domanda, altrettanto complessa: *Is Computer Science Science?*. Quando un campo può essere definito *scienza*? Secondo Denning [50] quando soddisfa sei criteri:

1. ha un corpo organizzato di conoscenza;
2. i risultati sono riproducibili;
3. ha metodi sperimentali consolidati;
4. permette di fare previsioni, lasciando però spazio a sorprese;
5. offre ipotesi falsificabili;
6. studia oggetti naturali;

Se dei primi punti è facile convincersi [48], la discussione negli anni si è concentrata sull'ultimo. Se definiamo l'informatica come come *lo studio dei*

procedimenti effettivi di elaborazione dell'informazione, allora ci basterà mostrare che non sia "la scienza dell'artificiale". Negli ultimi vent'anni molti scienziati in molti campi hanno scoperto processi di informazione in natura, sostenendo il sesto criterio [49].

Possiamo quindi considerare l'informatica non solo un insieme di **applicazioni**, ma anche una **tecnologia** che rende possibili quelle applicazioni, ma anche una **scienza** che fonda quella tecnologia. Una scienza che contribuisce con concetti propri e originali, ma che ovviamente ne condivide altri con altre scienze e ha notevoli interazioni con esse [97].

1.3 Pensare computazionalmente

Già nel 1962, Alan Perlis [115] sosteneva la necessità degli studenti del college, di tutte le discipline, di imparare la programmazione e la teoria della computazione, per comprendere *in termini computazionali* materie quali Matematica ed Economia.

La locuzione *computational thinking* è stata usata per la prima volta da Seymour Papert in [110], parlando di educazione matematica con il suo linguaggio di programmazione didattico LOGO, sviluppato al MIT. Papert è il padre della teoria dell'apprendimento chiamata *costruzionismo*, una declinazione del costruttivismo (di cui parleremo estesamente nel § 2.1.3) che pone l'accento sugli *artefatti cognitivi*[1]. Secondo il matematico sudafricano infatti la mente umana, per imparare bene, ha bisogno di tali artefatti come "materiali da costruzione" per costruire rappresentazioni reali del mondo con cui interagisce. Per Papert il computer è un ottimo strumento didattico, in quanto, con la programmazione, può aiutare la costruzione di tali rappresentazioni.

[1]Gli artefatti cognitivi sono quei dispositivi artificiali che mantengono, mostrano, o operano sull'informazione al fine di svolgere una funzione rappresentativa, e che influiscono sulle prestazioni cognitive umane [106]. Sono le "cose artificiali" che sembrano favorire o migliorare le nostre capacità cognitive, alcuni esempi sono i calendari, le to-do list, i computer o semplicemente il "legarsi una corda intorno al dito come reminder" [14].

Nel suo famoso libro, *Mindstorms*, afferma che la programmazione favorisce il *pensiero procedurale*: insegna come spezzare il problema in componenti più semplici e "fare debug" su di esse se non funzionano. Questo modo di pensare, valido nella programmazione, può essere applicato a tutti gli altri aspetti della vita. "Pensare come un computer" è uno dei modi di pensare, e ha un suo valore, in quanto favorisce un modo *procedurale* di approcciare i problemi [109].

1.4 Un "movimento" per il Pensiero Computazionale

In un breve ma seminale articolo di otto anni fa [145], Jeannette Wing porta all'attenzione della comunità scientifica il concetto di **computational thinking**. Il concetto è indicato come skill indispensabile per tutti, non solo per gli informatici, che può essere usata nella vita di tutti i giorni oltre che nel lavoro e nelle scienze.

Il concetto può essere, in prima battuta, definito come un *modo di pensare* atto a "*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.*

Il pensiero computazionale viene proposto come quarta abilità di base ("the 4th of the 3 R's") oltre a leggere, scrivere, calcolare. Dunque bisognerebbe aggiungere il pensiero computazionale come abilità da insegnare ad ogni bambino. L'invenzione della stampa ha permesso la diffusione delle prime tre, la diffusione della tecnologia deve permettere, secondo Wing, la diffusione del pensiero computazionale.

A sostegno di questa tesi, vengono elencati:

- una lunga serie di *strumenti mentali* che gli informatici possiedono (es. farsi domande sulla complessità e sull'efficienza, chiedersi se una soluzione approssimata o basata su metodi *random* può funzionare,

pensare ricorsivamente, usare astrazione e decomposizione quando si affronta un problema complesso, ...);

- una serie di aspetti pratici della vita in cui usare il pensiero computazionale (es. ordinare un mazzo di carte, disporre in modo ordinato ed efficiente i vari piatti e portate di un buffet, specificare le istruzioni per montare un mobile, ...);

- una serie di esempi che testimoniano come l'informatica nelle altre discipline sia *più* che utilizzare gli strumenti tecnici messi a disposizione da questa, ma cambi il modo di pensare (es. basi azotate come un linguaggio), ponga nuove domande e nuove risposte (es. quelle date dall'analisi di enormi quantità di dati).

Vengono elencate poi alcune caratteristiche del pensiero computazionale.

- Astrazione. Non è (solo) *computer programming*, ma pensare a più livelli di astrazione.

- Una skill fondazionale, non tecnica.

- Un modo di pensare umano: intelligente, creativo. Avere a disposizione i computer ci dà il coraggio di affrontare problemi complessi. Le funzionalità che vogliamo sono realizzabili tramite i computer, l'unico limite è la nostra immaginazione.

- Completa e combina *pensiero matematico* e *pensiero ingegneristico*: si basa su di essi perché è fondato dalla matematica e lavora con oggetti nel mondo reale. Rispetto al pensiero matematico deve tener conto del dispositivo ma, potendo costruire mondi virtuali può decidere non curarsi dell'aspetto fisico.

- Riguarda le idee, non gli artefatti, cioè riguarda i concetti e gli approcci per risolvere problemi, gestire la nostra vita, comunicare.

- Riguarda tutti, in ogni luogo. Il pensiero computazionale sarà davvero realtà quando sarà così integrato nel comportamento umano da non richiedere di essere più esplicitato come filosofia di pensiero.

Infine viene sottolineato come la diffusione del pensiero computazionale possa cambiare la percezione dell'informatica nel mondo. Per fare ciò è necessario:

- spiegare che la ricerca scientifica in informatica è ancora attiva e ha tantissime domande aperte;
- spiegare come le lauree in Informatica permettano moltissime opportunità di lavoro al di fuori dell'informatica stessa;
- inserire un esame di Pensiero Computazionale in tutti i corsi di laurea;
- inserire il pensiero computazionale nel curriculum K-12.

A partire da questo articolo, è stata prodotta una certa quantità di letteratura che ha provato a definire il pensiero computazionale e possibili strategie per insegnarlo, in particolare nel contesto K-12 (ovvero che va dal *Kindergarten* al *Grade 12*, in Italia dalla scuola materna alla fine delle scuole superiori).

1.5 Una definizione operativa

Sebbene le disquisizioni filosofiche sulla natura dell'informatica e del pensiero computazionale siano di grande importanza per costituire le fondamenta di tali discipline, è da molti [22] riconosciuta la necessità *pratica* di una definizione "operativa" di pensiero computazionale, per permettere - nel complesso e altamente burocratizzato sistema dell'educazione pre-uniersitaria - la definizione di obiettivi educativi e la loro valutazione.

Un'analisi della letteratura 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.

1.5.1 Concetti

I concetti ricorrenti nelle definizioni di pensiero computazionale sono elencati di seguito, con accanto un'etichetta che aiuterà ad identificarli nelle varie definizioni proposte in letteratura e con i riferimenti agli articoli principali che li hanno riconosciuti come componenti di tale pensiero.

	Concetto	***Riferimenti***
Dati	**Collezione e analisi dei dati**	[76]
Rappr	**Rappresentazione dei dati**	[76]
Decomp	**Decomposizione dei problemi**	[76, 63]
Astr	**Astrazione**	[43, 76, 63]
Gen	**Generalizzazione e ricon. pattern**	Personale, idea da [40, 63]
Algo	**Algoritmi**	[76, 63]
Auto	**Automazione**	[43, 76, 63]
Test	**Simulazione, test, debug**	Simulazione: [76]; Test/debug: person. idea in [43, 89]
Par	**Parallelizzazione**	[76]
Comp	**Calcolabilità e complessità**	Personale

Tabella 1.1: Concetti costituenti il pensiero computazionale

Possiamo dare, in modo molto generale, le seguenti definizioni.

1. **Collezione e analisi dei dati**. Il processo di raccolta delle informazioni appropriate, e di analisi - per dare loro un senso, trovando pattern comuni e traendo conclusioni dai dati stessi.

2. **Rappresentazione dei dati**. Il processo di rappresentazione e organizzazione di dati e risultati, sia visiva (grafici, testo o immagini) sia astratta (strutture dati).

3. **Decomposizione dei problemi**. Il processo di divisione del problema in parti più piccole e affrontabili.

4. **Astrazione**. Il processo di riduzione della complessità, per far emergere l'idea principale mantenendo solo alcuni aspetti e tralasciandone altri.

5. **Generalizzazione e riconoscimento di pattern**. L'abilità di riconoscere come alcune parti di soluzione possono essere riusate nella stessa o riapplicate a problemi simili.

6. **Algoritmi**. Una serie ordinata di passi per risolvere un problema o raggiungere un obiettivo.

7. **Automazione**. Lasciare ad una macchina i compiti ripetitivi o noiosi, formalizzandoli e facendoglieli eseguire.

8. **Simulazione, test, debug**. Modellare un processo ed eseguire esperimenti su di esso. Individuare problemi/errori e correggerli.

9. **Parallelizzazione**. Organizzare risorse per far loro eseguire task simultanei allo scopo di raggiungere un obiettivo comune.

10. **Complessità e calcolabilità**. Individuare un metodo che raggiunga un risultato, possibilmente il migliore e usando meno risorse (tempo, memoria, potenza di calcolo, energia[2]).

[2]Oggi particolarmente importante nei dispositivi portatili.

1.5.2 Definizione proposta

Possiamo a questo punto dare una definizione "operativa" di pensiero computazionale per l'educazione pre-universitaria, ispirandoci a quella prodotta dalla International Society for Technology in Education (ISTE) e la Computer Science Teachers Association (CSTA) [77], ma con alcune personali modifiche e integrazioni.

Definizione 1.1 (**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)[3] 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.

1.5.3 Saper fare

Sono state individuate le competenze da acquisire:

[3]Storicamente, il termine *computer* si riferiva a impiegati contabili che eseguivano calcoli [42].

- saper progettare soluzioni ad un problema (collezionando dati e analizzandoli, astraendo per conservare gli aspetti importanti, scomponendo il problema in sottoproblemi, creando algoritmi);
- saper implementare i progetti (programmando e automatizzando la soluzione);
- saper eseguire test e debug;
- saper modellare la realtà ed eseguire simulazioni;
- saper riflettere sul lavoro svolto valutandolo secondo certi criteri;
- saper generalizzare una soluzione e adattarla ad altri ambiti;
- saper rappresentare e comunicare i propri risultati;
- saper lavorare in team e comunicare con gli altri;
- saper usare il vocabolario appropriato.

1.5.4 (Pre)requisiti

Le caratteristiche elencate sono supportate ed esaltate da un numero di predisposizioni e attitudini, quali [22]:

- capacità di gestione della complessità;
- perseveranza nell'affrontare problemi difficili;
- tolleranza per l'ambiguità;
- saper fronteggiare problemi aperti;
- saper comunicare e lavorare con gli altri per raggiungere un obiettivo o una soluzione comune;
- conoscere i propri punti di forza e le proprie carenze quando si lavora con gli altri.

1.5.5 Strategie in classe

Per definire una "cultura della classe" che stimoli il pensiero computazionale, possono essere seguite alcune strategie [22]:

- insegnanti e studenti devono adottare il vocabolario del pensiero computazionale quando appropriato per descrivere problemi e soluzioni;
- insegnanti e studenti devono accettare soluzioni errate e tentativi falliti, riconoscendo che sono parte di un percorso per un risultato positivo;
- gli studenti dovrebbero lavorare in team essendo esplicitamente incoraggiati ad utilizzare:
  - decomposizione dei problemi in piccole parti più facilmente risolvibili,
  - astrazioni: semplificare e generalizzare i problemi,
  - negoziazione: lavorare in gruppi e poi lavorare per integrare le soluzioni in un tutto,
  - costruzione del consenso: lavorare per avere l'appoggio del gruppo attorno a un'idea.

1.6 Altre definizioni

Esploriamo ora le altre definizioni proposte in questi anni. In molte di esse ritroveremo, indicati tra parentesi con le loro etichette, i concetti già individuati e inseriti nella definizione proposta.

1.6.1 Google

Google ha istituito un sito web [63] dal titolo *Exploring Computational Thinking*, con l'obiettivo di promuovere l'insegnamento del pensiero computazionale nell'educazione pre-universitaria. Il sito fornisce una definizione e un elenco di skill ed esempi.

Definizione 1.2. Il **pensiero computazionale** coinvolge una serie di abilità di problem-solving e tecniche che gli informatici utilizzano per scrivere i programmi, ma può essere applicato ad ogni disciplina. Gli studenti che apprendono il pensiero computazionale iniziano a scoprire relazioni tra diverse materie e tra ciò che si studia in classe e la vita al di fuori di essa.

Il pensiero computazionale include specifiche ***tecniche***:

- **Decomposizione** (Decomp). Si tratta dell'abilità di spezzare un compito in "passi atomici", così da poterlo spiegare a un'altra persona o ad un computer. Questo porta spesso a *riconoscere pattern*, a *generalizzarli* e quindi a *progettare un algoritmo*. Esempio: dando indicazioni stradali per casa nostra, stiamo decomponendo il processo di "andare da un posto ad un altro".

- **Riconoscimento di pattern** (Gen). Si tratta dell'abilità di notare similitudini o differenze comuni che ci permettono di fare previsioni o ci portano a scorciatoie. È spesso la base per risolvere problemi e progettare algoritmi. Esempi: i bambini identificano pattern nella reazione degli educatori a loro comportamenti per capire cosa è giusto e cosa è sbagliato; le persone cercano pattern nell'andamento dei prezzi delle azioni in borsa per decidere quando comprare e quando vendere.

- **Generalizzazione di pattern e astrazione** (Astr, Gen). Si tratta dell'abilità di filtrare informazioni non necessarie e generalizzare quelle che invece sono utili. Ci permettono di rappresentare un'idea o un processo in termini generali, così da poterlo utilizzare per risolvere problemi simili. Esempi: in matematica, formule generalizzate tramite variabili invece di numeri possono essere utilizzate per risolvere problemi con valori diversi; un'astrazione della Terra in termini di latitudine e longitudine ci aiuta a descrivere posizioni su di essa.

- **Progettazione di algoritmi** (Algo). Si tratta dell'abilità di sviluppare una strategia passo-passo per risolvere un problema. È basato sulle

abilità precedenti. Esempi: uno chef che crea una ricetta da seguire, uno studente che trova il minimo comune multiplo tra due denominatori di frazioni o che automatizza con un programma in Python il calcolo degli interessi su un debito.

1.6.2 Wing et al.

Wing, insieme con Cuny e Snyder, in un'opera citata ma mai apparsa [43], in un articolo divulgativo [147], e anche indirettamente in [89], assume questa definizione.

Definizione 1.3. Il **pensiero computazionale** è il processo mentale coinvolto nel formulare problemi (Decomp, Astr) e loro soluzioni (Algo) rappresentate in una forma che sia effettivamente eseguibile (Comp, Auto) da un *agente che processa informazioni.*

Le sue caratteristiche principali sono tre.

- **Astrazione** (Astr): consiste nel generalizzare da specifiche istanze, catturarne le caratteristiche comuni per rappresentare tutte le istanze aventi tali proprietà.
  Per Wing è un concetto cruciale del pensiero computazionale e è importante sia decidere **cosa** astrarre sia il "livello di astrazione" e le relazioni tra i diversi livelli di astrazione. Si tratta del ***tool mentale*** del pensiero computazionale.

- **Automazione** (Auto, Par): consiste nel delegare al computer una serie di compiti ripetitivi e noiosi, che esso svolge velocemente e con precisione. Si può pensare ai programmi come "automazione di astrazioni". Per automatizzare le astrazioni abbiamo bisogno di questi ***tool di metallo***: i computer. Possono essere macchine (del presente e del futuro, quindi anche computer quantistici o basati su strutture biologiche) oppure umani (che, ad esempio sono ancora migliori dei computer nel processare immagini) oppure la combinazione dei due, o reti di combinazioni.

- **Analisi** (Test, Comp): è la pratica riflessiva sulla correttezza delle astrazioni effettuate. Include domande quali: *Sono state fatte le giuste assunzioni per caratterizzare il problema nella sua generalità? Sono stati dimenticati fattori importanti? L'automazione delle astrazioni aveva dei difetti o non funzionava o invece era corretta?*

Il pensiero computazionale può essere riassunto nella domanda: Come faccio a far risolvere questo a un "computer"[4]? Non si vuole usare semplicemente un approccio *brute force*, ma trovare le astrazioni giuste e il computer giusto in modo intelligente e *creativo*.

1.6.3 csprinciples.org

Allo scopo di creare il nuovo corso di Informatica per l'Advanced Placement, NSF e College Board hanno dato vita al programma chiamato "Computer Science: Principles". Il contenuto fondamentale della disciplina è stato identificato per essere alla base di un curriculum di ampio interesse per gli studenti delle scuole superiori.

Espressi nelle "Sette Grandi Idee", questi principi sono accompagnati da "Sei Pratiche di Pensiero Computazionale" che *individuano le attività svolte dagli informatici, ma che possono essere apprese da tutti e poi applicate ad altre discipline.*

Seven Big Ideas

1. L'uso del computer è un'attività umana creativa che genera innovazione e promuove l'esplorazione.

2. L'astrazione (Astr) riduce i dettagli per concentrarsi solo sugli aspetti rilevanti per la comprensione e la soluzione di problemi.

3. Dati e informazioni (Dati) facilitano la creazione di conoscenza.

[4]Nel senso ampio visto prima.

4. Gli algoritmi (Algo) sono gli strumenti per sviluppare ed esprimere soluzioni a problemi computazionali.

5. Programmare (Auto) è un processo creativo che produce artefatti computazionali.

6. I dispositivi digitali, sistemi, e le reti (Par) che le interconnettono permettono e favoriscono approcci computazionali alla soluzione di problemi.

7. L'uso del computer permette innovazione in altri campi (Gen): scienze, scienze sociali, discipline umanistiche, arti, medicina, ingegneria, economia.

Six Computational Thinking Practices

- **Analizzare gli effetti della computazione**
  - Identificazione di innovazioni esistenti e potenziali permesse dalla tecnologia informatica
  - Identificazione delle implicazioni etiche dello sviluppo della computazione
  - Identificazione degli impatti positivi e negativi delle innovazioni informatiche sulla società
  - Analisi delle conseguenze delle decisioni di design
  - Valutazione dell'usabilità di un artefatto computazionale
  - Caratterizzazione delle connessioni tra bisogni umani e funzionalità informatiche
  - Spiegazione delle problematiche relative alla proprietà intellettuale
- **Creazione di artefatti computazionali**

  - Creazione di un artefatto ritenuto dagli studenti interessante e importante
  - Progettazione di una soluzione ad un problema dato
  - Scelta dell'approccio appropriato per risolvere un problema
  - Uso appropriato di algoritmi conosciuti
  - Uso appropriato di costrutti della programmazione e strutture dati
  - Valutazione di un artefatto usando criteri multipli
  - Individuazione e correzione di errori
  - Uso di appropriate tecniche di sviluppo di un artefatto computazionale

- **Usare astrazioni e modelli**

  - Spiegazione di come dati, informazioni e conoscenza sono rappresentati per un uso computazionale
  - Uso della simulazione per investigare domande poste e/o sviluppare nuove domande
  - Scelta di principi algoritmici ad un appropriato livello di astrazione
  - Uso di diversi livelli di astrazione
  - Specifica del tipo di progettazione per un modello o simulazione
  - Uso dell'astrazione sui dati
  - Collezione e generazione di dati appropriati per modellare un fenomeno
  - Confronto tra dati generati ed esempi empirici

- **Analizzare problemi e artefatti**

  - Identificazione di problemi e artefatti che hanno una data proprietà

  - Confronto tra tool disponibili per la soluzione di un problema
  - Valutazione di una soluzione proposta per un problema e implicazioni dell'uso di tale soluzione
  - Analisi dei costi/benefici di una soluzione
  - Anali del risultato di un programma
  - Valutazione delle caratteristiche di problemi e artefatti

- **Comunicare processi e risultati**
  - Spiegazione del significato di risultati
  - Descrizione dell'impatto di una tecnologia o artefatto
  - Descrizione del comportamento di un artefatto computazione
  - Spiegazione del progetto di un artefatto
  - Descrizione di una tecnologia
  - Giustificazione di correttezza e appropriatezza

- **Lavorare in modo efficace in team**
  - Applicazione di strategie di *teamwork* efficaci
  - Collaborazione
  - Produzione di artefatti che dipendono dal contributo attivo di tutti i partecipanti
  - Documentazione che descriva uso, funzionalità ed implementazione di un artefatto

1.6.4 Perković e Settle

Nel tentativo di creare un framework [114] per implementare la visione di Wing nei corsi di Laurea, i due autori elencano una serie di caratteristiche del pensiero computazionale, riprendendole dai "Grandi principi dell'informatica" proposti da Denning [47] ma dando definizioni di più ampio respiro, in

quanto applicabili a discipline tra le più varie (arte e letteratura, filosofia, religione, ricerca scientifica, scienze sociali, storia). Le elenchiamo con una breve spiegazione e con l'indicazione di parole chiave che meglio le identificano.

- **Computazione** (Algo, Auto): è l'esecuzione di un algoritmo, un processo che parte da uno stato iniziale contenente l'algoritmo stesso e i dati di input, e passa attraverso una serie di stati intermedi, fino a raggiungere uno stato finale (obiettivo).
  Parole chiave: stato e transizione di stato, algoritmo, programma, ricerca esaustiva, backtracking, ricorsione e iterazione, albero di decisione, randomizzazione, complessità, calcolabilità.

- **Comunicazione** (Par): è la trasmissione delle informazioni da un processo od oggetto ad un altro.
  Parole chiave: informazione e sua rappresentazione, messaggi, mittente/destinatario, protocollo di comunicazione, compressione dei messaggi, cifratura dei messaggi, correzione d'errore, canale di comunicazione, codifica/decodifica, rumore, autenticazione.

- **Coordinazione** (Par): è il controllo, ad esempio attraverso la comunicazione, della temporizzazione dei processi che partecipano al calcolo, al fine di raggiungere un certo scopo.
  Parole chiave: processi e agenti interagenti, protocolli inter-processo che includono protocolli di comunicazione, sincronizzazione, eventi e gestione degli eventi, dipendenze, concorrenza.

- **Memoria** (Dati, Rappr): è la codifica e l'organizzazione dei dati in modo che sia efficiente ricercarli e svolgere le altre operazioni desiderate su di essi.
  Parole chiave: dispositivi di memoria, gerarchia e organizzazione dei dati, manipolazione dei dati come inserimenti, aggiornamenti, rimozioni e query, localizzazione dei dati e caching, rappresentazione virtuale, riferimenti assoluti e relativi.

- **Automazione** (Auto): significa mappare la computazione ai sistemi fisici che la eseguono.
  Parole chiave: mapping di un algoritmo in un oggetto che esegue fisicamente il calcolo, meccanizzazione, applicata a processi ripetitivi per offrire esecuzioni consistenti, senza errori, veloci e computazionalmente efficienti.

- **Valutazione** (Dati, Comp): è l'analisi statistica, numerica o sperimentale dei dati.
  Parole chiave: visualizzazione, analisi dei dati, statistica, data mining, simulazione, esperimenti computazionali.

- **Design** (Decomp, Astr, Gen): è l'organizzazione (usando astrazione, modularizzazione, aggregazione, decomposizione) di un sistema, di un processo, di un oggetto, ecc.
  Parole chiave: astrazione, livelli di astrazione, modellazione, modularità, occultazione dell'informazione, classe, architettura, aggregazione, pattern, struttura sottostante.

1.6.5 Altri

Bundy

Alan Bundy [33], tirando le somme di una serie di seminari tenutisi all'Università di Edimburgo, analizza la pervasività del concetto di pensiero computazionale.

Il pensiero computazionale aiuta i ricercatori a farsi nuove domande e ad accettare nuove risposte, basate sul processamento di una grande quantità di dati. Usare il pensiero computazionale è più che usare la tecnologia: è un nuovo linguaggio per descrivere ipotesi e teorie, un'estensione delle nostre facoltà cognitive.

In particolare, Bundy riconosce tre aspetti chiave:

1. **e-science**: alcune domande possono trovare risposta solo collezionando e analizzando una grande quantità di dati, spesso in modo distribuito "grid like" (per citarne una, si pensi alle attività svolte al CERN di Ginevra);

2. inserimento di **metafore computazionali** per descrivere ipotesi e teorie (una tra tutte: *brain as computer and mind as a program*);

3. estensione delle nostre **facoltà cognitive** (es. sistemi esperti per aiutare giudici, investigatori, medici).

Royal Society

Nel già citato report [57], si definisce il pensiero computazionale come *il processo di riconoscimento degli aspetti della computazione nel mondo che ci circonda, e l'applicazione di tool e tecniche dell'informatica per capire e ragionare su sistemi e processi, sia naturali che artificiali.*

1.7 A cosa serve?

Perché insegnare il pensiero computazionale nell'ambito pre-universitario? Secondo il "Workshop per stabilirne la natura e l'ambito" [104], permetterebbe alle persone di destreggiarsi meglio nella società contemporanea, che non può prescindere dalla tecnologia:

- permetterebbe un *accesso egualitario* a skill, risorse e dispositivi tecnologici, favorendo una crescita personale dell'individuo e una partecipazione più equa alla vita sociale che richiede oggi tali abilità;

- aumenterebbe l'*interesse degli studenti* nei campi dell'informatica e dell'ingegneria, e di tutte le discipline ad alto contenuto tecnologico;

- preparerebbe gli studenti con *competenze* necessarie ad essere forza lavoro in un ambiente internazionale molto competitivo.

Per Jeannette Wing i benefici si vedono sia nella vita di tutti i giorni, sia in ambito professionale [147].

Tutti dovrebbero essere capaci di:

- capire quali aspetti di un problema possono essere demandati ad una computazione;
- valutare l'intersezione tra strumenti di calcolo, tecniche e problemi (tenendo conto delle limitazioni e della potenza di strumenti e tecniche);
- applicare un tool o una tecnica ad un uso nuovo;
- rendersi conto di poter usare la computazione in modi nuovi e creativi;
- applicare strategie computazionali in ogni dominio.

Scienziati, ingegneri, professionisti dovrebbero essere inoltre in grado di grado di:

- applicare nuovi metodi computazionali ai loro problemi;
- riformulare problemi di modo che possano essere attaccati con strategie computazionali;
- scoprire nuovi fatti scientifici grazie all'analisi di grandi quantità di dati;
- farsi nuove domande rimaste nascoste a causa della dimensione del problema, oggi attaccabile computazionalmente;
- spiegare problemi e soluzioni in termini computazionali.

Barr, Harrison e Conery [21] pongono l'attenzione sul fatto che non siano skill nuove quelle da insegnare, ma il pensiero computazionale differisce dal pensiero critico/matematico in quanto:

- è una combinazione unica di skill che, se usate insieme, forniscono una nuova forma di problem solving;

- è più orientato al tool;

- usa tecniche come *prove ed errori*, *iterazioni*, e persino il *provare a caso* in contesti in cui non erano prima praticabili, ma lo diventano tramite l'automazione e la velocità di elaborazione (basti pensare all'algoritmo *shotgun* per il sequenziamento del DNA).

La mente umana è ancora il tool migliore per risolvere problemi, ma possiamo *estendere il suo potere con i computer*. Fanno parte della nostra vita e dobbiamo imparare a comunicare con gli strumenti che possono aiutarci a risolvere problemi.

Ovviamente gli studenti imparano alcune delle abilità del pensiero computazionale in altre materie, ma dobbiamo assicurarci che le imparino tutte, di modo da usarne il potere della loro combinazione, assicurandoci che sappiano trasferirle a problemi diversi e a contesti diversi.

1.8 Una fotografia (sfocata)

La letteratura sul pensiero computazionale riporta numerosi esempi di suo utilizzo nella vita e nelle discipline scientifiche [145, 146, 147].

Le Università (seguendo l'esempio della Carnegie Mellon, ateneo di Jeannette Wing [32]) stanno rivedendo alla luce del pensiero computazionale i loro curriculum per i corsi di laurea in Informatica e soprattuto gli esami di "Introduzione all'Informatica" negli altri corsi di laurea. In Irlanda esiste una laurea interamente dedicata al pensiero computazionale [5].

Visto che lo *scope* di questa tesi è la formazione pre-universitaria, elenchiamo alcune organizzazioni che promuovono il pensiero computazionale, e alcuni strumenti pensati per insegnare tale pensiero (o, più in generale, i principi dell'informatica) a bambini e ragazzi.

1.8.1 Organizzazioni

Molti enti governativi, scientifici, società private, associazioni di volontari stanno concentrando tempo e risorse nella diffusione del pensiero computazionale.

Regno Unito Dopo il report [57] che testimoniava l'infelice situazione nelle scuole britanniche, il Ministero dell'Istruzione del Regno Unito sta implementando i consigli della Royal Society per l'insegnamento dell'informatica e del pensiero computazionale nella propria riforma scolastica [11].

Carnegie Mellon La Carnegie Mellon University ha istituito il "Center for Computational Thinking" [3], allo scopo di diffondere e ravvivare l'uso del pensiero computazionale attraverso progetti di ricerca, attività e seminari. In particolare, finanzia le cosiddette "PROBE - PORBlem-oriented Explorations", progetti di ricerca che dimostrano l'importanza del pensiero computazionale (applicando concetti computazionali a problemi non tradizionali, esplorando aspetti didattici o collaborazioni tra informatici ed esperti in altri campi).

National Science Foundation La NSF americana, come già visto, sta preparando - insieme al College Board - un curriculum per l'Advanced Placement (corsi per le scuole superiori ma di livello universitario) per il 2016. La sperimentazione sta coinvolgendo sempre più scuole e dando buoni risultati [7].

NSF finanzia inoltre molti progetti [105] per la diffusione del pensiero computazionale tra gli studenti e tra gli insegnanti.

Computer Science Teachers Association CSTA mette a disposizione una serie di materiali [6] per l'insegnamento e l'apprendimento del pensiero computazionale. In particolare fornisce risorse per gli insegnanti [76] in cui, per ognuna delle caratteristiche individuate dalla loro definizione [77],

propone una serie di attività interdisciplinari, multidisciplinari, pensate per specifiche fasce di età, allo scopo di favorire l'apprendimento di tale aspetto del pensiero computazionale.

Google Come già visto, Google raccoglie in un sito web [63] moltissime risorse online per l'insegnamento del pensiero computazionale nell'informatica e in altre discipline scientifiche.

Inoltre, con il progetto CS4HS [62], finanzia annualmente corsi di formazione professionale di informatica per insegnanti di scuola secondaria inferiore e superiore.

Organizzazioni volontarie e non-profit Le associazioni "Computing in the Core" e "Code.org" organizzano la "Computer Science Education Week" [41] ogni dicembre, per ispirare gli studenti a interessarsi all'informatica.

Movimenti quali Fab Labs, Makerspaces, Maker Faire promuovono la costruzione di artefatti computazionali tangibili.

CoderDojo [4] dà la possibilità a bambini e ragazzi di frequentare club gratuiti in cui imparare a programmare, realizzare siti web, app, giochi e confrontarsi con altri ragazzi e volontari esperti e appassionati.

1.8.2 Strumenti

Moltissimi sono gli strumenti, hardware e software, che permettono lo sviluppo del pensiero computazionale. Ne elenchiamo alcuni dei più famosi, per dare un'idea delle diverse tipologie e ambiti in cui si collocano.

Computer Science Unplugged CS Unplugged [24] è una collezione libera di attività di apprendimento che insegnano l'informatica tramite giochi sfidanti e puzzle da risolvere usando carte, corde, matite, pastelli o il proprio corpo. Si introducono gli studenti a concetti quali numeri binari, algoritmi, compressione di dati, procedure, ma senza le complicanze dovute all'uso

del computer. Le attività sono adatte a tutte le età ma particolarmente consigliate per la scuola primaria e secondaria inferiore.

Scratch Scratch [12] è un linguaggio di programmazione visuale (la programmazione avviene unendo tra loro "blocchetti" colorati che rappresentano le istruzioni), che permette di realizzare storie interattive, giochi e animazioni, ma è anche una community con cui condividere le proprie creazioni ed esplorare quelle degli altri membri.

Scratch è stato creato dal MIT Media Lab, che ha anche sviluppato un framework per studiare e valutare lo sviluppo del pensiero computazionale [31]. In particolare viene mostrato come Scratch permetta di sviluppare, in spirito costruzionista, il pensiero computazionale nelle tre dimensioni in esso identificate:

- **concetti computazionali**: i concetti che gli sviluppatori utilizzano quando programmano (sequenze, eventi, parallelismo, condizionali, operatori, dati);
- **pratiche computazionali**: pratiche che si apprendono programmando (essere incrementali e iterativi, testing e debugging, riuso e remixing, astrazione e modularizzazione);
- **prospettive computazionali**: i modi di vedere il mondo e se stessi che i ragazzi sviluppano grazie alla programmazione (esprimersi, connettersi, farsi domande).

Scratch è inoltre utilizzato come strumento in un curriculum, sempre del MIT [30], per un corso di "Informatica Creativa", che ha lo scopo specifico di sviluppare il pensiero computazionale con attività legate all'arte, alla creazione di storie e di videogiochi, realizzate sia con attività di gioco "unplugged", sia con Scratch.

Python Molte delle attività proposte da Google per il pensiero computazionale richiedono l'utilizzo di Python.

Python [9] è un linguaggio di programmazione che supporta paradigmi come object oriented, imperativo e funzionale, fortemente tipizzato, con librerie estremamente ricche e versatili. Come scelta di design, è un linguaggio intuitivo, con sintassi snella e pulita. Si tratta di un linguaggio semi-interpretato e dunque permette una rapida produttività.

Altri strumenti Tra le altre attività proposte per insegnare il pensiero computazionale ce ne sono molte legate alla robotica (si pensi ad esempio a Bee-Bot, ape robotica pensata per la scuola materna e primaria) o all'elettronica (Raspberry Pi [10], Arduino [2]), alla creazione di applicazioni (MIT App Inventor [8]), alle animazioni 3D (Alice [1]) o alla realizzazioni di modelli e simulazioni scientifiche.

1.9 Limiti della ricerca attuale

Una recente review della letteratura [143] identifica una serie di punti deboli nella ricerca sul pensiero computazionale svolta finora.

- Dall'articolo seminale della Wing, tuttora la comunità sta discutendo su quale sia una definizione concreta di pensiero computazionale. Finché non si raggiungerà una definizione accettata e precisa di cosa sia, ogni autore darà la sua definizione per spiegare meglio al lettore quale sia la sua visione.
  Attualmente alcune definizioni sono vaghe e non forniscono esempi soddisfacenti, dando spesso luogo a dubbi e fraintendimenti su cosa il pensiero computazionale sia. Peggio: studi che hanno come obiettivo quello di promuovere skill di pensiero computazionale si basano su tali, vaghe, definizioni.
  Solo uno studio [23] riconosce l'ambiguità come intrinseca del pensiero computazionale, e quindi la necessità di darne una definizione dipendente dal contesto in cui gli studi sono realizzati.

- Il concetto di pensiero computazionale è ancora troppo acerbo perché sia possibile misurarlo come risultato quantitativo di uno studio. Per ora la ricerca che propone interventi per raggiungere il pensiero computazionale fornisce solo "prove aneddotiche" dei risultati, utili per formulare ipotesi da verificare ma non di certo per confermarle.

- Il pensiero computazionale ha ricevuto poche critiche o giudizi, è semplicemente preso come un dato di fatto.

- Gli studi presenti non sono rigorosi e non rispettano gli standard di ricerca in ambiti educativi (informazioni psicometriche, statistiche affidabili, dettagli su interventi, procedure, partecipanti, basi teoriche che guidano lo studio, informazioni per riprodurre o replicare lo studio).

1.10 Critiche

In questi anni il pensiero computazionale ha ricevuto un ampio consenso e la maggioranza degli autori riconosce la sua importanza.

Alcuni però hanno sollevato dei dubbi sulla scelta, ritenuta arrogante, di "insegnare a pensare come gli informatici" [69]. I detrattori sostengono in particolare che i concetti e le pratiche proposte non siano tratti distintivi dell'informatica ma si possano trovare in altre discipline. Ci si chiede quindi se sia il caso di fare spazio, in curriculum "a somma zero" come sono quelli dell'istruzione per-universitaria, per questi concetti [40].

Inoltre è chiaro che, al momento, non esiste un consenso sulla definizione e gli aspetti costitutivi del pensiero computazionale, né sulle modalità con cui insegnarlo (materia generale, argomento di una disciplina o argomento multidisciplinare) [104, 40].

Ci sentiamo di sostenere però, come si evince dalla discussione iniziale di questo capitolo, che l'informatica abbia certamente degli aspetti in comune con discipline quali matematica e ingegneria, ma che porta anche concetti nuovi ed originali (ad esempio: la possibilità di cambiare *sempre* livello di

astrazione, e di far interagire tra loro questi livelli, la possibilità di *eseguire* le soluzioni realizzate [98]). Questi aspetti sembrano mancare nell'istruzione oggi, e, così come riconosciamo l'importanza di insegnare la matematica e le scienze, così dovremmo fare con l'informatica [65].

Un'altra critica può essere vista nell'articolo di Mark Guzdial [66], che, in senso molto propositivo, invita i ricercatori informatici a coordinarsi con gli esperti di educazione (cosa fatta solo in piccola parte, per ora) e sfruttare le scoperte sulla comprensione della computazione da parte dei non-informatici (disponibili ma poco utilizzate) per rendere i linguaggi il più possibile accessibili ad un vasto numero di persone. Rispondere a questo appello è quello che cercheremo di fare nei prossimi capitoli.

Capitolo 2

Imparare a programmare è difficile e non (sempre) intuitivo

> *On two occasions I have been asked, – "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" [...] I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.*
>
> – Charles Babbage (1864)

2.1 Imparare (cosa vuol dire imparare)

L'apprendimento può essere definito come *un cambiamento che avviene nell'individuo per effetto dell'esperienza.* Quale sia l'oggetto di tale cambiamento è oggetto di studio delle discipline che vanno sotto il nome di "Psicologia dell'apprendimento e dell'istruzione" e generalmente di "Scienze dell'educazione".

In particolare, si sono sviluppate negli anni numerose *teorie dell'apprendimento*, legate alle principali scuole di pensiero della psicologia.

Esploriamo, seguendo principalmente un testo universitario di Psicologia [99] e la prima parte di una tesi di dottorato sull'apprendimento della programmazione [133], alcuni aspetti di queste teorie rilevanti per i nostri scopi.

2.1.1 Comportamentismo

Il comportamentismo nasce ai primi del Novecento, con l'ambizioso compito di rifondare lo studio della psicologia su basi scientifiche, bandendo lo studio di concetti quali "mente" o "coscienza", ritenuti troppo vaghi e non osservabili. Il movimento, come indicato dal nome, sceglie di studiare il comportamento *manifesto* delle persone, in particolare come reazione a stimoli ambientali esterni.

Il ruolo dell'ambiente è centrale in questa teoria: si rifiuta totalmente l'innatismo, poiché si ritiene che le esperienze vissute da un soggetto siano sufficienti a giustificare il suo comportamento. Di conseguenza è possibile instaurare i comportamenti desiderati in un soggetto predisponendo un ambiente adatto a creare associazioni "corrette" tra stimoli e risposte dell'individuo.

Concetto chiave del comportamentismo è il **condizionamento** classico, secondo il paradigma *stimolo non naturale* $\rightarrow$ *risposta naturale*. Questo principio riesce a spiegare però solo comportamenti piuttosto semplici.

Skinner introdusse perciò il **condizionamento operante**: ad un comportamento (risposta), inizialmente poco frequente, viene associato uno stimolo (rinforzo positivo) e la frequenza di tale comportamento aumenta molto. Secondo Skinner l'apprendimento segue questo stesso principio, sia che si tratti di contenuti, sia di comportamenti logici, linguistici, operativi.

Le metodologie di insegnamento che seguono questa impostazione non prevedono quindi che la persona debba passare attraverso una fase di errori per apprendere qualcosa. Gli errori possono essere evitati ("sbagliando si im-

para a sbagliare"), e con essi una serie di effetti collaterali della "punizione", se la metodologia di insegnamento è progettata in base a questi metodi.

La didattica proposta da Skinner, chiamata "istruzione programmata", è dunque incentrata sull'insegnamento di piccole unità di sapere (di modo da rafforzare positivamente e senza fraintendimenti in caso di successo), in modo individualizzato (tramite le profetiche "macchine per insegnare"). Per contro, sono criticati i rinforzi negativi, ritenuti capaci di portare lo studente solo a disprezzare la disciplina. Più utile per ridurre la frequenza di un comportamento, secondo Skinner, è l'*estinzione operante*: lo stimolo di rinforzo non viene più emesso. In questo modo il comportamento non viene solo ridotto (come in seguito a punizioni, che però lasciano il comportamento nel repertorio del soggetto) ma lentamente estinto.

Volendo riassumere l'idea di apprendimento secondo il comportamentismo possiamo usare la metafora della **trasmissione della conoscenza** come *trasmissione dell'informazione da un calcolatore ad un altro*: essa non deve subire alterazioni durante il passaggio e la buona riuscita della trasmissione è data proprio dalla uguaglianza tra ciò che è trasmesso e ciò che è ricevuto. La valutazione avviene sulla capacità di riprodurre fedelmente tale conoscenza.

Uno studio [96] mostra come, ancora oggi, molte concezioni sull'apprendimento da parte di insegnati, sia novizi che esperti, sia vicina a una prospettiva comportamentista ("apprendere è come registrare il mondo con una videocamera", "gli studenti sono come spugne che si inzuppano in acqua", "insegnare è come scrivere in un quaderno bianco", eccetera).

Il comportamentismo ha dominato la scena fino agli anni Cinquanta, quando si è cominciato a mettere in dubbio il condizionamento come unica spiegazione del comportamento umano, in particolare a causa degli studi sulle relazioni interpersonali e sugli aspetti sociali delle situazioni di apprendimento. Negli stessi anni prendeva piede un approccio che avrebbe dominato - e domina ancora - la scena, e di cui ci occuperemo estesamente: il *cognitivismo*.

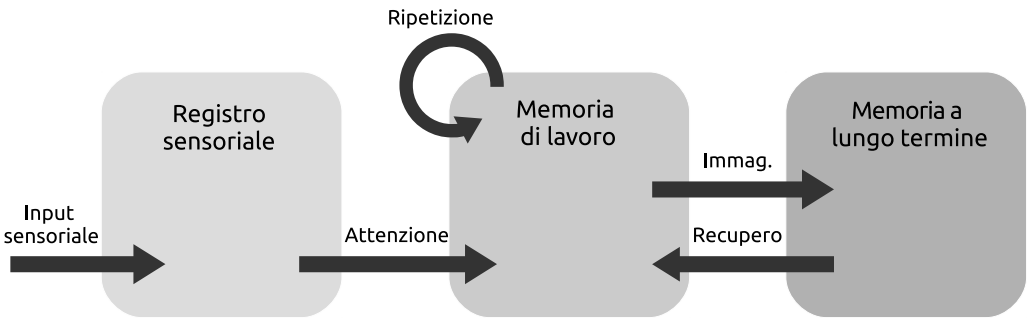

Figura 2.1: Magazzini di memoria, secondo Atkinson e Shiffrin

2.1.2 Cognitivismo

I cognitivisti ripresero lo studio della mente umana tramite inferenze tratte dai comportamenti osservabili.

Il termine "cognitivo" si riferisce, in questo contesto, a tutti i processi di manipolazione dell'informazione (trasformazione, elaborazione, riduzione, immagazzinamento, recupero e combinazione di input sensoriali). Scontata oggi, ma già chiara agli albori dell'approccio, è *l'analogia tra cervello umano e calcolatore elettronico*: l'essere umano, come il computer, processa input e li trasforma in output, memorizzando istruzioni e dati in una memoria temporanea e limitata, ed eventualmente utilizzando una memoria permanente più capiente. Ancora, entrambi presentano limitazioni dovute allo spazio di memoria di lavoro ristretto e alla velocità di elaborazione.

Strutture mentali e sistema di memoria

Larga parte della psicologia cognitiva si occupa di studiare come la conoscenza sia rappresentata nel nostro sistema cognitivo e come questo processi tali **rappresentazioni mentali**.

Centrale è il concetto di "magazzini di memoria", formulato originariamente da Atkinson e Shiffrin [20], che può essere ben schematizzato in Figura 2.1. Tale modello prevede tre "magazzini" attraverso cui transitano gli stimoli sensoriali.

- **Registro sensoriale**: è collegato con l'organo di senso corrispondente, conserva l'informazione per frazioni di secondo (tempo necessario a fare alcuni confronti per riconoscere lo stimolo stesso).

- **Memoria a breve termine o memoria di lavoro**: ha una capienza limitata in numero di elementi (in media, 7 unità) e in tempo (qualche decina di secondi, aumentabile con la ripetizione). Ha la duplice funzione di transito di informazioni che dall'ambiente andranno nella memoria a lungo termine, e viceversa, contenendo informazioni che vengono recuperate dalla memoria a lungo termine per farle interagire con quelle provenienti dall'ambiente. In questa memoria avviene la manipolazione attiva dell'informazione da parte di un soggetto.

- **Memoria a lungo termine**: è un archivio dalle capacità potenzialmente illimitate, memorizza grandi quantità di informazione per tempi molto lunghi. Per la maggior parte del tempo non siamo coscienti di queste memorie: per utilizzarle dobbiamo recuperarle portandole nella memoria di lavoro.

Com'è possibile che l'uomo riesca a svolgere attività mentali complesse con una *memoria di lavoro* tanto limitata? Usando il **raggruppamento** (*chunking*) e **l'automatizzazione**. Raggruppando l'informazione (ad esempio, gruppi di cifre in un numero di telefono) possiamo memorizzarne di più, perché ogni gruppo in memoria di lavoro conta come un'unità. Inoltre, più abbiamo familiarità con un'informazione, più la sua elaborazione avviene in modo automatico (ad esempio, quando leggiamo non pensiamo alle singole leggere, ma alle parole).

Imparare qualcosa significa quindi *conservarlo nella memoria a lungo termine*. Per accedervi, però, dobbiamo passare dal "collo di bottiglia" della memoria di lavoro: per farlo, si deve diventare bravi a elaborare le informazioni in grandi gruppi e in modo automatico.

Strutture di rappresentazione della conoscenza

Il concetto più usato dal cognitivismo per parlare di rappresentazione della conoscenza è quello di **schema**. Uno schema è una *struttura mentale che contiene una conoscenza concettuale generica.* Possiamo avere schemi che ci dicono cosa è una casa, le azioni che di solito servono per mangiare al ristorante, o come scrivere un programma che calcola la media di numeri presi in input.

Gli schemi hanno delle variabili, sono inseriti gli uni negli altri, rappresentano conoscenze a diversi livelli di astrazione. Risiedono nella memoria a lungo termine e non c'è limite al numero degli schemi memorizzabili. Uno schema generico ci permette di fare inferenze su istanze specifiche, senza conoscerle.

Gli schemi sono creati, modificati, estesi tramite l'esperienza. Gli schemi preesistenti giocano un ruolo chiave nel modo di percepire le nuove esperienze (ovvero: cerchiamo di mantenerle coerenti con gli schemi che possediamo) e di integrarle nelle rappresentazioni mentali. In particolare apprendere vuol dire modificare i propri schemi:

- per **accrescimento**, cioè incorporare nuove informazioni in schemi esistenti (es. avere sempre più familiarità con un concetto di cui si incontrano molte istanze);
- per **sintonizzazione**, cioè apportare piccoli aggiustamenti agli schemi, per esempio modificandone alcune variabili per interpretare nuove conoscenze;
- per **ristrutturazione**, cioè attuare la creazione, modifica, cancellazione di schemi vecchi o l'astrazione di schemi, per interpretare nuove informazioni che richiedono tali drastici cambiamenti.

Avere uno schema non significa "averlo corretto". Le **misconcezioni** si formano quando chi apprende integra informazioni nuove in schemi sbagliati o forma nuovi schemi tramite analogie errate.

Sono state elaborate nozioni più specifiche di schema, per esempio gli **script**, ovvero la rappresentazione di una sequenza di eventi che organizza in un ordine temporale una serie di azioni compiute per raggiungere uno scopo (es. tutte le azioni necessarie per prendere l'aereo).

Utili per raggiungere uno scopo in situazioni specifiche sono anche gli **schemi di problem-solving** (es. trovare a in un'equazione del tipo $a/b = c$, con $a, b \neq 0, c$ qualsiasi). Quando ci troviamo di fronte ad un problema del genere riusciamo a individuare un *pattern* più generale e, comunque complesso sia b, sappiamo risolvere l'equazione moltiplicando entrambi i lati per esso. Gli schemi ci permettono *di non essere sopraffatti dai dettagli* nelle situazioni che stiamo vivendo.

In memoria di lavoro uno schema "occupa" un singolo *chunk*, cioè solo una piccola parte, o non ne occupa proprio, se l'azione diviene automatica tramite la pratica.

Gli schemi riescono a spiegare bene anche il raggiungimento dello status di "esperto" in un certo campo. Non si tratta tanto di maggiore memoria di lavoro o intelligenza o talento, ma del fatto che gli esperti hanno una conoscenza di dominio specifica, facilmente accessibile tramite schemi (generali, script, di problem-solving) gerarchizzati, collegati tra loro, astratti. Non a caso gli esperti passano più tempo ad analizzare i problemi qualitativamente, ed in profondità, per usare gli schemi giusti; i novizi, al contrario, giudicano il problema superficialmente e usano schemi inefficienti.

Carico cognitivo

La teoria del **carico cognitivo** sostiene che *si apprende al meglio quando la memoria di lavoro non è né troppo piena né troppo vuota*, e la maggior parte di essa è usata per la formazione di schemi. Il *carico cognitivo*, cioè la quantità di memoria di lavoro utilizzata, può essere suddiviso in *intrinseco, pertinente e estraneo.*

- Il carico cognitivo **intrinseco** è la quantità di memoria di lavoro imposta dallo specifico compito che si sta svolgendo, non può essere ridotto

senza compromettere gli obiettivi di quell'attività.

- Il carico cognitivo **pertinente** è la parte di memoria di lavoro che non è essenziale alla soluzione del compito, ma è necessaria per imparare (ovvero ad accrescere, sintonizzare o ristrutturare schemi, automatizzare schemi); si pensi per esempio alle attività volte a notare similarità e principi comuni in esempi all'apparenza diversi.

- Il carico cognitivo **estraneo** è la parte di memoria di lavoro non essenziale e non necessaria per imparare; pensiamo ad esempio a dati non necessari in un problema, rumore di sottofondo, eccetera.

Si ritiene che per l'apprendimento, sia necessario un *carico cognitivo pertinente elevato*, a scapito degli altri due. Se il carico intrinseco è alto (perché il compito è troppo difficile, o perché mancano gli schemi adatti e dunque c'è un alto numero di elementi da tenere in memoria), qualsiasi carico estraneo sarà eccessivo e non ci sarà spazio per l'apprendimento. Al contrario, se il carico intrinseco è troppo basso, si rischia di non utilizzare (per esempio, per mancanza di motivazione) lo spazio rimasto per apprendere.

Molte sono le implicazioni per la progettazione didattica. Il **problem-solving** è stato particolarmente studiato sotto la luce del carico cognitivo. Tale teoria suggerisce che *cercare di risolvere problemi* non è il metodo migliore per imparare a farlo. La ricerca di informazioni per la risoluzione, unita alla mancanza di schemi risolutivi, impegna moltissima memoria di lavoro, non lasciando spazio all'apprendimento.

Possibili soluzioni sono lo studio di "esempi risolti" - e spesso commentati - da esperti, oppure, per ovviare alla possibile noiosità di tali esercizi, il completamento di problemi parzialmente risolti.

Modelli mentali

I concetti generali presenti negli schemi non sono sufficienti per spiegare come gli esseri umani interagiscono con oggetti e sistemi. Per questo si definiscono i **modelli mentali** come *strutture mentali che rappresentano*

alcuni aspetti dell'ambiente di una persona. Un modello mentale riguarda cose specifiche (es. me stesso, la ragazza di cui sono innamorato, l'ambiente LyX che sto usando ora per scrivere, eccetera) più che concetti generici.

I modelli mentali, creati in parte consciamente e in parte inconsciamente, aiutano a ragionare su come questi oggetti o sistemi del proprio ambiente funzionino in situazioni diverse: studiando i modelli mentali possiamo comprendere come le persone spiegano lo scopo, i meccanismi sottostanti dei sistemi e come predicono gli stati futuri degli stessi.

I modelli mentali aiutano a sentirsi a proprio agio in situazioni complesse, e sono per questo utili. Ma possono essere anche dannosi: modelli mentali poveri o errati possono portare ad errori; nonostante questo diamo loro grande valore e li consideriamo realistici, e, sebbene possano essere sviluppati e corretti tramite la pratica, tendiamo a rimanere attaccati ai nostri modelli più familiari.

I modelli mentali sono *eseguibili*: possono essere usati per ragionare sui sistemi in particolari situazioni, per "visualizzare" come funziona un sistema, per predire comportamenti o stati del sistema passati o futuri, dato un insieme di condizioni iniziali (es. predire la traiettoria di una pallina in un sistema fisico, cosa accadrà se premo il tasto "bold" nel mio editor, il comportamento di un algoritmo che sto progettando). Si parla in questi casi di *simulazione mentale* del sistema. Tale simulazione è spesso *visuospaziale* e avviene nella limitata memoria di lavoro, quindi non può coinvolgere troppi stati o variabili, perciò si tendono a fare simulazioni qualitative più che quantitative.

Non dobbiamo confondere il modello mentale che un utente ha di un sistema (diverso dal modello mentale di chi ha progettato quel sistema e ancora diverso dal modello dell'insegnante che lo spiega) con un *modello concettuale* di tale sistema. Quest'ultimo è una spiegazione del sistema creata apposta dal progettista o dall'insegnante allo scopo di spiegarne la struttura e il funzionamento, per facilitare la formazione di un modello mentale "corretto" di quel sistema. Studi classici mostrano come soggetti a cui viene spiegato

"come funziona" ottengono risultati migliori rispetto a coloro viene spiegato solo "cosa fare" per controllare un sistema.

L'approccio cognitivista domina la ricerca psicologica attuale. Nonostante questo, la sua visione dell'apprendimento, che qui abbiamo solo superficialmente riassunto, e a cui spesso ci si riferisce con la locuzione *Human Information Processing* (HIP), non è esente da critiche. In particolare, vengono sottolineate: l'eccessiva semplificazione data dall'analogia mente umana-computer; l'eccessivo focus sulla conoscenza individuale e libera dal contesto, che non tiene conto gli aspetti sociali, culturali, ambientali dell'apprendimento; la difficoltà di accesso e studio oggettivo di strutture mentali solo ipotizzate. Alcune tra queste questioni sono affrontate dall'approccio costruttivista.

2.1.3 Costruttivismo

Riassumendo in una frase l'idea del paradigma educativo chiamato **costruttivismo** possiamo dire che *le persone costruiscono attivamente la conoscenza, piuttosto che ricevere e memorizzare passivamente conoscenza preesistente.* La conoscenza non viene presa "così com'è" da un mondo esterno, non è la copia di ciò che dice il libro o il professore, bensì è unica rispetto all'individuo (o al gruppo) che l'ha costruita. Il modello educativo proposto è attivo e studente-centrico.

Il termine costruttivismo ha una portata molto ampia, e comprende in realtà molti "costruttivismi" diversi. Il *costruttivismo sociologico* si occupa della conoscenza costruita dalle società, mentre quello *psicologico* si occupa più degli individui, ponendo più enfasi sulla costruzione collaborativa (*socio-costruttivismo*) o dal singolo (*costruttivismo personale*). Per avere un'idea del variegato panorama, elenchiamo una serie di affermazioni costruttiviste, ben riassunte in [133].

- La conoscenza è costruita da chi apprende, non è (e non può essere) trasmessa così com'è.

- La conoscenza che gli individui - o i gruppi - possiedono è diversa dalla conoscenza di altri individui che hanno apparentemente "imparato la stessa cosa".

- La costruzione della conoscenza avviene tramite l'interazione tra la conoscenza precedente e le nuove esperienze.

- La conoscenza non riguarda, rappresenta o deriva da una realtà esterna e indipendente dall'osservatore, ma da una "realtà" esperienziale, personale (o condivisa col gruppo).

- Non c'è nulla di oggettivamente corretto o sbagliato, vero o non vero, esiste solo conoscenza che è più *adatta* a un determinato obiettivo.

- Il contesto culturale e sociale media la costruzione della conoscenza.

- L'apprendimento effettivo ha come protagonista il discente come costruttore attivo di conoscenza; un buon insegnamento significa facilitare e motivare tale costruzione.

- Far leva sulle conoscenze pregresse è un buon metodo di insegnamento.

- Agli studenti dovrebbero essere presentate un numero minimo di informazioni, permettendo loro di scoprire principi e regole da soli.

- Un'esplorazione pratica, gestita dallo studente è un metodo efficace per imparare.

- Un apprendimento efficace richiede situazioni di apprendimento complesse, realistiche e autentiche. Gli studenti dovrebbero risolvere problemi mal-strutturati, aperti, che rispecchiano quelli reali risolti dagli esperti.

- Un apprendimento efficace ha una dimensione sociale: il lavoro di gruppo è importantissimo.

- L'educazione non dovrebbe imporre obiettivi indipendenti dallo studente.
- Non è possibile - non ha senso - applicare metodi di valutazione standard per valutare l'apprendimento.

Tramite i lavori seminali di Jean Piaget e Lev Vygotsky, il costruttivismo è diventato predominante nelle teorie dell'educazione durante il Novecento. Ai giorni nostri, in particolare, è impossibile trovare uno studioso di Educazione che non si riconosca almeno in parte nelle affermazioni precedenti (tutti, ad esempio, riconoscono che imparare non può essere ridotto alla mera trasmissione di conoscenza). Tutti possiamo dunque chiamarci *costruttivisti*, tutto sta in quanta enfasi poniamo sull'idea di conoscenza come costruzione.

La conoscenza è una costruzione (inter-)soggettiva

Due idee importanti della visione costruttivista dell'apprendimento sono le seguenti:

- l'apprendimento si costruisce sulla base della conoscenza esistente e degli interessi che il discente porta nel contesto educativo;
- l'apprendimento è la costruzione di nuove idee tramite l'interazione tra la conoscenza esistente e nuove esperienze.

Si tratta di idee molto generiche: alcuni costruttivisti non cercano di essere più specifici, mentre altri si rifanno alle idee cognitiviste (es. formazione di schemi) per giustificare queste affermazioni.

Più nel dettaglio, esistono alcune teorie legate chiamate *teorie del cambiamento concettuale*, che studiano come cambiano le strutture concettuali dei soggetti durante l'apprendimento. In particolare si interessano delle concezioni ingenue (naïve) che emergono dall'esperienza di tutti i giorni e che sono molto resistenti al cambiamento.

Le "famiglie" di pensiero sono due. La prima, che possiamo definire "**conoscenza come teoria**", sostiene che la conoscenza, se pur ingenua, sia

organizzata e coerente, applicabile in contesti diversi. Quando lo studente si rapporta a qualcosa di nuovo, cerca di metterlo in relazione alle idee già esistenti e a giudicare la sua coerenza con esse. Apprendere un concetto implica cambiarne molti altri, e questo è un compito arduo: per questo le misconcezioni sono difficili da sradicare. Perché il *cambiamento concettuale* [117] avvenga, bisogna stimolare un *conflitto cognitivo*: il soggetto non deve essere più soddisfatto della concezione precedente, deve ritenere quella nuova plausibile e utile in un futuro.

Diametralmente opposta sul piano della *coerenza delle teorie* è la famiglia della "**conoscenza come elementi**", che vede invece la conoscenza naïve come una collezione quasi indipendente, frammentata, molto contestualizzata di elementi con una connessione lasca. Una nuova idea può quindi convivere con altre che sono contraddittorie, semplicemente perché ognuna è coerente in un particolare contesto; sarebbe proprio questo che rende le misconcezioni difficili da sradicare. Provando a farlo si otterrebbe l'effetto opposto: aggiungere una comprensione "parallela" in un contesto diverso. I principali sostenitori dei questa seconda visione *non ritengono le misconcezioni un problema*, bensì utili e, a volte, necessarie per l'apprendimento futuro [131]. Infatti, non essendoci a loro parere nessuna teoria generale da modificare, l'obiettivo dell'educazione deve essere piuttosto quello di aiutare a organizzare meglio gli elementi di conoscenza, per esempio trovando il giusto grado di astrazione e generalizzazione, di similarità tra contesti.

La verità non esiste

Più chiara, sebbene con diverse (e distanti) sfumature è l'idea di *verità* del costruttivismo. L'**ontologia** (cioè la teoria dell'esistenza) **comune** afferma che esiste un "mondo reale" indipendente dalla nostra comprensione di esso. L'**epistemologia** (cioè la teoria della conoscenza - individuale e collettiva) **comune** suggerisce che la conoscenza riflette la realtà esistente, e dunque il valore di verità di un'affermazione dipende dalla sua corrispondenza con tale realtà.

Per il costruttivismo invece la realtà ontologica è al più irrilevante (se esiste): la realtà va intesa epistemologicamente. La conoscenza non è una copia fedele e oggettiva di una realtà ontologica, ma è una costruzione della mente dell'individuo (o del gruppo) basata, inevitabilmente, sul suo vissuto e sulla conoscenza già costruita. Si tratta di conoscenza non fondazionale (cioè da non prendere come verità assoluta su cui basarsi) e di conseguenza *imperfetta e fallibile*. Per questo il costruttivismo preferisce evitare di parlare di "corretto" o "sbagliato", "vero" o "falso", parlando piuttosto di **applicabilità** (*viability*): un'azione, operazione, struttura concettuale e applicabile se è utile (secondo il soggetto) a raggiungere un obiettivo.

La conoscenza inoltre *dipende dal contesto* in cui è stata costruita.

Queste affermazioni, se pur effettuate con diversi gradi di radicalismo, portano ovviamente a opporsi al modello di educazione tradizionale, in particolare contestando: il primato dell'insegnante (che dovrebbe essere solo uno degli attori nella costruzione della conoscenza), la scelta degli obiettivi educativi (che non possono essere gli stessi per ogni studente, ma che dovrebbero invece emergere da essi, dai loro interessi, dall'interazione con gli altri), il ruolo della valutazione (in quanto la conoscenza non è univoca, dipende dal contesto, dal singolo, dallo scopo).

Collaborare in contesti autentici e in una comunità

Un trend generale delle idee costruttiviste enfatizza ambienti di apprendimento che siano strutturati come contesti complessi, realistici o reali, che richiedano agli studenti di risolvere problemi mal strutturati (perché sono quelli che incontreranno nella vita). Si tratta di forme di *apprendimento per scoperta* (gli studenti scoprono da soli le soluzioni) e di *apprendimento per problem-solving* (gruppi di studenti risolvono problemi realistici con aiuto limitato da parte dei docenti).

Una branca del costruttivismo enfatizza in particolare l'apprendimento come partecipazione in una comunità: si tratta del cosiddetto *situated learning*. Lo studente partecipa a una comunità, non solo con ruoli di osservazio-

ne e imitazione, ma essendone parte attiva, collaborativa, utile, prendendo familiarità con le attività, col "cosa" e col "come" e passando da un ruolo marginale, periferico, ad uno centrale, acquisendo esperienza e, possibilmente, diventando un esperto.

2.2 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 [54, 87].

Analizziamo nei prossimi paragrafi alcuni aspetti, pedagogici e psicologici, che si ritiene siano alla base di tali difficoltà.

2.2.1 Gli obiettivi 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).

Nella tradizione pedagogica, il lavoro più influente e acclamato riguardo agli obiettivi cognitivi è quello di Bloom del 1956 [26]. In tale opera viene presentata una gerarchia di obiettivi di apprendimento classificati a seconda della loro complessità cognitiva. Dal più basso al più alto, essi sono:

- **conoscenza**: lo studente ricorda e può ripetere fatti o metodi;
- **comprensione**: lo studente capisce il significato di fatti o metodi;
- **applicazione**: lo studente può risolvere problemi applicando conoscenza a situazioni concrete nuove;
- **analisi**: lo studente può scomporre le informazioni nelle sue parti per determinare motivi o cause, o per fare inferenze;

- **sintesi**: lo studente può combinare elementi in nuovi modi per produrre nuovi "mondi";
- **valutazione**: lo studente può fare giudizi sulla base di alcuni criteri.

La tassonomia è stata rivisitata da due allievi di Bloom [86] per renderla più attinente alle recenti scoperte sull'apprendimento (in particolare rilassando un po' il vincolo gerarchico, in quanto alcuni obiettivi possono essere raggiunti in contemporanea o in ordine inverso - ad esempio, prima valutazione e poi sintesi).

La tassonomia rivista di Bloom è anche utilizzata dall'ACM e dall'IEEE nelle loro indicazioni per i curriculum informatici [75, 137].

Osservando la tassonomia ci si rende conto, anche solo intuitivamente, di quanto gli obiettivi dei corsi di Introduzione all'Informatica siano impegnativi dal punto di vista cognitivo: scrivere un programma che risolve un problema espresso in linguaggio naturale può collocarsi nella parte più alta della tassonomia (*sintesi*). Analisi effettuate con altre tassonomie [29] portano a considerazioni analoghe.

2.2.2 Schemi e carico cognitivo

Nell'informatica, come in altri campi, l'esperienza è data dalla elevata conoscenza di soluzioni generiche che facilitano la risoluzione di problemi complessi.

I lavori più influenti in questo campo sono quelli di Elliot Soloway [132] che mettono in relazione la teoria degli schemi (§ 2.1.2) con la programmazione. Consideriamo il seguente esempio:

```
media = 0
somma = 0
for (i=0; i<len(valori); i++)
    somma = somma + valori[i]
media = somma / len(valori)
```

Un novizio applicherà una serie di *schemi*: quello per *inizializzare le variabili*, quello per *scorrere tutti gli elementi di un vettore*, quello per *effettuare un assegnamento*, quello per *calcolare una divisione*, quello per *usare una funzione di libreria*, eccetera. Si tratta di schemi a livelli di astrazione diversi, che possono essere concatenati, annidati e uniti.

Evidenze empiriche [132, 124] mostrano che questi schemi costituiscono i *chunk* cognitivi presenti in memoria di lavoro nella mente. Per uno studente alle prime armi risolvere un problema del genere richiede un grande sforzo, perché tiene in memoria, separatamente, tutti quegli schemi. Con l'esperienza, riesce a costruire uno schema più astratto, più generale quale "*calcolare la media degli elementi di un array*", che ora costituirà un solo *chunk* nella sua memoria di lavoro.

Anche la *teoria del carico cognitivo* può spiegare alcune difficoltà di apprendimento della programmazione. Linn e Clancy [91] mostrano come gli alunni che abbiano studiato *commenti di esperti per la risoluzione di problemi* riescano meglio nei successivi compiti di programmazione. Questo tipo di attività non è attualmente molto in voga nei corsi introduttivi.

Molti autori (es. [53]) sottolineano come le sfide affrontate quando si programma (notazione, dinamica a runtime, la necessità di rappresentarsi mentalmente programmi e dominio, tool di programmazione, problem-solving) generino un carico cognitivo intrinseco inusuale ed elevato.

2.2.3 Misconcezioni

Avere una **misconcezione** significa credere di conoscere (come funziona) qualcosa, ma in realtà lo si è "compreso male", cioè se ne ha un'interpretazione dannosa o non desiderabile.

Come già detto, a seconda del proprio punto di vista sull'educazione, si possono vedere le misconcezioni come errori da affrontare e correggere oppure come materiale grezzo che può evolvere in un una migliore comprensione. Ad ogni modo, è utile promuovere il *conflitto cognitivo*, cioè la distanza tra quanto osservato dallo studente e la sua comprensione corrente [130].

Nel corso degli ultimi decenni, sono stati catalogati dai ricercatori i modi con cui gli studenti di programmazione mispercepiscono i concetti fondamentali, e le loro comprensioni incomplete e scorrette. I concetti più problematici che vengono segnalati sono: le variabili, l'assegnamento, i riferimenti e i puntatori, classi, oggetti, costruttori e ricorsione, oltre a problematiche generali riguardanti le "capacità" e il funzionamento di una macchina che esegue i programmi [133].

Una lunga lista, presentata in Appendice A, comprende una vasta scelta di misconcezioni per ognuno di questi ambiti. Leggerla fa trasparire in modo netto ed evidente ciò che emerge in modo unanime dalla letteratura: gli studenti hanno enormi difficoltà nel comprendere i concetti fondamentali della programmazione. Se è vero che tali modi "sbagliati" di percepire i concetti non sono completamente inutili (possono andar bene in certi contesti), non sono generalizzabili e dunque destinati a causare, prima o poi, problemi ai programmatori e ai programmi da loro scritti.

Secondo Clancy [39] sono due le macro-cause di misconcezioni: *sovra- o sottogeneralizzazione* e *confusione nel modello computazionale*. I linguaggi ad alto livello forniscono infatti l'astrazione sul controllo e sui dati, permettendo una programmazione più semplice e potente, ma, di contro, nascondono dettagli della macchina esecutrice all'utente, il quale può di conseguenza trovare misteriosi alcuni costrutti e comportamenti.

Elenchiamo e discutiamo alcune cause di misconcezioni individuate, divise in categorie.

Inglese. Le parole chiave di un linguaggio non hanno lo stesso significato in Inglese e nella programmazione. Per esempio, la parola *while* in inglese indica un test costantemente attivo, mentre il costrutto `while` può testare di nuovo la condizione solo all'inizio dell'iterazione successiva. Alcuni studenti ritenevano che il loop terminasse nel preciso momento in cui la condizione si fosse falsificata [135, 27]. Analogamente alcuni pensavano al costrutto `if` come a un test continuamente attivo e in attesa del verificarsi di una condizione, e che il ramo `then` venisse eseguito non appena la condizione

diventasse vera [112].

Sintassi. Sebbene si possa pensare che sia una delle maggiori fonti di misconcezioni, gli studi mostrano come sia un problema solo nelle primissime fasi. In particolare gli studenti sarebbero in grado di scrivere programmi sintatticamente validi, anche se non utili alla risoluzione del problema dato o semanticamente scorretti [126, 135, 80, 129].

Notazione matematica. Riportata da molti autori, classica è la confusione che genera l'assegnamento con il simbolo `=` (per esempio, visto come un'equazione o come uno swap di valori tra le variabili [53]) o l'incremento (`a = a + 1`) pensato come equazione impossibile.

Esperienze precedenti di programmazione. Vengono evidenziate difficoltà a cambiare paradigma (es. passare da Scheme a C).

Sovrageneralizzazioni di esempi. Fleury [56] elenca una serie di vincoli inesistenti (metodi in classi diverse che devono avere nomi diversi, argomenti che possono essere solo numeri, operatore "punto" usabile solo nei metodi) dettati dal fatto che gli studenti non avessero visto nessun controesempio per tali situazioni.

Analogie. L'analogia "una variabile è come una scatola" può far scaturire l'idea che - come una scatola - possa contenere contemporaneamente più elementi [53]. L'analogia "programmare col computer è come conversare con esso" può portare ad attribuire *intenzionalità* al computer e dunque a pensare che esso [112]:

- abbia un'intelligenza nascosta che comprende le intenzioni del programmatore e lo aiuta a raggiungere il suo scopo ("superbug");

- abbia una visione generale, conosca cioè anche cosa accadrà alle linee di codice che non sta eseguendo in quel momento.

Alcuni aspetti della programmazione sono particolari portatori di misconcezioni.

Sequenza. Molte misconcezioni sono dovute alla mancata comprensione del flusso del programma: tutte le linee attive allo stesso momento [112],

parallelismo "magico", ordine delle istruzioni non importante, difficoltà a capire i branch in casi di `while` o `if` [53].

Passaggio di parametri. Gli studenti presentano difficoltà in questo ambito, per esempio confondendo le tipologie di passaggio (valore, riferimento...), facendo confusione col valore di ritorno o con lo scope dei parametri. Ci sono prove di studenti che riuscivano a produrre codice funzionante pur avendo gravi misconcezioni sui parametri [55, 95].

Input. Gli statement di input sono particolarmente problematici. In particolare gli studenti non capiscono da dove i dati di input provengano, come sono memorizzati e resi disponibili al programma. Alcuni credono che un programma ricordi tutti i valori associati a una variabile (la sua "storia") [67].

Allocazione di memoria. Ci sono notevoli difficoltà a comprendere il modello di memoria di Java o C++, ad esempio, poiché l'allocazione avviene implicitamente. Alcuni studenti per esempio credono che se si chiama un metodo che setta gli attributi, il costruttore non serva [56].

Object oriented. Possiamo vedere il paradigma ad oggetti come un'estensione della programmazione imperativa, che si porta dietro tutte le misconcezioni già viste, con l'aggravio di molte altre generate dalla sovrastruttura che linguaggi come Java aggiungono. Nell'Appendice A molto spazio è dedicato a tali misconcezioni, su cui la letteratura si è concentrata negli ultimi anni.

2.2.4 "Macchine concettuali" e tracing

Con *notional machine* (personalmente tradotta come "macchina concettuale"), locuzione usata per la prima volta in [53], si indicano le proprietà generali della macchina che si sta imparando a controllare. In altre parole, si tratta di *un computer idealizzato, le cui proprietà sono implicate dai costrutti del linguaggio di programmazione impiegato, ma che possono essere esplicitate durante l'insegnamento.*

Lo scopo di questa astrazione è quello di spiegare l'esecuzione dei programmi. La macchina concettuale[1] è dunque una caratterizzazione del computer nel suo ruolo di esecutore di programmi (in un particolare linguaggio), racchiude le capacità e i comportamenti hardware e software che siano sufficientemente astratti, ma precisi, per spiegare come i programmi sono eseguiti [134].

La macchina concettuale è legata a un preciso linguaggio di programmazione e può essere più o meno "machine like" (se basata, per esempio, sul lambda calcolo). Non deve (necessariamente) essere un modello del computer fisico.

Ci possono essere più macchine, a livelli di astrazione diversi, per uno stesso linguaggio (per esempio, una macchina per Java che definisce il computer come capace di maneggiare oggetti che si scambiano messaggi, una a più basso livello che vede i vari costrutti maneggiare aree di memoria, stack delle chiamate e heap, una ancora più a basso livello che descrive Java in termini di bytecode).

Come già in parte evidenziato, la letteratura sulle misconcezioni ha messo in luce, tra gli altri, il ruolo delle macchine concettuali nella formazione dei problemi di comprensione. Molte misconcezioni infatti hanno a che fare con aspetti che non sono visibili, ma nascosti nel *mondo a tempo di esecuzione* (es. riferimenti, oggetti, incremento automatico delle variabili di controllo dei cicli). Queste difficoltà possono essere imputate al fatto che gli studenti non riescono a comprendere ciò che sta accadendo ai programmi in memoria, poiché non riescono a formarsi un modello mentale della loro esecuzione.

La formazione di un modello mentale (della macchina concettuale e del programma) permette al programmatore di fare inferenze sul comportamento del programma e prevederne i cambiamenti futuri.

[1] Si tratta di una nozione più specifica, perché con scopi didattici, della nozione classica e generale di **macchina astratta**. Un ottimo testo in italiano per approfondire questo concetto e avere una panoramica dettagliata dei principi che guidano la progettazione e l'implementazione dei linguaggi di programmazione è [58].

La *teoria cognitivista* (§ 2.1.2) afferma che le persone usano analogie basate solo sulle caratteristiche superficiali per formarsi modelli mentali di nuovi sistemi. Per questo gli studenti utilizzano i costrutti del linguaggio di programmazione (l'unica cosa che possono "vedere") per formarsi modelli mentali della macchina concettuale, facendo inferenze sul comportamento di essa. Spesso però i linguaggi sono, in superficie, simili al linguaggio naturale o ad altri mondi incontrati dallo studente, quali la matematica: questo è spesso fonte di misconcezioni.

I novizi, come è facile immaginare, hanno modelli mentali incompleti, non scientifici, carenti, non ben definiti nei loro limiti. Spesso capita che abbiano modelli multipli e contraddittori per situazioni invece uniformi (es. assegnamento di interi o di record). Gli esperti, al contrario, hanno modelli mentali più stabili, astratti e accurati, basati su principi generali. L'educazione deve facilitare l'evoluzione dei modelli mentali e portare il prima possibile lo studente a formali correttamente, poiché cambiare modelli già acquisiti è un compito cognitivamente difficile.

Si ritiene che i modelli mentali siano *eseguibili* (§ 2.1.2), dunque le persone possono usarli per ragionare su un sistema in una certa situazione. Vista la capacità limitata della memoria di lavoro, per risolvere un problema con successo è cruciale simulare un sistema al livello di astrazione giusto per lo scopo (troppo astratto significherebbe non risolvere il problema, troppo concreto significherebbe essere sopraffatti dai dettagli).

Nel contesto della programmazione, la simulazione dell'esecuzione è detta **tracing**. Si tratta di una skill essenziale dei programmatori esperti, che la usano durante il design, la comprensione di altri programmi, la scrittura e il debug [132].

Tecnicamente il tracing potrebbe essere visto come "l'esecuzione di un modello mentale del programma mentre si esegue un modello mentale della macchina concettuale". Molti ritengono però che nella nostra mente avvenga una *crasi* di questi modelli, infatti spesso diciamo "il programma fa X" [53].

Gli esperti usano un tracing simbolico, ad alto livello, mentre i novizi sono

obbligati ad usare più spesso il tracing "concreto", con i valori esplicitati. Purtroppo alcuni studi (es. [93]) mostrano come l'abilità degli studenti di fare tracing sia molto bassa, anche dopo uno o due corsi di programmazione. Inoltre gli studenti la ritengono una attività inutile o difficile [113].

Costruttivismo, Informatica, Macchine Concettuali

Due lavori sono particolarmente influenti nel campo della ricerca nell'educazione informatica costruttivista, uno di Greening (2000) [64] e l'altro di Ben-Ari (2001) [25].

La proposta di Greening è un costruttivismo forte e radicale: la vera sfida non è l'insegnamento di linguaggi, sintassi e semantica, ma piuttosto vedere la programmazione come un processo creativo di sintesi e problem solving. Propone quindi un *apprendimento basato sul problem-solving* e la risoluzione di problemi realistici in contesti autentici.

Ben-Ari propone un costruttivismo più personale, moderato e "cognitivista", focalizzandosi sull'importanza della conoscenza pregressa nell'apprendimento della programmazione. In particolare sottolinea come *lo studente che incontra un nuovo sistema costruisca un modello mentale di esso*, in modo inevitabile, *indipendentemente* dal fatto che questo modello venga o meno insegnato esplicitamente. La letteratura sulle misconcezioni mostra come questo modello "autogenerato" sia incompleto e non sostenibile.

In secondo luogo, Ben-Ari mette in evidenza il fatto che, sebbene il costruttivismo rifiuti la realtà ontologica del mondo, o la ritenga inaccessibile, il computer formi proprio una "realtà ontologica accessibile". Il computer infatti è un artefatto che si comporta in un modo ben preciso e finché la comprensione dello studente (se pur personale, unica e costruita) non è sufficientemente simile a tale realtà, l'apprendimento risulterà ostico.

Greening critica[2] la centralità del modello mentale proposta da Ben-Ari, in quanto ritiene che i livelli di astrazione nei linguaggi di program-

[2]Non disponendo di una sfera di cristallo, ma semplicemente di una versione preliminare del lavoro di Ben-Ari.

mazione stiano continuamente salendo, rendendo sempre meno importante comprendere il funzionamento a basso livello della macchina esecutrice.

Ben-Ari ribatte sostenendo che, a qualunque livello di astrazione si lavori, è comunque indispensabile comprendere il funzionamento del livello "immediatamente inferiore", per spiegare i fenomeni in cui ci si imbatte. Il ruolo della macchina concettuale sarà dunque importante anche per i programmatori del futuro.

I modelli mentali sono davvero centrali?

Secondo Sorva [133] la capacità degli studenti di comprendere la macchina concettuale è centrale nella soluzione dei problemi finora elencati, in quanto:

- previene e corregge numerose misconcezioni;
- serve come base per la formazione di schemi di basso livello e, iterando, quelli - caratteristici degli esperti - di alto livello;
- di conseguenza, indirettamente, riduce il carico cognitivo, poiché gli schemi di problem solving che si formano sono più complessi e astratti;
- è centrale nella capacità di fare tracing dei programmi, che permette una migliore comprensione dei programmi e facilita il debug.

Molti autori (es. [138, 27, 112, 61, 80]) concordano in tutto o in parte con questa tesi. Alcuni esperimenti sono stati svolti [28, 36], ma i risultati sono ancora poco risolutivi.

2.3 Imparare a programmare non è sempre intuitivo

Fino ad ora ci siamo occupati di studenti che apprendono (poco) a programmare. Avendo in mente però lo scopo più ampio di estendere, almeno

in parte, 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 "portano con sé", "hanno già" prima di approcciarsi alla programmazione.

2.3.1 Il modo naturale di esprimere le istruzioni

Un lavoro del 2001 di Pane e altri [108] prende atto della difficoltà che la programmazione comporta, sottolineando come il design dei linguaggi di programmazione non abbia tenuto conto di questioni di *interazione persona-computer*. In particolare i programmatori devono pensare ai programmi e ai dati in un modo molto diverso da quello che per loro sarebbe *naturale*. Proprio questa distanza rende difficile ai principianti imparare a programmare, e richiede un inutile sforzo anche per gli esperti. Per questo gli autori hanno effettuato due studi, concentrandosi su soggetti *che non avevano mai programmato prima*, sia bambini che adulti.

Ai bambini (10-11 anni, equamente distribuiti per sesso e di etnie diverse) è stato chiesto di scrivere su dei fogli, usando liberamente parole e/o disegni, cosa "direbbero" al computer e come, per fare in modo che esso realizzi il comportamento di PacMan, dopo che un video (con poche istruzioni testuali, per non influenzare le risposte) sul funzionamento del gioco era stato loro mostrato.

Un secondo studio ha coinvolto giovani adulti (studenti e docenti) della Carnegie Mellon University (18-34 anni) e un altro gruppo di bambini analogo al precedente. Ad entrambi i gruppi sono stati forniti una serie di problemi di manipolazione di un database di nomi e valori numerici, chiedendo ancora una volta di scrivere su fogli bianchi come esprimerebbero (con le proprie parole o con disegni) le istruzioni da dare al computer perché risolva tali problemi.

Analizziamo i risultati, divisi nelle categorie individuate dagli autori.

Stile di programmazione Le istruzioni sono per la maggior parte *regole di produzione* o *basate sugli eventi* e iniziano con *if* o *when*. Vengono osservati comunque, in misura molto minore, vincoli, statement dichiarativi e imperativi.

Si notano soluzioni più vicine ai sistemi reattivi, con scarsa attenzione per il flusso di controllo globale. Se è vero che anche la programmazione imperativa possiede il costrutto `if`, questo deve essere raggiunto dal flusso di controllo.

Lo stile imperativo viene utilizzato soprattutto per descrivere il flusso di controllo locale. Lo stile dichiarativo invece è utilizzato soprattutto per "fare il setup dello scenario".

Prospettiva La maggioranza dei partecipanti ha assunto la prospettiva del giocatore o dell'utente, oppure si è immedesimato in uno dei personaggi del gioco, oppure una ha optato per una prospettiva in terza persona. Infrequenti i casi in cui la prospettiva sia stata quella del programma.

Operazioni su oggetti multipli e cicli Nei linguaggi di programmazione più popolari bisogna iterare su una collezione per effettuare operazioni sui suoi elementi. I partecipanti hanno invece preferito molto usare insiemi e sottoinsiemi (pensiamo ad esempio al costrutto `foreach`), oppure semplicemente esprimendo le istruzioni al *plurale*.

Questo elimina molte situazioni in cui è necessario l'uso dei cicli. Le poche volte che sono stati utilizzati, sono stati terminati con termini quali "until", "and so on...".

Condizionali complessi e `not` Invece di usare condizionali complessi, i partecipanti usano una serie di regole semplici mutuamente esclusive.

Preferiscono "`se A allora fai qualcosa a meno che B`" invece che "`se A e non B allora fai qualcosa`". Il costrutto `try...catch` può essere un buon meccanismo per andare incontro a questo stile.

In generale si fa poco uso di negazioni (in accordo con studi che mostrano come è più difficile esprimere concetti negativi [142]). Quando il `not` è utilizzato, ha precedenza bassa (in contrasto con quanto avviene solitamente nei linguaggi).

Operazioni matematiche I bambini hanno espresso le operazioni matematiche in linguaggio naturale. Solo pochi adulti hanno scelto una notazione matematica (es. $i + 3$) o addirittura informatica (`i = i+3`). Alcune espressioni matematiche non avevano esplicitato su quale valore operare o l'ammontare dell'operazione.

Specificare intervalli I bambini non hanno usato notazione matematica, circa la metà degli adulti sì. Chi non ha usato notazione matematica ha specificato spesso intervalli che si sovrapponevano, o ha usato termini ambigui sul fatto che fossero inclusivi o esclusivi. Chi ha usato la notazione matematica è stato accurato al 100%.

Ricordare lo stato Non sono state usate variabili per ricordarsi il progresso in un task. Piuttosto si trovano espressioni del tipo "quando tutti / quando nessuno... allora il task è finito"

Non vengono in generale usate le variabili, se si vuole usare un'informazione precedente per una decisione attuale, o un'informazione attuale per una decisione futura, i partecipanti parlano 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. Inoltre *and* veniva usato anche quando serviva *or* ("90 and above"), e ciò può essere spiegato dal fatto che "and" si usa in tante situazioni in inglese, in modo inconsistente dall'operatore.

Or e *but* appaiono poco (e d'altra parte è più difficile esprimere espressioni disgiuntive), ma quando vengono usati lo si fa in modo corretto.

`Then` ed `else` Il termine `then` viene usato per dire "dopo", quindi come operatore di sequenza, e non per dire "allora" come intendono i linguaggi di programmazione.

La clausola `else` non è mai presente.

Operazioni di inserimento, cancellazione, ordinamento Pochi hanno considerato problemi di memorizzazione di strutture come gli array quando si tratta di inserimenti o cancellazioni, mostrando di pensare piuttosto a strutture dati di tipo lista.

L'ordinamento viene preso come operazione elementare già disponibile.

Object oriented Alcuni aspetti del paradigma Object Oriented sono stati trovati, in particolare le entità sono trattate come *aventi uno stato* e *rispondenti a richieste di azioni*. Al contrario non sono stati trovati aspetti quali ereditarietà e polimorfismo.

Uso di disegni Due terzi dei partecipanti usa diagrammi per esprimere le loro soluzioni, soprattutto all'inizio, per specificare setup e layout.

Uno studio del 2010 di Good e altri [61] si propone di espandere il focus dello studio di Pane, ponendo attenzione a una delle capacità riconosciute centrali nel pensiero computazionale, ovvero "l'abilità di definire istruzioni chiare, specifiche e non ambigue per eseguire un processo". Si chiede di specificare tali regole in linguaggio naturale, di modo da rimuovere la possibilità di errori dovuti alla sintassi e quindi di analizzare gli aspetti semantici, con focus particolare sugli errori dei giovani programmatori.

Anche in questo caso i soggetti sono ragazzi e ragazze di 11-12 anni, fatti giocare a un famoso gioco di ruolo, presentando loro alcune situazioni create con l'editor del gioco, e chiedendo in seguito di scrivere su un foglio le regole che potessero aver causato i comportamenti osservati mentre giocavano.

Analisi delle descrizioni La struttura delle regole era per la maggior parte "event based", risultato in parte atteso e dovuto alla struttura del gioco. Molto più bassa in percentuale, viene riscontrata una struttura dichiarativa delle regole. Raramente è presente una struttura imperativa.

La prospettiva usata è la seconda persona singolare (fatto che evidenzia la capacità di astrarre dalla specifica esperienza di gioco provata), il tempo è il presente indicativo.

Keyword informatiche sono presenti. Da notare in particolare *when* usato come condizionale e *then* come costrutto di sequenza.

Gli insiemi sono stati usati nella quasi totalità dei casi al posto dei cicli.

In caso di condizionali complessi, il più delle volte veniva specificata solo una delle condizioni (lasciando l'altra implicita), oppure usando un insieme di regole mutuamente esclusive per descriverli. Poche volte veniva utilizzato il pattern "`se A allora fai qualcosa a meno che B`".

Relazioni tra errori e analisi descrittiva In particolare sono state trovate correlazioni statisticamente rilevanti tra correttezza delle risposte da un lato e frequenza delle keyword, uso di insiemi o cicli dall'altro. Si riscontra inoltre un'elevata correlazione tra correttezza delle risposte e uso di condizionali espressi in forma di regole mutuamente esclusive.

Questo fa emergere l'osservazione che *l'aver davvero compreso la computazione sta alla base dell'utilizzo dei termini o delle strutture che esprimono tali concetti*. Inoltre, le soluzioni più complete erano anche le più corrette, e dunque gli studenti che scrivevano di più non stavano solo includendo dettagli, ma le parti *essenziali* per esprimere al meglio tali regole.

Analisi degli errori La prima osservazione interessante è che, nonostante l'assenza dei problemi legati alla sintassi, solo un quinto delle risposte sia completa, corretta e non ambigua.

Seconda osservazione è che gli errori di "omissione" (parti di regole necessarie non esplicitate) sono circa tre quarti del totale, rispetto a solo un quarto lasciato agli errori "commessi" (parti di regole presenti ma sbagliate).

Le regole sono quindi accurate ma non complete. Questo ridimensiona anche il timore dovuto al fatto che il linguaggio naturale potesse generare regole ambigue.

2.3.2 Commonsense Computing

Una serie di studi [128, 38, 90, 127, 100, 138], raggruppati sotto il titolo emblematico di "Commonsense Computing" e guidati dallo spirito costruttivista di Ben-Ari [25], sono stati svolti nell'ultimo decennio per investigare quali concetti legati alla programmazione, e più in generale all'informatica, siano posseduti dagli studenti *prima* di ricevere un'istruzione formale in questo senso.

Gli studi riguardano studenti che stavano per iniziare il loro primo anno di Università, alcuni iscritti a corsi di Informatica e alcuni ad altri corsi (es. Economia). I risultati sono analizzati in modo rigoroso e con metodi statistici consolidati nella ricerca psicologica. Alcuni risultati e commenti su di essi sono elencati e discussi di seguito.

Posti di fronte a un **problema di ordinamento di cifre** [128], da risolvere algoritmicamente in linguaggio naturale:

- più della metà degli studenti di Informatica e circa un quarto degli studenti di Economia *sono riusciti a fornire una soluzione corretta* e funzionante per una generica lista di numeri, esibendo dunque buone capacità di problem solving;
- la maggioranza degli studenti *ha trattato i numeri non come tipo di dato primitivo ma scomponendoli in cifre e trattando le cifre come caratteri*, mostrando quindi di avere un modello a basso livello del computer e diverso da quello tradizionalmente introdotto;
- gli studenti hanno usato l'iterazione nelle loro risposte, nessuno però ha usato la parola *while*, mentre tipicamente hanno espresso i concetti come *repeat...until*, mostrando come sia più intuitiva la tipologia di cicli con test posticipato;

- sorprendentemente, gli studenti a cui è stato chiesto di ripetere l'esercizio dopo alcune settimane di corso introduttivo alla programmazione, hanno *peggiorato* le loro prestazioni.

Mettendo altri studenti di fronte ad un **problema di ordinamento di date** (nel formato americano mese/giorno/anno) [38]:

- la correttezza delle risposte è addirittura aumentata rispetto al problema dei numeri;
- il focus in questo caso è stato sullo "spezzare" la data nelle sue tre parti costituenti, e ordinarli per anno, mese e giorno; sorprendentemente tali elementi sono stati trattati questa volta dalla quasi totalità come numeri;
- l'operazione di sorting all'interno dei gruppi *anni*, *mesi*, *giorni* è stata vista da molti come primitiva (solo un quarto degli studenti si è occupato per esempio di "trovare l'anno più piccolo", gli altri hanno semplicemente assunto che il computer sapesse ordinare gli anni)
- le risposte contenenti condizionali e iterazione non erano correlate alla correttezza delle stesse, spesso tali strutture erano invece lasciate implicite.

Ad alcuni studenti è stato chiesto di **risolvere un problema di concorrenza** [90]: modellare un sistema di vendita telefonica di biglietti per un concerto (assegnazione dei posti migliori, marcatura dei posti come non disponibili, pagamento) con più venditori contemporanei. Si è osservato che:

- la quasi totalità degli studenti *ha identificato il problema principale*, cioè la possibilità che venissero venduti gli stessi posti a più di una persona;
- circa il 70% ha fornito una soluzione "ragionevole" al problema principale;

- il 55% delle soluzioni ragionevoli ha fatto *uso di una entità centralizzata* a cui si demandava l'assegnazione dei posti, il 45% ha dato invece in qualche modo *responsabilità ai singoli venditori* di prendere decisioni in merito;

- le soluzioni errate spesso "spostano il problema" dall'acquisto alla prenotazione, o si pensa di poter risolvere il problema avendo un sistema "abbastanza veloce";

- gli studenti non si sono curati del livello di granularità delle operazioni, in generale; i pochi che l'hanno fatto, l'hanno usata troppo grossa (es. leggere e scrivere il database come operazione atomica).

Quattro problemi [127], che descrivevano **quattro situazioni di vita reale** (una stanza di una casa in affitto con la luce non funzionante, il gioco del "telefono senza fili", trovare un negozio di una catena in un paese straniero, un problema a scelta affrontato nella propria vita) che potessero essere **analoghe a una situazione in cui un programmatore si trovi a fare debug**, sono stati posti agli studenti. Si è osservato che:

- in generale, le soluzioni sono molto più "ripara e testa" piuttosto che "testa per acquisire informazioni";

- un comportamento comune è "partire da una diagnosi", che sia o meno esplicitata;

- viene fornito un processo strutturato per risolvere il problema, indicando che è stato generato e seguito un piano;

- frequentemente viene indicata come strategia il "riprovare"; questo ha un senso perché la vita reale è aleatoria, mentre difficilmente ricompilare un programma può risolvere un bug;

- il semplice "testing" è mediamente frequente, indicando che non viene considerato come obbligatorio nella soluzione di un problema;

- "provare più di una soluzione", "avere un'opzione di fallback" e "aggirare il problema" sono stati comportamenti indicati molto raramente;
- "l'uso dettagliato della conoscenza di dominio" o "l'uso di strategie per ottenere informazioni dettagliate del sistema" non sono presenti;
- "disfare ciò che si è fatto" non è intuitivo: pochissimi studenti riconoscevano necessario annullare un tentativo di riparazione fallita per riportare il sistema allo stato precedente.

Ad un gruppo di studenti è stato mostrato **un problema di efficienza di un algoritmo** [100], chiedendo loro di scegliere tra due soluzioni proposte quella che fosse migliore *nel caso pessimo*. Ad alcuni di loro è stato chiesto di calcolare esplicitamente i risultati degli algoritmi su alcuni valori concreti (ma non risolutivi per la scelta tra i due algoritmi). Si è osservato che:

- molti studenti non hanno usato la metrica del *caso pessimo*, sebbene esplicitamente richiesta, preferendo una serie di altre metriche, non sempre corrette (es. "caso medio", "migliore nella maggior parte dei casi", "migliore in ogni caso", "più corretto", "migliore nella vita reale"); si ipotizza che gli studenti avessero idee diverse o confuse sul concetto;
- la maggior parte degli studenti *ha individuato l'algoritmo migliore*, sebbene non sempre si è riuscito a stabilire a quale ragionamento abbia fatto seguito tale scelta;
- gli studenti a cui è stato chiesto di fare ***tracing*** dell'algoritmo hanno avuto risultati significativamente migliori, anche nei casi in cui il tracing era sbagliato;
- gli studenti che, nonostante non fosse richiesto, *hanno fatto uso di esempi* nelle loro risposte hanno ottenuto risultati migliori;
- gli studenti *hanno avuto difficoltà ad astrarre da alcuni dettagli*, non fondamentali, presenti nel problema.

Uno studio [138] ha chiesto di **interpretare la correttezza di varie ricette di cucina, sottoposte a due vincoli** sugli ingredienti che contenessero l'operatore logico NAND (*non entrambi*) e l'operatore logico IFF (*se e solo se*), espresse in linguaggio naturale. Si è osservato che:

- gli studenti non erano fuorviati da "distrattori", ovvero ingredienti aggiunti alle ricette che non comparivano nelle regole;
- gli studenti hanno nella stragrande maggioranza *applicato correttamente il vincolo che conteneva il NAND* (in contrasto con i risultati di uno studio precedente);
- *solo il 20% degli studenti ha correttamente applicato la regola IFF*, la grande maggioranza degli altri l'ha interpretata come IF, in misura molto minore come OR o NOR;
- riscrivendo la frase originale "use nutmeg **if and only if** you use cinnamon" ("*usa la noce moscata **se e solo se** usi la cannella*") in "use **either both** cinnamon **and** nutmeg **or neither** cinnamon **nor** nutmeg" ("*usa **sia** la noce moscata **sia** la cannella **oppure né** la noce moscata **né** la cannella*") si è ottenuto un risultato enormemente migliore (80%) nella comprensione della regola;
- ripetendo il test per il NOR durante l'esame finale, gli studenti hanno fatto *peggio*.

Capitolo 3

Spianare la strada al Pensiero Computazionale

To understand a program you must become
both the machine and the program.
– Alan J. Perlis (1982)

3.1 Informatica, Pensiero Computazionale e Programmazione

Così come l'informatica è molto di più della programmazione, a maggior ragione il pensiero computazionale non può essere identificato con essa.

Indubbiamente però il pensiero computazionale ha una forte componente linguistica e di programmazione: al cuore di tale pensiero c'è il pensiero algoritmico, e il suo obiettivo principale è quello di descrivere procedimenti effettivi per la risoluzione dei problemi. Inoltre, almeno nel panorama del prossimo futuro, tale metodologia sarà usata *di fatto* soprattutto per risolvere

problemi tramite i calcolatori, sempre più pervasivamente presenti attorno a noi.

La definizione di pensiero computazionale, inoltre, origina proprio dal riconoscimento delle skill acquisite dagli informatici. 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.

Sposare alla cieca questo semplice ragionamento vorrebbe dire però far finta di non vedere i problemi che questa strada genererebbe (oltre a terminare qui il capitolo). Non è pensabile prendere semplicemente un corso introduttivo di programmazione, per esempio quello di un primo anno di università, e mapparlo sui gradi più bassi di istruzione, per alcune ragioni:

- l'età conta - la capacità di astrazione, per fare un esempio, completa il suo sviluppo solo all'inizio dell'adolescenza;
- l'interesse conta - l'istruzione universitaria è molto tecnica e specialistica, non si può pensare che un bambino o un ragazzo trovi stimoli in tale tipo di attività;
- i corsi, così come sono presentati oggi, faticano a insegnare la programmazione, come dimostrano le difficoltà elencate in precedenza;
- la programmazione non è tutto: si può programmare senza curarsi della complessità degli algoritmi, della loro generalità, della riusabilità del codice, della leggibilità, eccetera;
- lo scopo non è quello di creare dei professionisti ma quello di diffondere il più possibile questo modo di pensare, di modo che sia applicato nella vita di tutti i giorni e in tutte le altre discipline e attività professionali;
- per rimarcare la sua centralità come abilità centrale del XXI secolo, il pensiero computazionale deve avere uno spazio dedicato: non vogliamo

che sia (solo) un side-effect dell'apprendimento della programmazione; quest'ultima, semmai, va vista come uno strumento.

La letteratura sul pensiero computazionale è comunque unanime nel riconoscere la necessità di un formalismo per descrivere, in modo effettivo, processi computazionali. Quando però ci si trova a dover descrivere tale formalismo, si finisce inevitabilmente per pensare ad un linguaggio di programmazione.

Sebbene si riconosca che rimanere troppo legati all'insegnamento dei linguaggi di programmazione sia una visione non troppo lungimirante, si ritiene altresì che cercare di migliorarne l'insegnamento sulla base degli studi a disposizione sia comunque una prima risposta alla crescente necessità di tali abilità per la propria vita personale e professionale.

Si riconosce comunque che i suggerimenti proposti di seguito vadano inseriti in una ricerca più ampia, con prospettive temporali più lunghe, su quali siano le caratteristiche *intrinseche* del pensiero computazionale indipendenti dallo strumento "linguaggio di programmazione", quali siano le età e le metodologie giuste per introdurlo.

Per fare ciò è necessaria una grossa collaborazione con la ricerca in campo psicologico e pedagogico. Questo richiede sforzi notevoli da parte della comunità scientifica e non è, ovviamente, stato attuabile con i tempi e i mezzi ristretti a disposizione di una tesi di laurea.

Proponiamo quindi due strade, da seguire in parallelo. La prima sta portando e porterà a risultati concreti nel breve periodo, la seconda, invece, potrebbe essere più dura da percorrere all'inizio ma probabilmente poterà a risultati più generali e duraturi.

3.2 Programmazione a portata di mano

Seguire la prima strada significa rispondere all'appello di Mark Guzdial [66], 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.

Come possiamo facilitare il compito di chi sta apprendendo? Le possibilità sono molte. Quelle individuate, dalla più conservativa alla più radicale sono:

- porre maggior attenzione agli aspetti più problematici quando si insegna la programmazione (es. spiegare esplicitamente i costrutti e gli aspetti fonti di misconcezioni, fornire esempi totalmente o parzialmente risolti, far fare tracing e debug, spiegare esplicitamente una macchina concettuale);
- usare strumenti ausiliari, quali linguaggi visuali, strumenti di visualizzazione (e manipolazione) dello stato durante l'esecuzione del programma;
- progettare i linguaggi di programmazione in modo che i costrutti risultino più "naturali";
- programmare in linguaggio "naturalistico" (vicino al linguaggio naturale ma senza ambiguità, con limitazioni).

Analizziamole nel dettaglio.

3.2.1 Porre attenzione agli aspetti problematici

Quando esprimiamo dei concetti, usiamo la categoria che ci sembra "più naturale": quella che Lakoff [88] chiama "categoria di livello base". Non diremmo mai "c'è un mammifero sul tavolo" o "c'è un soriano sul tavolo" quando il nostro gatto non ci lascia pranzare.

Allo stesso modo, gli studenti nello studio sull'ordinamento di date [38] non si sono soffermati sui tipi di dato o su operazioni di controllo del flusso di esecuzione o su funzioni come l'ordinamento, in quanto le trovavano ovvie o primitive. Così non è, e dunque è necessario, quando si incontrano questi aspetti, porre particolare attenzione durante la loro introduzione, per favorire le migliori concezioni o il superamento di quelle errate.

Alcune fonti di misconcezioni sono intrinseche nel linguaggio, e l'unica cosa che si può fare senza cambiarlo è evidenziare esplicitamente le differenze tra linguaggio naturale e linguaggio di programmazione. Altre misconcezioni, invece, sono dovute alla mancata comprensione della macchina concettuale o a schemi e modelli mentali errati o assenti: vediamo alcune proposte per affrontarle.

Facilitare la creazione di schemi

Gli studi citati nel Capitolo 2 mettono in evidenza la necessità degli studenti di formarsi schemi, partendo da quelli più semplici e, gradualmente, passare a schemi più complessi e ad un livello più alto di astrazione. Riteniamo quindi importante 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. Uno studio [74] ha testato un linguaggio di programmazione visuale a cui erano state aggiunte delle "meta-istruzioni" per gestire esplicitamente schemi e piani, con la possibilità di combinarli, testarli separatamente, raffinarli, ottenendo buoni risultati in un corso introduttivo.

Va di contro notato che questo tipo di approccio favorisce uno stile estremamente bottom-up, che, secondo gli studi, va in un secondo momento scoraggiato per favorire un'analisi più astratta e top-down. È dunque importante non indugiare troppo su questo approccio.

Insegnare a leggere prima di insegnare a scrivere

Così come ci viene insegnato a leggere nella nostra lingua prima che ci venga insegnato a scrivere, così dovrebbe essere nella programmazione. A titolo aneddotico: ho visto vari programmatori alle prime armi (sia bambini che giovani universitari), bloccati su un bug a loro dire "introvabile", illuminarsi alla mia richiesta di "spiegarmi cosa fa il programma, riga per riga".

Nel Capitolo 2 è stata sottolineata l'importanza di far fare tracing agli studenti da un lato, e la loro scarsa capacità e attenzione nei confronti di tale tecnica dall'altro. È importante prevedere:

- studio di esercizi risolti e commentati (così da ridurre il carico cognitivo e lasciare spazio all'apprendimento);
- esercizi di completamento, meno noiosi dei precedenti;
- esercizi di debug, come "caccia all'errore", che necessitino di fare tracing per essere risolti;
- utilizzo dei debugger spesso presenti negli ambienti di sviluppo software, e poco o per nulla sfruttati dagli studenti.

Chiedere esplicitamente di eseguire mentalmente o di spiegare il funzionamento del codice è anche un buon modo di insegnare agli studenti a ragionare *dal punto di vista del programma*, cosa che, come detto, non fanno intuitivamente. Risultati concreti si sono visti negli esperimenti sull'efficienza degli algoritmi [100], dove un tracing (persino se errato) portava a notevoli miglioramenti nella comprensione degli stessi.

Fare tracing può aiutare a individuare un errore in una porzione di codice limitata, ma non è sufficiente come unica attività di debug. Gli studi [127] infatti mostrano come le tecniche per trovare gli errori non siano tra le più intuitive e vadano pertanto insegnate esplicitamente. In particolare:

- "prova a ricompilare" non è un buon consiglio: il mondo reale in cui viviamo è aleatorio e *ritentare* sembra una buona strada, mentre il mondo del computer è deterministico e dunque questo raramente poterà a miglioramenti;
- bisogna spiegare esplicitamente che è necessario disfare ciò che si è fatto se il tentativo è fallito, prima di provare un'altra soluzione, di modo da riportare il sistema a uno stato noto;

- va suggerito esplicitamente di trovare l'errore, prima di correggerlo, insegnando tecniche quali stampe di controllo; anche la comprensione dell'errore è importante per una corretta correzione (per esempio suggerendo un'analogia col medico che fa esami diagnostici prima di proporre una cura);
- porre l'attenzione sul fatto che le cause di un errore possono essere molte, per esempio sottoponendo gli studenti a esempi ed esercizi con bug causati da fattori diversi.

Esplicitare una macchina concettuale

Come già visto, 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 [25], 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 e in particolare si consiglia di:

- presentare esplicitamente un modello concettuale del computer (es. Macchina di Von Neumann) e di macchina concettuale del linguaggio;
- ritardare gli esercizi di programmazione fino a quando gli studenti non abbiano costruito un modello mentale adatto agli scopi degli esercizi stessi;
- guidare lo studente nel cambiamento dei modelli "sbagliati".

L'importanza della macchina concettuale è riconosciuta in letteratura, ma non messa in atto in pratica: per esempio, nei testi introduttivi si parla spesso di Macchina di Von Neumann, ma pochissimi corsi insegnano a comprendere il mondo a runtime di un linguaggio.

Il risultato inatteso dello studio sugli ordinamenti di naturali [128], in cui gli studenti hanno trattato i numeri non come tipo di dato primitivo

ma come sequenze di caratteri, mostra come il loro modello di macchina (già presente, nonostante fossero al primo giorno di corso) sia molto diverso da quello normalmente usato dai linguaggi insegnati. È perciò importante guidarli verso un modello in cui i numeri siano oggetti primitivi.

I costruttivisti più estremi, quali Greening, osteggiano l'idea di insegnare esplicitamente la macchina, puntando a un approccio più pratico e immersivo in contesti realistici. Nonostante questo, in letteratura non è stato trovato nessun supporto a favore di questa ipotesi [134].

3.2.2 Rendere più naturali i costrutti

Alcuni [103] 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. Purtroppo i linguaggi tradizionali richiedono al programmare di compiere radicali trasformazioni tra la descrizione astratta del proprio task e l'implementazione. Pensiamo per esempio al programma per sommare gli elementi di una collezione in C, confrontato con la funzione `SOMMA()` di un foglio di calcolo.

Gli studi mostrati nel Capitolo 2 [128, 138] evidenziano spesso che gli stessi studenti, dopo un periodo di istruzione formale di codice, abbiano ottenuto risultati *peggiori* nella risoluzione di problemi, rispetto a quando erano totalmente nuovi alla programmazione. Gli autori hanno individuato il problema nel tentativo (spesso fallimentare) di esprimersi in modo più formale o nel linguaggio insegnato. Emblematico il caso [138] di uno studente che ha chiesto al docente: «Come devo rispondere a queste domande? Secondo ciò che lei ci ha insegnato o nel modo con cui penso solitamente a queste cose?».

La letteratura, fin dagli anni Ottanta, ha evidenziato i punti critici nel design dei linguaggi di programmazione. Purtroppo durante il design di molti linguaggi non si è tenuto conto di queste scoperte [108].

Proporre di cambiare i linguaggi di programmazione è sempre azzardato, vista la radicazione dei linguaggi e quindi le forti resistenze all'adozione. In questo contesto però ci stiamo occupando di linguaggi che avranno scopo didattico e, ancora più specificamente, saranno progettati per insegnare al meglio il pensiero computazionale. Poiché ancora non ne esistono con questo scopo specifico, sarà bene tenere in considerazione il modo con cui i bambini o i neofiti della programmazione esprimono le istruzioni nella progettazione degli stessi. Elenchiamo alcuni consigli basati su quanto emerso dagli studi.

- L'iterazione su collezioni (array, insiemi, liste) dovrebbe essere più astratta: costrutti come il "foreach" (`for` su oggetti `Iterable` in Java) devono costituire la norma. Operazioni a più alto livello sulle collezioni, quali mapping e filtri sono inoltre preferibili, in quanto le più utilizzate ("incrementa tutti gli elementi dell'insieme"). Questo eliminerebbe la maggior parte delle situazioni in cui si usano i cicli tradizionali.

- Bisogna favorire i cicli con test posticipato della condizione e con le parole chiave `repeat..until`, per lo meno all'inizio. In un secondo momento è utile avere il costrutto `while` (parola chiave però non intuitiva, da riconsiderare quindi), per mostrare come esso sia più generale (può sostituire il `for` e dà la possibilità di scrivere un ciclo eseguito potenzialmente 0 volte).

- 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" - quindi un'istruzione bloccante - o "non appena la condizione è falsa smetti di fare qualcosa" - quindi con controllo continuo. I nuovi linguaggi dovrebbero prendere in considerazione questo tipo di costrutti e stabilirne la forma migliore.

- La parola chiave `then` viene sempre intesa come costrutto di sequenza ("dopo"), è quindi assolutamente da evitare nel costrutto `if`.

- Il ramo `else` è scarsissimamente utilizzato perché ciò che accade quando la condizione termina si considera implicito. Il programma però non può dedurre dal contesto cosa fare se la condizione è falsa. Studi [102] mostrano che una struttura alternativa quale

```
if <condizione>: istruzioni;
not <condizione>: istruzioni;
end <condizione>
```

  migliora di molto la capacità di comprensione del codice e l'uso corretto del costrutto.

- Le variabili sono fonti di numerose misconcezioni. In particolare sarebbe da evitare il simbolo `=` per esprimere l'assegnamento. Più in generale, è bene evitare simbologie che possono essere confuse con quelle altri ambiti (es. funzioni matematiche contro funzioni e procedure).

- Molti errori e omissioni sono dovuti alla difficoltà di esprimere condizionali complessi, in particolare se contengono negazioni.

  - Si è osservata la tendenza a mettere, nelle condizioni, il caso generale all'inizio e poi elencare le eccezioni alla fine, in positivo, con la parole chiave `unless`. Il supporto di questo costrutto, in stile `try... catch` può essere utile.
  - Alcuni errori sono generati dalle parole chiave usate per gli operatori logici, in quanto sovraccaricate o ambigue nel linguaggio naturale (es. `and`). Bisogna pertanto prevedere sintassi alternative per tali operatori.
  - Alcuni suggeriscono 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).

- Gli studi mostrano come gli errori siano spesso di omissione: non vengono esplicitati alcuni elementi necessari per le istruzioni. Si propone

quindi di fornire template che richiedono il completamento per guidare il programmatore.

- I due punti precedenti suggeriscono un cambio radicale dei linguaggi, che andrebbero sempre di più verso lo stile visuale, con blocchi o template preconfezionati da usare tramite interfaccia grafica. Gli studi mostrano come l'uso di disegni per esprimere i concetti sia larghissimo nei bambini e nei novizi, e dunque si dovrà tenere conto di questo nella progettazione dei nuovi linguaggi.

Questi consigli riguardano soprattutto la sintassi. C'è però una domanda ancora più fondamentale per il design di un nuovo linguaggio.

Che paradigma adottare?

La scelta del paradigma da adottare per l'insegnamento dei linguaggi di programmazione ha generato nel corso della storia recente innumerevoli dibattiti. Senza entrare nel dettaglio, cerchiamo di vedere quali suggerimenti vengono dal comportamento dei neofiti della programmazione.

I linguaggi funzionali e ancor più quelli dichiarativi, di certo, sono più astratti e dunque più vicini alla rappresentazione mentale del programmatore. D'altro canto però esprimono più il "cosa fare" piuttosto che il "come farlo" e, come argomenteremo meglio in seguito, potrebbero in un certo senso indebolire l'apprendimento del pensiero computazionale, che si basa proprio sulla necessità di esprimere procedimenti algoritmici.

Negli ultimi anni la letteratura di Didattica dell'Informatica si è concentrata in particolare su tutti gli aspetti legati all'Object Oriented, tanto da generare materiale sufficiente per una tesi ad hoc.

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, usato nell'implementazione di classi e metodi.

Gli studi mostrano come il paradigma sia intuitivo solo in parte: se è vero che sono stati riscontrati, nelle descrizioni in linguaggio naturale, alcuni aspetti del paradigma ad oggetti, ovvero *entità trattate come aventi uno stato e rispondenti a richieste di azioni*, 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. Uno studio [45] mostra come le gerarchie di ereditarietà causino difficoltà nei bambini.

L'uso del paradigma orientato agli oggetti è stato suggerito come uno dei modi con cui si può "passare ad una macchina concettuale" più astratta: insegnare la programmazione in termini di oggetti che si scambiano messaggi tra loro. Attualmente però questo vorrebbe dire imparare *due macchine*: una ad alto livello e una che invece è sostanzialmente una macchina procedurale estesa.

I bambini, specialmente, usano regole di produzione o programmazione basata su eventi e in seconda battuta regole dichiarative. I ragazzi sembrano più inclini alla programmazione imperativa (sempre come seconda scelta, però).

Il mix di stili utilizzati in diverse situazioni (es. imperativo solo localmente, basato su regole o eventi per specificare il comportamento degli elementi, dichiarativo per fare setup iniziale) suggerisce di non limitare il linguaggio ad un singolo paradigma, ma lasciare la libertà di usarne di più. Un buon esempio potrebbe essere The Wolfram Language [13], implementato nel software commerciale Mathematica e che permette di adottare vari stili di programmazione: basato su regole, funzionale, imperativo.

3.2.3 Usare strumenti di visualizzazione

La cognizione umana ha una forte componente visiva. La vista è di gran lunga il nostro senso dominante e, in generale, apprendiamo meglio tramite immagini che non tramite parole scritte o pronunciate [101].

Usare supporti grafici per facilitare la progettazione, realizzazione e comprensione dei programmi non è un'idea nuova: vale la pena di citare il lavoro seminale di Von Neumann che ha introdotto i diagrammi di flusso per la programmazione [60]. Negli anni, con lo sviluppo dei linguaggi ad alto livello, gli informatici si sono affrancati da questo formalismo. L'utilizzo di supporti grafici però non si è estinto, basti pensare alla diffusione di UML.

Nell'ambito della Didattica dell'Informatica, si è fatto largo uso di strumenti visivi negli ambienti di programmazione per novizi. Per una panoramica sugli ambienti e i linguaggi per l'educazione, si veda il lavoro di Kelleher e Pausch[1] [81]. Alcuni aspetti in particolare stanno avendo risalto nell'ambito del pensiero computazionale.

- **Linguaggi di programmazione visuale**. Si tratta di linguaggi di programmazione che permettono agli utenti di creare i programmi manipolando elementi grafici piuttosto che specificare le istruzioni testualmente. La notazione può essere costituita, ad esempio, da diagrammi o da "blocchi" da unire tra loro per costruire il programma. Sebbene le difficoltà legate alla sintassi non siano ritenute cruciali, insegnare la programmazione al più ampio pubblico possibile, in particolare ai bambini, rende necessario e utile semplificare il metodo con cui le istruzioni vengono specificate e limitare gli errori di sintassi.

- **Animazione dei programmi**. Si tratta di una forma di visualizzazione in cui l'ambiente mostra esplicitamente alcuni aspetti dell'esecuzione di un programma (es. stato delle variabili, stack delle chiamate, oggetti...). Sono simili a un debugger, ma non indirizzati a esperti, bensì progettati con scopi didattici, dunque selezionando quali dettagli mostrare. Tramite questo tipo di strumenti si può rendere visibile la macchina concettuale[2], riducendo così il carico cognitivo legato alla

[1] Randy Pausch (1960 - 2008), di cui non posso non consigliare "The Last Lecture".

[2] Non necessariamente mostrando fedelmente la macchina astratta del linguaggio, ma evidenziandone alcuni aspetti o astraendoli, mostrando cioè una macchina concettuale a più alto livello.

sua intangibilità e aiutando gli studenti nella formazione di un modello concettuale. Sebbene questo tipo di strumenti siano abbastanza diffusi (per una review si veda [133]), ci si sta sempre più rendendo conto - in pieno indirizzo costruttivista e costruzionista - che *mostrare* qualcosa non è sufficiente.

- **Simulazione visuale di programmi**. Si tratta della naturale evoluzione dell'animazione dei programmi, in cui però lo studente riprende un ruolo attivo: 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 stimola il conflitto cognitivo, che aiuta a superare le misconcezioni eventualmente presenti nello studente [130]. Inoltre "forzare" il tracing come compito può aiutare gli studenti a comprenderne l'importanza.

3.2.4 Programmare in linguaggio naturale

L'idea di avvicinare il piano mentale (l'algoritmo) al programma, portata all'estremo, punta inevitabilmente nella direzione della *programmazione in linguaggio naturale*.

Secondo [83] infatti le difficoltà sono dovute al fatto che, per programmare, compiamo un processo in tre fasi: per prima cosa emerge nella nostra mente l'idea dell'algoritmo, cioè quello che vogliamo fare; a questo punto trasformiamo questa idea in una forma che supponiamo possa andare bene al compilatore o interprete scelto; infine, scriviamo effettivamente il programma nel linguaggio scelto. Questo processo ci costringe di fatto a modificare, almeno in parte, la nostra idea, per adattarla al linguaggio. Il formalismo adottato, inoltre, influenza anche il nostro modo stesso di pensare: se so che dovrò usare Java mi verrà naturale ideare il programma in termini di classi, metodi e attributi.

Programmare in linguaggio naturale risolverebbe tutti questi problemi, annullando il gap tra idea e realizzazione, e avrebbe alcuni importanti vantaggi:

- il codice prodotto sarebbe estremamente generale, portabile, e duraturo nel tempo, perché esprimerebbe concetti persistenti e validi in moltissimi campi di applicazione;

- gli sviluppatori potrebbero concentrarsi su ciò che è veramente importante nella scrittura di software: descrivere e realizzare l'idea; al contrario oggi si passa molto tempo a confrontarsi con problemi minori (es. la codifica dei caratteri), a scapito delle attività più creative;

- tutti coloro che non si occupano di informatica ma hanno bisogno, nella vita quotidiana o nelle discipline, di programmare potrebbero farlo *senza imparare un linguaggio di programmazione*.

Sebbene la programmazione in linguaggio naturale sia utopica, in quanto le lingue umane hanno un grado elevato e intrinseco di ambiguità, si può pensare di arrivare a linguaggi di programmazione *naturalistici*: un linguaggio formale, con vincoli e aggiustamenti per permettere una sintassi e semantica non ambigua, ma che sfrutti tutti i vantaggi e le caratteristiche (riferimenti impliciti, località dei riferimenti, compressione, economicità, dipendenza dal contesto) che rendono naturale il *linguaggio naturale*, appunto. Alcuni studi stanno compiendo passi in questa direzione [82].

Programmare in linguaggio naturalistico è forse la destinazione dell'evoluzione dei linguaggi di programmazione. Essi, nei decenni, sono diventati sempre più astratti e più vicini all'utente. Dopotutto, l'interazione con i dispositivi, oggi ubiquamente presenti attorno a noi, avviene sempre più massicciamente tramite l'uso della voce e dei comandi espressi nella nostra lingua. Quando il sogno di programmare in linguaggio naturale o naturalistico *tutti* i computer che ci circondano sarà realtà, probabilmente si ridimensionerà di molto la necessità del pensiero computazionale.

Mantenendo però una prospettiva a breve termine, insegnare a programmare con linguaggi troppo simili a una lingua umana non sembra vantaggioso in termini di pensiero computazionale, per almeno due ragioni:

- il linguaggio naturale nasconderebbe troppi dettagli, rendendo ancora più oscuro ciò che succede "dietro le quinte" [72];

- una sintassi naturale sarebbe portatrice di numerose misconcezioni sul funzionamento e sull'intelligenza del computer (un computer non può fare inferenze dal contesto, fare domande per chiarire le ambiguità, ecc.);

Nel breve periodo, comunque, si possono sfruttare gli studi e i progressi in questo campo per favorire l'apprendimento, usando il linguaggio naturale per far emergere le misconcezioni e aiutare lo studente a comprendere il modello mentale. Una proposta [61] è quella di fornire una rappresentazione in linguaggio naturale (*read only*) accanto al codice, così da fornire allo studente la possibilità di confrontare la sua idea con le istruzioni che ha veramente dato al computer. Alcuni studi hanno provato questo metodo essere abbastanza efficace [34].

3.3 Leggere, Scrivere, Calcolare, Pensare computazionalmente

Seguire la seconda strada significa sposare l'idea di Jeannette Wing, che ha proposto di vedere il pensiero computazionale come abilità di base, insieme a leggere, scrivere e calcolare [145]. Così come la lingua scritta e parlata ci serve per comunicare, e la matematica di base per quantificare, così il pensiero computazionale ci permette di *elaborare* correttamente ed efficacemente *informazioni* e spiegare *come* eseguire un compito.

3.3.1 Pensiero Computazionale senza Programmazione

Come già evidenziato, l'impronta che la ricerca sta dando al pensiero computazionale è strettamente connessa alla programmazione. Alcuni però sono molto critici verso questa posizione. In particolare, Lu e Fletcher [94] evidenziano come sarebbe necessario porre le basi del pensiero computazionale molto tempo prima che gli studenti imparino il loro primo linguaggio di programmazione.

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. Scrivere descrizioni in un linguaggio formale sconosciuto può non essere facile se non si ha una solida conoscenza del processo che queste descrizioni catturano.

Non volendo insegnare la programmazione, l'enfasi, secondo Lu e Fletcher, 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 facilitare questo tipo di conoscenze e competenze si contraddistinguono per un fattore comune a nostro dire cruciale: in esse **gli studenti sono gli agenti computazionali.** Questo tipo di attività sembrano in linea con i risultati citati in questa tesi, per vari motivi:

- permettono allo studente di assumere il punto di vista del computer, cosa che di solito non tende a fare intuitivamente;
- impongono allo studente una forma di tracing in prima persona;
- dovendo seguire delle regole/istruzioni codificate, lo studente si rende conto dei limiti che questo comporta, evitando così misconcezioni legate a un computer "senziente";

- portano all'estremo il consiglio di posticipare gli esercizi di programmazione: i bambini passerebbero addirittura anni a comprendere il modello, prima di mettere le mani su un vero linguaggio; una volta arrivati al momento di impararlo, potrebbero concentrarsi sugli aspetti tecnici e implementativi, avendo una solida base teorica alle spalle;

- affrontano il problema del transfer (sollevato per esempio in [40]): coloro che ritengono che la conoscenza sia frammentata e molto legata al contesto ricordano che, 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 e facendo confronti tra contesti; in sintesi, se lo si insegna solo con la programmazione, sarà molto difficile che venga applicato in altri ambiti.

Un approccio di questo tipo, inoltre, aiuterebbe bambini e ragazzi a comprendere la differenza tra informatica e programmazione, favorendo magari la scelta di un eventuale corso di laurea per motivi intellettuali e non solo legati alle possibilità lavorative.

Gli autori hanno proposto, in questi ultimi anni, una grande quantità di attività. I riferimenti presentati nella rassegna iniziale (§ 1.8) permettono di accedervi. Elenchiamo, a titolo di esempio, alcune tipologie di attività, che sembrano essere adatte alla luce di quanto presentato in precedenza.

- La prima modalità, che permette un'applicazione vastissima (ma forse troppo frammentata), consiste nel mettere in evidenza gli aspetti del pensiero computazionale quando si utilizzano nelle altre discipline. Ad esempio, quando si introduce 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 (6×3) piuttosto che sei volte tre (3×6).

- 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***). Alcune raccolte di queste attività, in cui sono messi in evidenza gli aspetti del pensiero computazionale che possono insegnare, si trovano ad esempio in [63, 76].

- Si possono sfruttare i lavori di gruppo, che sono comuni - ad esempio - nelle lezioni di scienze naturali, per favorire una meta riflessione sul ***parallelismo***. Le interazioni per lo scambio di dati sono ottimi momenti per introdurre il concetto di ***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" [24]. In molte declinazioni diverse [24, 30, 76], una attività particolarmente suggerita consiste nel far eseguire istruzioni (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 (eventualmente) 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***).
  È facile riconoscere, in questo tipo di attività, i consigli del paragrafo precedente (punto di vista, debug, ecc.). Inoltre, se declinata dal punto di vista della programmazione, si tratta di un'alternativa cinestetica[3]

[3](Solo) una delle tipologie di intelligenza, secondo la teoria delle intelligenze multiple [99].

> alla visualizzazione della macchina astratta. Si pensi ad un'attività in cui gli studenti impersonano, come in un gioco di ruolo, gli oggetti (software) di un programma [18].

Escludere completamente la programmazione durante gli anni in cui si insegnano questi concetti ci sembra una scelta troppo radicale. Il problema del transfer infatti può essere 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.

È bene quindi prevedere la programmazione come *una* delle attività in cui si insegna il pensiero computazionale.

3.3.2 Un linguaggio serve

Se vogliamo che le persone acquisiscano la conoscenza per utilizzare concetti e strumenti computazionali, allora dovranno essere capaci di interagire con essi. Una qualche forma di rappresentazione delle strutture che vanno a manipolare è necessaria.

In pieno spirito costruzionista, si può osservare come scrivere codice permetta di convertire i concetti generali in una realizzazione concreta [32] (non necessariamente tangibile, ma *eseguibile*).

Ovviamente non si tratta di progettare *il* linguaggio: conoscerne più d'uno è un vantaggio per l'informatico, che può sfruttare i punti forti e deboli di ciascun linguaggio a seconda di ciò che vuole esprimere. L'importante è (lo ribadiamo ancora una volta) capire "cosa ci sta dietro", cioè conoscere (a fondo, se informatici di professione) i meccanismi e i principi che stanno alla base dell'ideazione, progettazione e implementazione del linguaggio stesso [97].

Caratteristiche

Tirando le somme di quanto visto finora, riassumiamo *alcune* caratteristiche che dovrebbe avere un buon linguaggio (o una famiglia di linguaggi)

adatto all'insegnamento del pensiero computazionale. Dovremmo parlare forse più correttamente di *ambiente di sviluppo e collaborazione*, in quanto, come vedremo, 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: supporta parti imperative, regole di produzione, agenti ed eventi;
- sia visuale o organizzato per template: la scrittura manuale di codice deve essere limitata;
- deve prevedere una forma grafica di rappresentazione del codice o almeno dei suggerimenti su cosa rappresentino *in quel momento* elementi quali variabili, parametri, funzioni;
- può prevedere una trasposizione read-only in linguaggio naturale;
- deve prevedere una forma di rappresentazione grafica dello stato della macchina concettuale (ad un *qualche* livello di astrazione) a compile time e a runtime;
- deve prevedere strumenti di tracing e debug completi e facili da utilizzare;
- deve essere il più possibile interattivo: i cambiamenti devono vedersi il prima possibile;
- può essere inserito in una piattaforma di collaborazione (in cui per esempio è possibile leggere il codice prodotto da altri, modificarlo in copia, o collaborare su uno stesso progetto);
- deve prevedere l'accesso selettivo a funzionalità o la loro complicazione a seconda dell'età o del livello dell'utente;
- può prevedere incentivi per coinvolgere in modo attivo chi apprende.

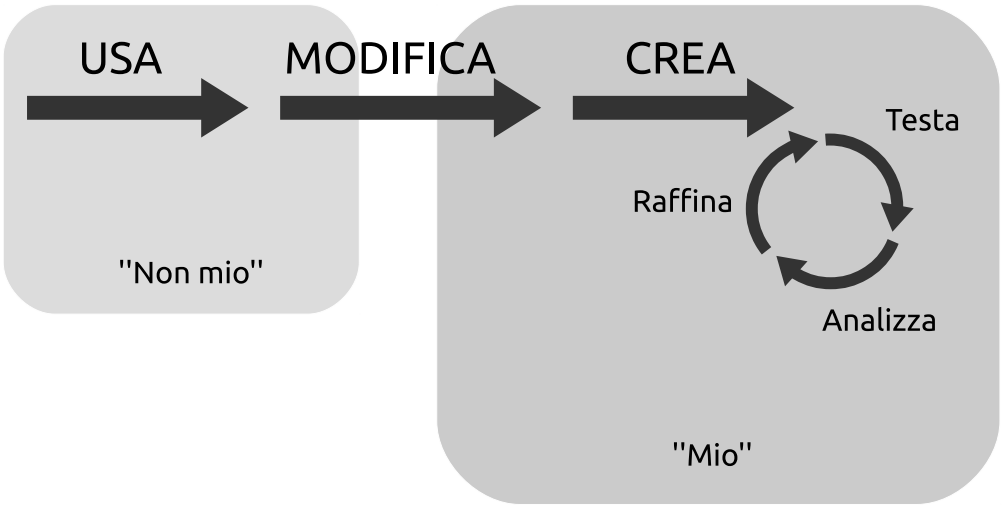

Figura 3.1: Progressione usa-modifica-crea, adattata da [89]

Usa, modifica, crea

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 [17, 89, 104, 40]), schematizzata in Figura 3.1.

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 qualcun altro (es. esegue simulazioni su modelli già esistenti, esegue un programma che controlla un robot o un videogame già creato); questa fase serve per imparare ad utilizzare gli strumenti, prendere confidenza con l'ambiente ed essere coinvolti;

- nella fase **"modifica"**, lo studente vorrà poi iniziare a modificare il programma, dapprima modificando qualche attributo (colore del personaggio, variabile *gravità* nella simulazione, ecc.) e poi scrivendo nuovi pezzi di codice, per i quali inizierà ad avere bisogno di comprendere i meccanismi che regolano il programma;

- nella fase **"crea"**, infine, lo studente verrà incoraggiato a realizzare un proprio progetto, applicando le conoscenze e le competenze che ha appreso nelle fasi precedenti, e che continua ad apprendere iterativamente migliorando i propri progetti.

Queste tre fasi non vanno ovviamente viste come compartimenti stagni, gli studenti potranno passare più volte da una fase all'altra, e utilizzandole tutte, ricorsivamente, nella realizzazione del proprio progetto.

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 [122].

Questo 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 [133] essere un buon esempio di "collaborazione in un contesto autentico" e di *situated learning* (§ 2.1.3), concetti molto cari ai sociocostruttivisti.

E adesso?

Attamen in mundo nostri temporis
rapidis mutationibus subiecto
et quaestionibus magni ponderis
[...] perturbato...
– Benedetto XVI (2013)
Rinuncia al papato

Come il lettore avrà notato, siamo solo all'inizio della seconda strada presentata nella sezione 3.3 (riconoscere il pensiero computazionale come quarta abilità di base). Per aiutare chi vi si avventurerà, ci permettiamo di indicare una possibile direzione, affidandoci alle parole di Jeannette Wing:

> Se il pensiero computazionale verrà usato in ogni campo, riguarderà tutti, direttamente o indirettamente. Questo solleva una sfida educativa. Se il pensiero computazionale verrà aggiunto al repertorio della abilità di pensiero, allora come e quando le persone dovrebbero imparare questo tipo di pensiero [...]?
>
> [...] la tendenza ad utilizzare il pensiero computazionale nella ricerca [...] è già in atto, [...] le università hanno già iniziato a incorporare il pensiero computazionale nei loro curriculum universitari, [...] cerchiamo di concentrare questa domanda sui

livelli di istruzione più bassi. Infatti, se vogliamo garantire una base comune e solida di comprensione e di applicazione del pensiero computazionale per tutti, allora sarebbe meglio che questo apprendimento avvenisse nei primi anni dell'infanzia. [...]

Quali sono i concetti elementari del pensiero computazionale? Gli educatori che si occupano di informatica hanno risposto e continuano a rispondere a questa domanda con la creazione di corsi, di solito per studenti universitari del primo anno, che si concentrano sui principi della computazione piuttosto che sull'abilità di programmazione dei computer. Poiché il campo dell'informatica continua a maturare, vale la pena rivisitare questa domanda, con un focus specifico sugli anni precedenti.

Inoltre, vale la pena di rivedere la domanda in collaborazione con studiosi di scienze dell'apprendimento e dell'educazione. Per esempio, quali sono, se ce ne sono, i concetti del pensiero computazionale innati nella cognizione umana come lo è[4] il concetto matematico di numero [naturale]? La visione umana è un'elaborazione parallela. Quali compiti svolgiamo o impariamo a svolgere in parallelo e quali in modo sequenziale? I bambini sperimentano le nozioni di infinito e di ricorsione attraverso la matematica e la lingua; nominare e insegnare presto questi concetti fondamentali in contesti formali di apprendimento potrebbe fornire una solida base per il pensiero computazionale.

Quale sarebbe un ordinamento efficace dei concetti da insegnare ai bambini man mano che la loro capacità di apprendimento progredisce nel corso degli anni? Per analogia, insegniamo numeri ai bambini nella scuola materna (quando hanno 5 anni), algebra alle medie (12 anni) e analisi alle superiori (18 anni). Ci possono essere molti modi per strutturare la progressione dei concetti del pensiero computazionale, qual è la più efficace per ogni tipo di

[4]In parte. Si veda ad esempio [148]

studente?

> Qual è il modo migliore per integrare lo strumento [il calcolatore elettronico] con l'insegnamento dei concetti? [...]. Il nostro campo [l'informatica] è in una situazione unica in quanto non solo ci sono concetti computazionali da insegnare, ma vi è anche uno strumento da insegnare. Questo strumento fornisce alcune sfide e alcune opportunità.
>
> Una sfida è data dal fatto che non vogliamo che lo strumento prenda il posto della comprensione dei concetti. Non vogliamo nemmeno che le persone siano in grado di utilizzare lo strumento, ma non abbiano imparato i concetti [...]. Peggio ancora, non vogliamo che la gente pensi di aver compreso i concetti perché è abile ad usare lo strumento. Una seconda sfida è la seguente: vogliamo sincronizzare l'apprendimento dello strumento con l'ordine di apprendimento dei concetti. In quale momento va introdotta ciascuna delle potenti funzionalità di un computer? In quale momento possiamo esporre i bambini alla complessità di funzionamento della macchina? [146]

Una volta individuati quali sono i concetti davvero intrinsechi al pensiero computazionale (le definizioni presentate nel Capitolo 1 sono solo un primo passo), bisognerà chiedersi quali siano i concetti "innati" o comunque "naturali" per l'uomo, e quali invece fatichiamo ad apprendere.

Il nostro cervello, per esempio, è bravo a eseguire molti task in parallelo. Un primo esperimento mostra come in effetti i ragazzi riescano a riconoscere e risolvere i problemi di concorrenza [90].

In questa tesi, inoltre, non si sono analizzate problematiche relative all'età. Dobbiamo però chiederci a quale età sia più facile e più giusto imparare ciascuno dei concetti fondamentali del pensiero computazionale. Ad esempio le teorie di Piaget[5] sullo sviluppo mostrano come solo allo stadio più alto del-

[5]Alcuni studi recentissimi [92, 136] cercano di parafrasare le fasi piagetiane con le fasi di sviluppo del programmatore, basando su di esse l'introduzione dei concetti.

lo sviluppo (quello "formale operazionale", raggiunto dai 12 anni in poi) gli studenti hanno l'abilità di pensare in modo astratto, sistematico e ipotetico. Studi mostrano come alcuni soggetti non raggiungano mai questo stadio o lo raggiungano molto tardi [85].

Si sottolinea quindi ancora la necessità di una stretta collaborazione tra informatici, psicologi, neuroscienziati e pedagogisti (anche con una evidenza aneddotica: le scarse conoscenze in questi ambiti da parte dell'autore di questa tesi sono state un ostacolo non indifferente).

Un altro aspetto di cui non ci siamo occupati, e che è stato trattato pochissimo, è quello - spinoso - della valutazione. Una volta che si sarà raggiunta una qualche forma di consenso sulla definizione, bisognerà prestare attenzione agli strumenti, alle modalità e alle metriche più utili per valutare l'apprendimento del pensiero computazionale, specialmente se, come probabilmente avverrà, costituirà una serie di abilità multidisciplinari e dunque distribuite in diversi contesti.

Una serie di studi [122, 123, 84, 23, 78], per ora unici nel loro genere, hanno infine cercato (con qualche successo, ma con necessità di ulteriori indagini) di valutare il transfer (specialmente rilevante in quanto vogliamo che il pensiero computazionale sia un'abilità trasversale), elaborando una tecnica di riconoscimento di "pattern di pensiero computazionale" per rispondere alla domanda: *ora che gli studenti sanno programmare Space Invaders, sanno programmare una simulazione scientifica?*

Appendice A

Menù di Misconcezioni

It is easier to write an incorrect program
than understand a correct one.
– Alan J. Perlis (1982)

Un esaustivo catalogo sulle misconcezioni (e più in generale: comprensioni parziali, incomprensioni, regole inventate dagli studenti, difficoltà, errori, bug) trovate in letteratura e riguardanti i corsi in cui veniva introdotta la programmazione è frutto di un ammirevole lavoro di pazienza in [133]. Si rimanda a tale lavoro per la bibliografia completa.

Lo riporto di seguito, tradotto e lievemente riorganizzato, per avere un quadro generale sulle difficoltà affrontate da chi si approccia per la prima volta alla programmazione.

A.1 Natura generale dei programmi

Il computer conosce le intenzioni del programma o di una parte di codice, agisce di conseguenza.

Il computer è capace di dedurre le intenzioni del programmatore.

I valori sono aggiornati automaticamente secondo un contesto logico.

Il sistema non permette operazioni irragionevoli.

Difficoltà a capire il tempo di vita dei valori.

Difficoltà a distinguere gli aspetti statici e dinamici dei programmi.

La macchina comprende l'Inglese.

Parallelismo magico: più righe di un programma (semplice, non concorrente) possono essere attive o conosciute.

A.2 Variabili, assegnamento, valutazione di espressioni

Una variabile può contenere più valori alla volta, oppure si "ricorda" i vecchi valori.

Una variabile riceve sempre un particolare valore di default al momento della creazione.

L'assegnamento funziona nella direzione opposta.

L'assegnamento funziona in entrambe le direzioni (swap).

Comprensione limitata delle espressioni dovuta alla mancanza del concetto di valutazione.

Una variabile è (solo) un accoppiamento di un nome a un valore modificabile (con un tipo). Non è memorizzata nel computer.

L'assegnamento memorizza equazioni o espressioni non valutate.

L'assegnamento sposta un valore da una variabile ad un'altra.

La semantica in linguaggio naturale dei nomi delle variabili influisce su quali valori vengono assegnati a quella variabile.

L'ordine di dichiarazione delle variabili influisce su quali valori vengono assegnati a quali variabili.

Non c'è limite nella dimensione e nella precisione dei valori che possiamo memorizzare in una variabile.

Incrementare un contatore è un'operazione indivisibile (il lato destro non richiede una valutazione separata).

I tipi primitivi (in Java) non hanno valori di default.

Variabili di tipo primitivo (in Java) non assegnate non hanno memoria allocata.

A.3 Flusso di controllo, selezione e iterazione

Difficoltà a comprendere la sequenzialità delle istruzioni.

Il codice dopo l'intera istruzione `if` non viene eseguito se viene eseguita la clausola `then`.

L'istruzione `if` viene eseguita non appena la sua condizione diventa vera.

Una condizione falsa termina il programma se non c'è un ramo `else`.

Sia il ramo `then` che il ramo `else` sono eseguiti.

Il ramo `then` viene eseguito sempre.

Usare `else` è opzionale (le istruzioni che seguono sono sempre, implicitamente, il ramo `else`).

Il codice adiacente (graficamente) viene eseguito all'interno di un ciclo.

Il controllo ritorna all'inizio se la condizione è falsa.

Difficoltà a comprendere il cambio automatico della variabile di controllo di un ciclo `for`.

Un ciclo `while` termina immediatamente quando la condizione diventa falsa.

La variabile di controllo di un ciclo `for` non ha valori all'interno del loop, o tali valori possono essere cambiati a piacere.

Le istruzioni di stampa sono sempre eseguite, indipendentemente dal flusso di controllo.

Tutte le istruzioni di un programma vengono eseguite almeno una volta.

A.4 Chiamata di sottoprogrammi, passaggio dei parametri

Il codice dei sottoprogrammi è eseguito sulla base dell'ordine in cui tali sottoprogrammi sono stati definiti.

Il valore di ritorno non ha bisogno di essere memorizzato (anche se è necessario in seguito).

Un metodo può essere invocato una volta sola.

Numeri e costanti numeriche sono gli unici parametri attuali corrispondenti a parametri formali di tipo `int`.

Difficoltà a distinguere tra parametri formali e parametri attuali. Confusione sulla provenienza dei valori dei parametri.

Difficoltà a comprendere l'invocazione di un metodo da parte di un altro metodo.

Confusione su dove vadano i valori di ritorno.

Il passaggio dei parametri forma un link basato sul nome tra variabili con lo stesso nome (nella chiamata e nella signature).

Il passaggio dei parametri forma un link procedura a procedura tra variabili con nome diverso (nella chiama e nella signature).

Il passaggio dei parametri richiede nomi diversi nella chiamata e nella signature.

I sottoprogrammi possono usare le variabili dei sottoprogrammi chiamanti.

Non si possono usare le variabili globali nei sottoprogrami quando non sono state passate come parametri.

Quando il valore di una variabile globale è modificato in una procedura, il nuovo valore non è disponibile al programma principale, a meno che non sia passato esplicitamente ad esso.

Una funzione cambia (sempre) le sue variabili di input per farle diventare di output.

Espressioni (non i loro valori) sono passati come parametri.

Espressioni negli argomenti modificano le variabili esistenti.

Al momento del ritorno, il valore di una variabile cambia per corrispondere a un parametro precedentemente dato.

A.5 Ricorsione

Modello nullo: la ricorsione è impossibile.

Modello attivo: compreso solo l'aspetto attivo della ricorsione, non i valori di ritorno.

Modello per passi: compresa la ricorsione in uno o due passi, ma non di più.

Modello del valore di ritorno: ogni istanziazione produce un valore, prima che la successiva sia iniziata, in seguito tutti i valori sono combinati per ottenere un risultato.

Modello passivo: la parte attiva non viene compresa, solo la combinazione dei valori di ritorno.

La ricorsione è solo un costrutto usato in alcuni tipi di programmi: il comportamento a runtime è *magia*.

La ricorsione è percepita come un problema di algebra.

Modello a loop (istanziazione singola): la ricorsione è solo un costrutto per produrre una ripetizione, non viene compreso l'autoriferimento.

A.6 Riferimenti, puntatori, assegnamento di oggetti

Anche i valori primitivi (in Java) sono gestiti tramite riferimenti.

Le variabili per memorizzare e assegnare valori primitivi sono fondamentalmente diverse dalle variabili usate per memorizzare gli oggetti (in Java).

Una variabile associata d un oggetto è solamente un nome che può essere usato per maneggiare un oggetto o qualcos'altro nel programma.

Una variabile (non primitiva, in Java) non contiene un riferimento ma un insieme di proprietà di un oggetto.

Assegnare un oggetto significa mandare un oggetto (o una sua copia) ad una variabile.

Assegnare ad un oggetto significa renderlo uguale all'oggetto assegnato.

Assegnare ad un oggetto significa che alcuni campi ottengono nuovi valori dall'oggetto assegnato.

Confusione tra un campo "nome" e una variabile che si riferisce all'oggetto.

Due oggetti con lo stesso valore per un campo "nome" sono lo stesso oggetto.

Il valore di un campo come "nome" modifica riferimento al valore di un attributo.

Un oggetto è rappresentato (nello stato del programma) dal solo valore di un particolare campo come "nome".

Una variabile che si riferisce ad un oggetto, si riferisce ad esso soltanto per tutto il tempo.

Una variabile che punta a un oggetto, punta sempre a quello.

Due variabili differenti devono per forza puntare a due oggetti diversi.

Due oggetti della stessa classe con lo stesso stato sono lo stesso oggetto.

Due oggetti possono avere lo stesso identificatore se c'è una qualche differenza nei valori dei loro attributi.

Gi oggetti sanno chi si riferisce a loro. I riferimenti vanno nella direzione opposta.

A.7 Rapporto tra classe, oggetto, istanziazione di una classe

Confusione tra una classe e una sua istanza.

Un oggetto è un sottoinsieme di una classe. / Una classe è una collezione di oggetti.

Un oggetto è un sottotipo di una classe.

Un insieme (es. una "squadra"), non può essere una classe.

I costruttori possono includere solo assegnamenti per inizializzare gli attributi.

L'istanziazione coinvolge solo l'esecuzione del corpo di un costruttore, non l'allocazione della memoria.

Difficoltà a capire il costruttore vuoto.

Inizializzare un attributo con una costante come parte della sua dichiarazione causa confusione nella distinzione tra una classe e un oggetto.

Inizializzare un attributo con una costante all'interno della dichiarazione del costruttore causa confusione nel distinguere tra una classe e un oggetto.

L'invocazione del costruttore può sostituire la sua definizione.

Difficoltà di comprensione quando oggetti di una classe semplice sono cerati prima della creazione dell'oggetto di una classe composta.

Difficoltà a capire gli oggetti se i loro attributi non sono esplicitamente inizializzati.

Gli oggetti "si creano da soli", senza bisogno di istruzioni esplicite per crearli.

Dichiarare una variabile crea anche un oggetto.

Puoi definire un metodo (non costruttore) per creare un nuovo oggetto.

La rappresentazione testuale di un oggetto è un riferimento all'oggetto.

Non c'è bisogno di invocare il costruttore, perché la sua definizione è sufficiente per la creazione di un oggetto.

Se un oggetto di una classe semplice esiste già, non c'è bisogno di creare un oggetto di una classe composta costruita su di esso.

Se gli attributi di un oggetto sono inizializzati nella dichiarazione di classe, non c'è bisogno di creare oggetti.

La creazione di un oggetto di una classe composta crea automaticamente gli oggetti della classe semplice che appare come attributo nella composta.

A.8 Stato degli oggetti, attributi

Durante la chiamata di un metodo, l'attributo di un oggetto è duplicato come variabile locale. L'assegnamento li aggiorna entrambi.

Durante la chiamata di un metodo, l'attributo di un oggetto è duplicato come variabile locale. La variabile locale è inizializzata dall'oggetto, aggiornata dal metodo e ritornata dall'oggetto alla fine.

Durante la chiamata di un metodo, l'attributo di un oggetto è duplicato come variabile locale, che è inizializzata al valore di default. Gli assegnamenti hanno effetto solo sulla variabile locale, non sull'oggetto.

I parametri appartengono all'oggetto chiamato.

Le variabili locali appartengono all'oggetto chiamato.

Le variabili locali di un metodo appartengono all'oggetto chiamato.

Un oggetto può avere variabili di istanza di un solo tipo.

Non ci sono attributi di tipo oggetto all'interno di un oggetto; essi esistono solo localmente durante le chiamate dei metodi.

Gli attributi di una classe composta contengono gli attributi delle classi semplici, invece degli oggetti.

Gli attributi di una classe composta contengono gli attributi delle classi semplici oltre agli oggetti.

Un oggetto è un wrapper per una singola variabile. L'oggetto è equiparato alla variabile.

Gli attributi in una classe semplice sono replicati automaticamente un una classe composta.

Per cambiare il valore di un attributo di un oggetto di una classe semplice che è il valore di un attributo in un oggetto di una classe composta, bisogna costruire un nuovo oggetto.

Difficoltà a rendersi conto di quali proprietà rappresentano lo stato di un oggetto.

Il nome della variabile a cui l'oggetto è stato assegnato più di recente è parte dello stato di quell'oggetto.

Memorizzare un oggetto significa memorizzare una copia del sorgente della classe dell'oggetto.

Memorizzare un oggetto significa memorizzare i parametri passati al costruttore al momento della creazione. Questi parametri definiscono in modo non ambiguo l'oggetto. I metodi non modificano lo stato.

Si può definire un metodo che aggiunge un attributo alla classe.

Gli oggetti di una classe semplice, usati come valori per gli attributi di una classe composta, devono essere identici.

In una classe composta, si può scrivere un metodo che aggiunge un attributo della classe semplice alla classe composta.

In una classe composta, si può sviluppare un metodo che rimuove un attributo della classe semplice dalla classe composta.

Oggetti della stessa classe non possono avere valori uguali per un certo attributo.

Gli attributi di una classe semplice devono essere acceduti dalla classe composta invece che tramite un'interfaccia.

Agli oggetti viene allocata la stessa quantità di memoria, indipendentemente dalla definizione e istanziazione.

A.9 Metodi

Gli oggetti "sanno" quali metodi stanno agendo su di loro (mentre i metodi non "sanno" su quale oggetto stanno operando).

Non si possono avere metodi con lo stesso nome in classi diverse.

L'operatore "punto" può essere applicato ai metodi.

Si può definire un metodo che rimpiazza l'oggetto stesso.

Si può definire un metodo che distrugge l'oggetto stesso.

Si può definire un metodo che divide l'oggetto in due oggetti diversi.

I metodi possono solo fare assegnamenti.

I metodi della classe semplice non sono usati; invece vengono definiti e duplicati nuovi metodi equivalenti nella classe composta.

Si può invocare un metodo su un oggetto solo una volta.

Difficoltà a capire che un metodo può essere invocato su ogni oggetto della classe.

I metodi possono essere invocati solo su oggetti della classe composta, non su oggetti della classe semplice definiti come valori nei suoi attributi.

Dopo che una classe composta è stata definita, non possono essere definiti nuovi metodi nella classe semplice.

Un metodo deve essere sempre invocato su un oggetto esplicito.

Esistenza statica e dinamica dei metodi mischiata.

I metodi dichiarati in una classe semplice devono essere dichiarati ancora nella classe composta, per ognuno degli oggetti semplici.

A.10 Altre relative all'Object Oriented

Non c'è bisogno di *getter* e *setter* per gli attributi degli oggetti di una classe semplice dentro una classe composta.

L'*object lookup* consiste nel cercare in ogni oggetto in memoria un ID adatto.

Gli oggetti sono memorizzati nelle cartelle dell'hard disk e il *lookup* consiste nel cercarli in esse.

Un oggetto è solo una porzione di codice (non un elemento attivo a runtime).

Un oggetto è solo un record.

Un oggetto è un "modo di lavorare" fatto di azioni: espressioni, ritorno... Ha un flusso di controllo ma non uno stato.

Difficoltà a capire come una classe riconosce un'altra.

Difficoltà a comprendere l'ereditarietà (dei metodi).

Difficoltà a comprendere come il computer faccia a sapere quali sono i metodi e gli attributi di una classe.

Assegnamento confuso con subclassing.

Le gerarchie di ereditarietà esprimono le parti di una classe composta.

A.11 Altre

Difficoltà a capire l'effetto dell'input nelle chiamate di funzione durante l'esecuzione.

Confusione tra un array e le sue celle.

Difficoltà con gli array a due dimensioni e i loro indici.

Difficoltà con array che contengono indici come dati.

Il valore di espressioni condizionali viene stampato.

I numeri sono semplicemente numeri (perché distinguere `int` da `float`, per esempio?)

Un tipo è un insieme di vincoli sui valori.

I tipi possono cambiare al volo in Java.

Confusione tra i dati in memoria e i dati sullo schermo.

Il computer tiene in memoria quello che viene stampato (come parte dello stato?)

Confondere rappresentazioni testuali e numeriche (es. la stringa `''456''` con il numero)

I valori booleani sono qualcosa usato solo nelle espressioni condizionali e non sono dati come numeri o stringhe.

Una variabile di controllo di un `for` può vincolare i valori che possono essere dati in input all'interno del ciclo.

Bibliografia

[1] Alice. URL http://www.alice.org/.

[2] Arduino. URL http://www.arduino.cc/.

[3] Center for Computational Thinking at Carnegie Mellon University. URL http://www.cs.cmu.edu/~CompThink/index.html.

[4] CoderDojo. URL http://coderdojo.com/.

[5] Computational Thinking at NUI Maynooth. URL https://www.cs.nuim.ie/courses/compthink/.

[6] Computational Thinking Resources. URL http://csta.acm.org/Curriculum/sub/CompThinking.html.

[7] CS Principles. URL http://www.csprinciples.org/.

[8] MIT App Inventor. URL http://appinventor.mit.edu/explore/.

[9] Python. URL http://www.python.org/.

[10] Raspberry Pi. URL http://www.raspberrypi.org/about.

[11] Reform of the National Curriculum in England. URL http://www.ism.org/images/uploads/files/National_Curriculum_-_consultation_document.pdf.

[12] Scratch. URL http://scratch.mit.edu/.

[13] Wolfram Language and System Documentation Center. URL `http://reference.wolfram.com/language/`.

[14] Cognitive Artifacts (2006). URL `http://www.interaction-design.org/encyclopedia/cognitive_artifacts.html`.

[15] A. V. Aho. Computation and Computational Thinking. *The Computer Journal* 55(7):832–835 (2012).

[16] V. Allan, V. Barr, D. Brylow and S. Hambrusch. Computational thinking in high school courses. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 390–391. ACM, New York, NY, USA (2010).

[17] W. Allan, B. Coulter, J. Denner, J. Erickson, I. Lee, J. Malyn-Smith and F. Martin. Computational thinking for youth. *White Paper for the ITEST Small Working Group on Computational Thinking (CT)* (2010).

[18] S. K. Andrianoff and D. B. Levine. Role Playing in an Object-oriented World. *SIGCSE Bull.* 34(1):121–125 (2002).

[19] A. Ater-Kranov, R. Bryant, G. Orr, S. Wallace and M. Zhang. Developing a community definition and teaching modules for computational thinking: accomplishments and challenges. In *Proceedings of the 2010 ACM conference on Information technology education*, SIGITE '10, pages 143–148. ACM, New York, NY, USA (2010).

[20] R. Atkinson and R. Shiffrin. Human Memory: A Proposed System and its Control Processes. volume 2 of *Psychology of Learning and Motivation*, pages 89 – 195. Academic Press (1968).

[21] D. Barr, J. Harrison and L. Conery. Computational thinking: A digital age skill for everyone. *Learning & Leading with Technology* 38(6):20–52 (2011).

[22] V. Barr and C. Stephenson. Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community? *ACM Inroads* 2(1):48–54 (2011).

[23] A. Basawapatna, K. H. Koh, A. Repenning, D. C. Webb and K. S. Marshall. Recognizing computational thinking patterns. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 245–250. ACM (2011).

[24] T. Bell. Computer Science Unplugged. URL `http://csunplugged.org/`.

[25] M. Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching* 20(1):45–73 (2001).

[26] B. Bloom. *Taxonomy of Educational Objectives: The Classification of Educational Goals. Handbook I: Cognitive Domain.* David McKay Company (1956).

[27] J. Bonar and E. Soloway. Uncovering Principles of Novice Programming. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 10–13. ACM, New York, NY, USA (1983).

[28] R. Bornat, S. Dehnadi and Simon. Mental Models, Consistency and Programming Aptitude. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78*, ACE '08, pages 53–61. Australian Computer Society, Inc., Darlinghurst, Australia, Australia (2008).

[29] C. Brabrand and B. Dahl. Analyzing CS Competencies Using the SOLO Taxonomy. *SIGCSE Bull.* 41(3):1–1 (2009).

[30] K. Brennan, M. Chung and J. Hawson. Scratch Curriculum Guide Draft. URL `http://scratched.media.mit.edu/resources/scratch-curriculum-guide-draft`.

[31] K. Brennan and M. Resnick. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada* (2012).

[32] R. E. Bryant, K. Sutner and M. J. Stehlik. Introductory Computer Science Education at Carnegie Mellon University: A Deans' Perspective. Technical report, Carnegie Mellon University, Pittsburgh (2010).

[33] A. Bundy. Computational thinking is pervasive. *Journal of Scientific and Practical Computing* 1(2):67–69 (2007).

[34] E. Cambranes. Supporting novice programmers with natural language in the early stage of programming. In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, pages 173–174 (2013).

[35] F. Carugati and P. Selleri. *Psicologia dell'educazione*. Il Mulino (2001).

[36] M. E. Caspersen, K. D. Larsen and J. Bennedsen. Mental Models and Programming Aptitude. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '07, pages 206–210. ACM, New York, NY, USA (2007).

[37] L. N. Cassel. Interdisciplinary computing is the answer: now, what was the question? *ACM Inroads* 2(1):4–6 (2011).

[38] T.-Y. Chen, G. Lewandowski, R. McCartney, K. Sanders and B. Simon. Commonsense Computing: Using Student Sorting Abilities to Improve Instruction. *SIGCSE Bull.* 39(1):276–280 (2007).

[39] M. Clancy. Misconceptions and attitudes that interfere with learning to program. *Computer science education research* pages 85–100 (2004).

[40] Committee for the Workshops on Computational Thinking and National Research Council. *Report of a Workshop on the Pedagogical Aspects of Computational Thinking*. The National Academies Press (2011).

[41] Computing in the Core and Code.org. Computer Science Education Week. URL `http://csedweek.org/about`.

[42] B. J. Copeland. The Modern History of Computing. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2008 edition (2008).

[43] J. Cuny, L. Snyder and J. M. Wing. Computational Thinking: a definition (2010). (to appear).

[44] P. Curzon, J. Peckham, H. Taylor, A. Settle and E. Roberts. Computational thinking (CT): on weaving it in. *SIGCSE Bull.* 41(3):201–202 (2009).

[45] A. Cypher and D. C. Smith. KidSim: End User Programming of Simulations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 27–34. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1995).

[46] C. Day. Computational Thinking Is Becoming One of the Three Rs. *Computing in Science and Engineering* 13(1):88 (2011).

[47] P. J. Denning. Great principles of computing. *Commun. ACM* 46(11):15–20 (2003).

[48] P. J. Denning. Is computer science science? *Commun. ACM* 48(4):27–31 (2005).

[49] P. J. Denning. Computing is a natural science. *Communications of the ACM* 50(7):13–18 (2007).

[50] P. J. Denning. The profession of IT Beyond computational thinking. *Communications of the ACM* 52(6):28–30 (2009).

[51] P. J. Denning. Great Principles of Computing Website (2013). URL `http://denninginstitute.com/pjd/GP/GP-site/welcome.html`.

[52] P. J. Denning and P. S. Rosenbloom. The profession of IT: Computing: the fourth great domain of science. *Commun. ACM* 52(9):27–29 (2009).

[53] B. Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research* 2(1):57–73 (1986).

[54] A. Elliott Tew. *Assessing fundamental introductory computing concept knowledge in a language independent manner.* Ph.D. thesis, PhD dissertation, Georgia Institute of Technology, USA (2010).

[55] A. E. Fleury. Parameter Passing: The Rules the Students Construct. In *Proceedings of the Twenty-second SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '91, pages 283–286. ACM, New York, NY, USA (1991).

[56] A. E. Fleury. Programming in Java: Student-constructed Rules. *SIGCSE Bull.* 32(1):197–201 (2000).

[57] S. Furber. Shut down or restart? The way forward for computing in UK schools. *The Royal Society, London* (2012).

[58] M. Gabbrielli and S. Martini. *Linguaggi di programmazione: principi e paradigmi.* McGraw-Hill Italia, 2nd edition (2011).

[59] D. D. Garcia, C. M. Lewis, J. P. Dougherty and M. C. Jadud. If _, you might be a computational thinker! In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 263–264. ACM (2010).

[60] H. Goldstein and J. von Neumann. Planning and Coding Problems for an Electronic Computer Instrument. *Collected Works of J. von Neumann* 5:80–235 (1963).

[61] J. Good, K. Howland and K. Nicholson. Young People's Descriptions of Computational Rules in Role-Playing Games: An Empirical Study. In *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '10, pages 67–74. IEEE Computer Society, Washington, DC, USA (2010).

[62] Google. Computer Science for High School. URL `http://www.cs4hs.com/`.

[63] Google. Exploring Computational Thinking. URL `http://www.google.com/edu/computational-thinking/`.

[64] T. Greening. Emerging Constructivist Forces in Computer Science Education: Shaping a New Future? In *Computer science education in the 21st century*, pages 47–80. Springer (2000).

[65] S. Grover and R. Pea. Computational Thinking in K-12: A Review of the State of the Field. *Educational Researcher* 42(1):38–43 (2013).

[66] M. Guzdial. Education: Paving the Way for Computational Thinking. *Commun. ACM* 51(8):25–27 (2008).

[67] B. Haberman and Y. B.-D. Kolikant. Activating Black Boxes Instead of Opening Zipper - a Method of Teaching Novices Basic CS Concepts. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '01, pages 41–44. ACM, New York, NY, USA (2001).

[68] O. Hazzan. Reflections on teaching abstraction and other soft ideas. *SIGCSE Bull.* 40(2):40–43 (2008).

[69] D. Hemmendinger. A plea for modesty. *ACM Inroads* 1(2):4–7 (2010).

[70] P. B. Henderson. Ubiquitous computational thinking. *Computer* 42(10):100–102 (2009).

[71] P. B. Henderson, T. J. Cortina and J. M. Wing. Computational thinking. *SIGCSE Bull.* 39(1):195–196 (2007).

[72] K. Howland, J. Good and K. Nicholson. Language-based Support for Computational Thinking. In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, VLHCC '09, pages 147–150. IEEE Computer Society, Washington, DC, USA (2009).

[73] C. Hu. Computational thinking: what it might mean and what we might do about it. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, ITiCSE '11, pages 223–227. ACM, New York, NY, USA (2011).

[74] M. Hu, M. Winikoff and S. Cranefield. Teaching Novice Programming Using Goals and Plans in a Visual Notation. In *Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123*, ACE '12, pages 43–52. Australian Computer Society, Inc., Darlinghurst, Australia, Australia (2012).

[75] Interim Review Task Force. Computer Science Curriculum 2008. Technical report, Association for Computing Machinery (ACM) & IEEE Computer Society (2008).

[76] International Society for Technology in Education and Computer Science Teachers Association. Computational Thin-

king Teacher Resources, 2nd edition (2011). URL `http://csta.acm.org/Curriculum/sub/CurrFiles/CompThinkingFlyer.pdf`.

[77] International Society for Technology in Education and Computer Science Teachers Association. Operational Definition of Computational Thinking for K-12 Education (2011). URL `http://csta.acm.org/Curriculum/sub/CurrFiles/CompThinkingFlyer.pdf`.

[78] A. Ioannidou, V. Bennett, A. Repenning, K. H. Koh and A. Basawapatna. Computational Thinking Patterns. In *Annual Meeting of the American Educational Research Association 2011* (2011).

[79] L. Kaczmarczyk, R. Dopplick and E. P. Committee. Rebooting the Pathway to Success - Preparing Students for Computing Workforce Needs in the United States. Technical report, Association for Computing Machinery (ACM) (2014).

[80] L. C. Kaczmarczyk, E. R. Petrick, J. P. East and G. L. Herman. Identifying Student Misconceptions of Programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 107–111. ACM, New York, NY, USA (2010).

[81] C. Kelleher and R. Pausch. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Comput. Surv.* 37(2):83–137 (2005).

[82] R. Knöll, V. Gasiunas and M. Mezini. Naturalistic Types. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ONWARD '11, pages 33–48. ACM, New York, NY, USA (2011).

[83] R. Knöll and M. Mezini. Pegasus: First Steps Toward a Naturalistic Programming Language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 542–559. ACM, New York, NY, USA (2006).

[84] K. H. Koh, A. Basawapatna, V. Bennett and A. Repenning. Towards the Automatic Recognition of Computational Thinking for Adaptive Visual Language Learning. In *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on*, pages 59–66 (2010).

[85] J. Kramer. Is abstraction the key to computing? *Commun. ACM* 50(4):36–42 (2007).

[86] D. R. Krathwohl. A Revision of Bloom's Taxonomy: An Overview. *Theory Into Practice* 41(4):212–218 (2002).

[87] W. M. Kunkle. *The Impact of Different Teaching Approaches and Languages on Student Learning of Introductory Programming Concepts.* Ph.D. thesis, Philadelphia, PA, USA (2010). AAI3430595.

[88] G. Lakoff. *Women, fire, and dangerous things : what categories reveal about the mind.* University of Chicago Press, 1997 edition (1987).

[89] I. Lee, F. Martin, J. Denner, B. Coulter, W. Allan, J. Erickson, J. Malyn-Smith and L. Werner. Computational thinking for youth in practice. *ACM Inroads* 2(1):32–37 (2011).

[90] G. Lewandowski, D. J. Bouvier, R. McCartney, K. Sanders and B. Simon. Commonsense Computing (Episode 3): Concurrency and Concert Tickets. In *Proceedings of the Third International Workshop on Computing Education Research*, ICER '07, pages 133–144. ACM, New York, NY, USA (2007).

[91] M. C. Linn and M. J. Clancy. The Case for Case Studies of Programming Problems. *Commun. ACM* 35(3):121–132 (1992).

[92] R. Lister. Concrete and Other neo-Piagetian Forms of Reasoning in the Novice Programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, pages 9–18. Australian Computer Society, Inc., Darlinghurst, Australia, Australia (2011).

[93] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon and L. Thomas. A Multi-national Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '04, pages 119–150. ACM, New York, NY, USA (2004).

[94] J. J. Lu and G. H. Fletcher. Thinking about computational thinking. In *ACM SIGCSE Bulletin*, volume 41, pages 260–264. ACM (2009).

[95] S. Madison and J. Gifford. Parameter Passing: The Conceptions Novices Construct. (1997).

[96] M. A. Martìnez, N. Sauleda and G. L. Huber. Metaphors as blueprints of thinking about teaching and learning. *Teaching and Teacher Education* 17(8):965–977 (2001).

[97] S. Martini. Elogio di Babele. *Mondo digitale* 2 (2008).

[98] S. Martini. Lingua Universalis. *Annali della Pubblica Istruzione* 4:65–70 (2012).

[99] L. Mason. *Psicologia dell'apprendimento e dell'istruzione*. Il Mulino (2007).

[100] R. McCartney, D. J. Bouvier, T.-Y. Chen, G. Lewandowski, K. Sanders, B. Simon and T. VanDeGrift. Commonsense Computing (Episode 5): Algorithm Efficiency and Balloon Testing. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 51–62. ACM, New York, NY, USA (2009).

[101] J. J. Medina. *Il cervello: istruzioni per l'uso*. Bollati Boringhieri (2013).

[102] L. A. Miller. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal* 20(2):184–215 (1981).

[103] B. A. Myers, J. F. Pane and A. Ko. Natural Programming Languages and Environments. *Commun. ACM* 47(9):47–52 (2004).

[104] National Research Council. *Report of a Workshop on the Scope and Nature of Computational Thinking*. National Academies Press (2010).

[105] National Science Foundation. Exploring the frontiers of computing - Fundings. URL `https://www.nsf.gov/dir/index.jsp?org=cise`.

[106] D. A. Norman. Designing Interaction. chapter Cognitive Artifacts, pages 17–38. Cambridge University Press, New York, NY, USA (1991).

[107] J. Pane and B. Myers. The Influence of the Psychology of Programming on a Language Design: Project Status Report. *Institute for Software Research* (2000).

[108] J. F. Pane, B. A. Myers and C. A. Ratanamahatana. Studying the Language and Structure in Non-programmers' Solutions to Programming Problems. *Int. J. Hum.-Comput. Stud.* 54(2):237–264 (2001).

[109] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA (1980).

[110] S. Papert. An exploration in the space of mathematics educations. *International Journal of Computers for Mathematical Learning* 1(1):95–123 (1996).

[111] R. Paul and L. Elder. *The miniature guide to critical thinking: Concepts & tools*, volume 2. Foundation Critical Thinking (2001).

[112] R. Pea. Language-independent conceptual bugs in novice programming. *Journal of Educational Computing Research* 2:25–36 (1986).

[113] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin and R. Simmons. Conditions of learning in novice programmers. *Journal of Educational Computing Research* 2(1):37–55 (1986).

[114] L. Perković, A. Settle, S. Hwang and J. Jones. A framework for computational thinking across the curriculum. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, ITiCSE '10, pages 123–127. ACM, New York, NY, USA (2010).

[115] A. J. Perlis. The computer in the university. *Computers and the World of the Future* pages 180–219 (1962).

[116] J. E. Pfeiffer. *The thinking machine*. Lippincott (1962).

[117] G. Posner, K. Strike, P. Hewson and W. Gertzog. Accommodation of a scientific conception: Toward a theory of conceptual change. *Sci. Ed.* 66(2):211–227 (1982).

[118] H. Qin. Teaching computational thinking through bioinformatics to biology students. *SIGCSE Bull.* 41(1):188–191 (2009).

[119] N. Ragonis and M. Ben-Ari. A long-term investigation of the comprehension of OOP concepts by novices (2005).

[120] N. Ragonis and M. Ben-Ari. On Understanding the Statics and Dynamics of Object-oriented Programs. *SIGCSE Bull.* 37(1):226–230 (2005).

[121] D. A. Reed, R. Bajcsy, M. A. Fernandez, J.-M. Griffiths, R. D. Mott, J. Dongarra, C. R. Johnson, A. S. Inouye, W. Miner, M. K. Matzke et al. Computational science: ensuring America's competitiveness. Technical report, DTIC Document (2005).

[122] A. Repenning and A. Ioannidou. Broadening Participation Through Scalable Game Design. *SIGCSE Bull.* 40(1):305–309 (2008).

[123] A. Repenning, D. Webb and A. Ioannidou. Scalable game design and the development of a checklist for getting computational thinking into public schools. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 265–269. ACM, New York, NY, USA (2010).

[124] R. S. Rist. Learning to Program: Schema Creation, Application, and Evaluation. *Computer science education research* (2004).

[125] P. S. Rosenbloom. *On Computing: The Fourth Great Scientific Domain.* The MIT Press (2012).

[126] B. A. Sheil. The Psychological Study of Programming. *ACM Comput. Surv.* 13(1):101–120 (1981).

[127] B. Simon, D. Bouvier, T.-Y. Chen, G. Lewandowski, R. McCartney and K. Sanders. Common sense computing (episode 4): debugging. *Computer Science Education* 18(2):117–133 (2008).

[128] B. Simon, T.-Y. Chen, G. Lewandowski, R. McCartney and K. Sanders. Commonsense Computing: What Students Know Before We Teach (Episode 1: Sorting). In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 29–40. ACM, New York, NY, USA (2006).

[129] T. Sirkiä. *Recognizing Programming Misconceptions*. Master's thesis, Aalto University (2012).

[130] T. Sirkiä and J. Sorva. Exploring Programming Misconceptions: An Analysis of Student Mistakes in Visual Program Simulation Exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, Koli Calling '12, pages 19–28. ACM, New York, NY, USA (2012).

[131] J. Smith, A. diSessa and J. Roschelle. Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition. *Journal of the Learning Sciences* 3(2):115–163 (1994).

[132] E. Soloway. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29(9):850–858 (1986).

[133] J. Sorva. *Visual program simulation in introductory programming education*. Ph.D. thesis, Aalto University (2012).

[134] J. Sorva. Notional Machines and Introductory Programming Education. *Trans. Comput. Educ.* 13(2):8:1–8:31 (2013).

[135] J. G. Spohrer and E. Soloway. Analyzing the High Frequency Bugs in Novice Programs. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, pages 230–251. Ablex Publishing Corp., Norwood, NJ, USA (1986).

[136] D. Teague and R. Lister. Neo-Piagetian Preoperational Reasoning in a Novice Programmer. In *Psychology of Programming Interest Group (PPIG) Work-in-Progress Workshop 2013*, page 10 (2013).

[137] The Joint Task Force on Computing Curricula. Computer Science Curricula 2013 - Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. Technical report, Association for Computing Machinery (ACM) & IEEE Computer Society (2013).

[138] T. VanDeGrift, D. Bouvier, T.-Y. Chen, G. Lewandowski, R. McCartney and B. Simon. Commonsense Computing (Episode 6): Logic is Harder Than Pie. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 76–85. ACM, New York, NY, USA (2010).

[139] M. Y. Vardi. Science has only two legs. *Commun. ACM* 53(9):5–5 (2010).

[140] E. Visser. Understanding Software through Linguistic Abstraction. *Science of Computer Programming* (2013). (to appear).

[141] M. G. Voskoglou and S. Buckley. Problem Solving and Computational Thinking in a Learning Environment. *CoRR* (2012).

[142] P. C. Wason. The processing of positive and negative information. *Quarterly Journal of Experimental Psychology* 11(2):92–107 (1959).

[143] A. E. Weinberg. *Computational Thinking: An Investigation Of The Existing Scholarship And Research.* Ph.D. thesis, Colorado State University (2013).

[144] C. Wilson, L. Sudol, C. Stephenson and M. Stehlik. Running on empty: The failure to teach K-12 computer science in the digital age. Association for Computing Machinery. *Computer Science Teachers Association* (2010).

[145] J. M. Wing. Computational thinking. *Communications of the ACM* 49(3):33–35 (2006).

[146] J. M. Wing. Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366(1881):3717–3725 (2008).

[147] J. M. Wing. Computational Thinking: What and Why? *Link Magazine* (2010).

[148] K. Wynn. Psychological foundations of number: numerical competence in human infants. *Trends in Cognitive Sciences* 2(8):296 – 303 (1998).

Ringraziamenti

I can no other answer make, but, thanks, and thanks.
– William Shakespeare

Ho deciso di iscrivermi a Informatica dopo aver seguito una sua conferenza ad AlmaOrienta, e ho davvero capito "cosa ci sta dietro" dopo il suo corso di Paradigmi; infine l'ho convinto a seguirmi (e ad aspettarmi) nell'esplorazione della Didattica dell'Informatica: il mio primo ringraziamento non può che andare al mio relatore, il Chiarissimo, di diritto e di fatto, Professor Simone Martini.

Un grazie di cuore al caro Dott. Ugo Dal Lago, mio relatore della laurea triennale, che ha sopportato i miei ripensamenti e mi ha indirizzato sulla strada giusta.

Un grazie "preventivo" anche al Prof. Renzo Davoli, che - ne sono sicuro - la leggerà con attenzione e interesse.

Grazie ai miei genitori, perché senza i loro sacrifici non avrei potuto prendermi il tempo per fare ciò che volevo fare.

Grazie alla mia Tata e alla sua famiglia, che insieme mi regalano sempre viaggi indimenticabili. Grazie alla mia Nonna che, silenziosamente e affettuosamente, mi "finanzia". Grazie allo Zio e grazie alla Zia, che mi propone sfide informatiche sempre più ardue.

Grazie ad Andrea, che mi supporta e mi sopporta nei miei sproloqui e grazie a Laura, per le impagabili risate nelle numerose (ma non maliziose) "cose a tre". Grazie a Giacomo, che potrebbe essere il quarto se solo riuscisse ad inventare qualche ora in più nella giornata, ma che è sempre nel mio cuore.

Grazie ad Alessandro, che mi riporta sempre alla scientificità del mondo e che non mi nega mai assurde discussioni, stimolanti e divertenti.

Grazie a Miriam, che mi ricorda quanto due persone possano essere diverse eppure amiche, e che mi chiede quando mi laureo dal Novembre 2012 ;)

Grazie a Morelli, Battista e Mantovani, che forse la smetteranno di dire che sono uno studente fannullone... per iniziare a dire che sono un laureato fannullone :)

Grazie alle amiche e agli amici vecchi e nuovi: Catia, Simona, Eva, Giulio, Serena, Lara, Giulia, Yas, Chiara, Antonio e Alessio, Irene, Elisa, Flavia e Daniele, che mi regalano sempre momenti divertenti e a tutti gli altri che - dimenticati - sono autorizzati a picchiarmi.

Grazie a Valeria, la mia informatica teorica preferita! Grazie a Domiziana, Federica, Francesco, Federico, Pietro, Carlo, Lorenzo e Filippo, e a tutti gli altri compagni di Università: senza di voi questo viaggio sarebbe stato di certo meno divertente e più faticoso.

Grazie a Carmelo, perché mi ricorda che "c'è il mondo reale là fuori" e che mi fa vivere, insieme agli altri Mentor, esperienze didattiche sempre nuove e stimolanti.

Grazie a Grazia, a Lucrezia (e a tutte le persone i cui nomi terminano in -zia), ai bambini di CoderDojo e ai ragazzi di Rimini, cavie inconsapevoli per questa tesi.

Infine grazie a Valentina, a cui, nonostante tutto, voglio tanto bene :)