Apprendimento della programmazione guidato dalla necessità: il Necessity Learning Design

Marco Sbaraglia, Michael Lodi, Simone Martini

Dipartimento di Informatica - Scienza e Ingegneria
Alma Mater Studiorum - Università di Bologna
Bologna, Italia
{marco.sbaraglia, michael.lodi, simone.martini}@unibo.it

Abstract

Descriviamo una metodologia didattica originale, il Necessity Learning Design, in cui gli studenti cercano di risolvere un problema per il quale serve un concetto di programmazione che ancora non conoscono, sentendone quindi la necessità. Le fasi di spiegazione e di soluzione seguono. Descriviamo i primi risultati di una sperimentazione di questa metodologia in un istituto tecnico informatico.

1 Il momento giusto per insegnare

Alcune osservazioni di Boaler [1, pp. 66-69] sono illuminanti: vengono posti problemi di matematica (es. recintare l'area maggiore possibile con 36 pezzi di staccionata, trovare il volume di un limone) a studenti che ancora non hanno tutti gli strumenti necessari a risolverli (es. trigonometria, integrali). Solo dopo che gli studenti hanno sperimentato, fatto ipotesi, tentato strategie di soluzione, gli insegnanti hanno spiegato i concetti necessari, ricevendo grande interesse e attenzione (Boaler racconta di un ragazzino che spiega euforico al suo gruppo "una cosa molto *cool* che ha imparato dall'insegnante": come usare la funzione trigonometrica seno). In altre classi Boaler aveva osservato lezioni tradizionali sugli stessi concetti, bollate dagli studenti come noiose.

In didattica della matematica e delle scienze si parla in questi casi di approcci dove il Problem Solving precede l'Insegnamento diretto (PS-I) [10]. Il titolo di uno dei lavori su questo tema è evocativo: *A time for telling* [15], che potremmo tradurre come "Il momento giusto di spiegare". Sperimentare difficoltà e fallire può stimolare la motivazione e preparare gli studenti all'apprendimento (es. tramite attivazione della loro conoscenza pregressa, favorendo una presa di consapevolezza delle lacune o il riconoscimento delle caratteristiche fondamentali di un problema) [10, 11]. L'approccio più conosciuto e sperimentato è quello del *Productive failure* [5, 6], in cui agli studenti viene chiesto di risolvere un problema complesso, mal strutturato, con dettagli inutili o sovrabbondanti, che ostacolano una soluzione "canonica". Segue una fase di insegnamento diretto, basata sul confronto delle soluzioni sub-ottime realizzate dagli studenti.

La maggior parte della ricerca educativa sostiene che le metodologie attive – specialmente quando gli studenti esplorano e costruiscono attivamente la conoscenza – favoriscono l'apprendimento [3, 12]. D'altra parte, così come in altre discipline, anche nell'introduzione alla programmazione è difficile far scoprire specifici concetti tecnici tramite l'esplorazione libera [4]. Di conseguenza, nei corsi introduttivi, resta comune l'insegnamento diretto degli elementi del linguaggio, seguito dalla loro applicazione in esercizi di programmazione. L'insegnamento diretto non sembra però ideale per i novizi: potrebbero annoiarsi e non cogliere in concreto l'utilità dei concetti [2], con conseguente bassa motivazione e scarsi risultati.

Abbiamo applicato le idee del PS-I al contesto specifico dell'introduzione alla programmazione, formalizzando l'idea generale del "meccanismo di necessità" e progettando una metodologia didattica specifica per l'Informatica che abbiamo chiamato *Necessity Learning Design* (NLD) [14].

Riassumiamo qui gli aspetti fondamentali del nostro modello [14], descrivendo poi i primi risultati di una sua sperimentazione alla scuola secondaria.

2 La necessità

2.1 Il meccanismo di necessità

Il *meccanismo di necessità* consiste nell'assegnare un problema progettato in modo che sia percepito come familiare perché simile (nelle parole, forma, richiesta) a molti già svolti in precedenza e quindi padroneggiati dagli studenti. Gli studenti capiscono facilmente il problema e le sue richieste, e credono di saperlo risolvere. Tuttavia, il problema è costruito in modo che manchi esattamente un solo elemento per poterlo risolvere: il concetto che si vuole insegnare (*concetto target*). Si tratta di un meccanismo generale, utilizzabile e adattabile (come abbiamo fatto ad esempio per la crittografia [7, 8, 9]) ad altri contesti o discipline.

Per fare un esempio, dopo che gli studenti hanno appreso costrutti per la scansione di sequenze (`foreach`) e l'iterazione determinata (`for` con indice), si può assegnare un problema risolvibile solo[1] con l'iterazione indeterminata (`while`), che essi ancora non conoscono, ad esempio:

> contare dopo quante generazioni di un numero random esce il numero 42.

2.2 Il Necessity Learning Design

Una sequenza di apprendimento progettata tramite il Necessity Learning Desing, la nostra metodologia didattica specifica per l'introduzione della programmazione che fa uso del meccanismo di necessità, è costituita da tre fasi successive: P!S, I, PS (Figura 1).

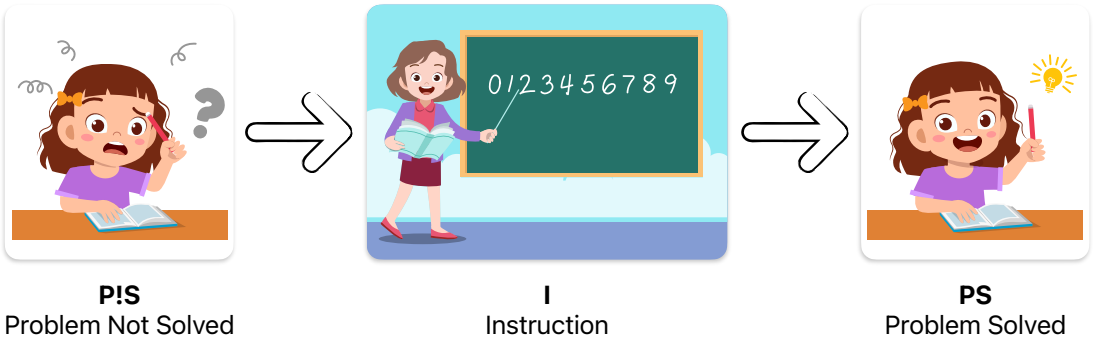

Figura 1: Le fasi di una sequenza di apprendimento progettata con il Necessity Learning Design.

Le immagini sono state acquistate con licenza *Shutterstock Standard Image License* (https://www.shutterstock.com/license).

P!S Si assegna un semplice esercizio, apparentemente familiare, che lo studente non sa risolvere senza il concetto target. Poiché il contesto e la difficoltà del problema sono simili a quelli che lo studente ha già affrontato, sentirà la necessità di qualcosa che ancora non conosce, ma di cui può intuire caratteristiche

[1] Pensiamo a linguaggi come Python in cui le iterazioni con il `for` sono sostanzialmente realmente *determinate* - solo con stratagemmi non ovvi per i novizi si può sfruttare il `for` per iterazioni indeterminate. Diversa la situazione per linguaggi quali il C, in cui un `for` è di fatto un'abbreviazione per un `while`. Ovviamente supponiamo anche che gli studenti non conoscano ancora la ricorsione.

e utilità. Poiché i programmi sono oggetti eseguibili, lo studente potrà verificare autonomamente la correttezza del proprio tentativo di soluzione.

I L'insegnante spiega direttamente il concetto target, fornisce esempi semplici e introduttivi, ma senza riferirsi all'esercizio della fase P!S, per non sprecare il potenziale di apprendimento insito nella necessità sperimentata in precedenza dallo studente.

PS Lo studente soddisfa la propria necessità applicando il concetto target alla risoluzione dell'esercizio iniziale. Avendolo già esplorato ampiamente, può giovarsi dei ragionamenti fatti durante i tentativi iniziali. Può inoltre comprendere più facilmente l'importanza del concetto target e la sua applicazione in uno specifico contesto.

2.2.1 Quando usare il NLD

Questa metodologia è stata ideata per introdurre (fare "bootstrap") in modo motivante e significativo nuovi concetti fondamentali (es. nuovi costrutti della programmazione). Non è adatta ad essere utilizzata in tutte le fasi dell'apprendimento, in cui vanno utilizzate altre metodologie per favorire lo sviluppo della padronanza.

Riteniamo utile usare NLD nei momenti in cui cambia il livello di astraizone all'interno di un linguaggio, ad esempio quando si introduce un costrutto più astratto (es. array invece di variabili scollegate) o uno meno astratto (es. `while` dopo il `for`).

2.2.2 Esempio: gli array

Una possibile sequenza P!S-I-PS, descritta in [14, §4.4.3], viene preceduta da una fase di avvicinamento in cui gli studenti, usando variabili semplici, ciclo `for` e numeri casuali, maturano padronanza su esercizi del tipo:

- contare teste e croci in 10.000 lanci casuali di una moneta;
- contare quante volte esce ciascuna faccia in 1 milione di lanci di un dado a 6 facce.

Nella fase P!S, poi, viene chiesto agli studenti di

- contare quante volte esce ciascun numero del Lotto (da 0 a 89) in 1 milione di estrazioni.

Sebbene questo esercizio abbia una soluzione sub-ottima che fa uso di 90 variabili (es. `estratto1`, ... `estratto90`) mediante un'altrettanto complessa struttura condizionale, gli studenti dovrebbero sentire la necessità di "una struttura di dati per raccogliere la frequenza con cui esce ogni numero, accedendo ai valori di tale struttura a runtime sulla base del numero estratto"[2].

Nella fase I, l'insegnante introduce gli array e la loro scansione con il `for` con indice.

Nella fase di PS, gli studenti possono risolvere il problema utilizzando un array di 90 interi, in cui ogni cella conterà la frequenza di estrazioni del numero corrispondente all'indice di tale cella nell'array.

[2] Ovviamente ci aspettiamo che gli studenti abbiano un'idea molto meno formale e intuitiva, ma comunque coerente, di ciò di cui hanno bisogno, come vedremo nella prossima sezione.

3 La sperimentazione

Abbiamo sperimentato la sequenza per introdurre gli array (vedi 2.2.2) in un Istituto Tecnico Tecnologico bolognese, articolazione Informatica. Il quasi-esperimento ha coinvolto due classi terze (stessi insegnante e ITP) di studenti alle prime armi con la programmazione. La classe sperimentale ha seguito la sequenza P!S-I-PS di NLD, la classe di controllo una classica sequenza I-PS.

Riportiamo qui le analisi qualitative delle esperienze di insegnanti, studenti (classe sperimentale) e ricercatori. Per altre analisi più dettagliate si veda [13, capitoli 9 e 10]. Ulteriori analisi, qualitative e quantitative, sono in corso.

La classe sperimentale (più fragile della classe di controllo) mostrava storicamente scarsi interesse e risultati e ha mantenuto un atteggiamento disinteressato e passivo durante gli esercizi di avvicinamento alla fase P!S (vedi 2.2.2). Gli studenti si sono animati nella fase P!S (problema del Lotto), formando spontaneamente gruppi per cercare soluzioni. Tutti si sono impegnati sul problema, chi prendendolo come sfida, chi cercando online, chi procedendo in modo "cieco" cercando – senza riuscirci per le difficoltà sintattiche di questa strategia – di risolverlo senza usare gli array. Tutti gli studenti hanno sentito la *necessità* di qualcosa in più (es., cambiare dinamicamente il nome di una variabile); alcuni hanno provato frustrazione, in particolare quelli abituati a buoni risultati.

Nella fase I l'insegnante ha introdotto gli array in modo generale e gli studenti sono rimasti focalizzati. Alcuni, però, cercando di applicare immediatamente gli array al problema del Lotto, hanno perso passaggi chiave della spiegazione. Un setting più adatto alla lezione frontale (lontano dai computer) potrebbe favorire l'attenzione nella fase I.

Anche nella fase PS gli studenti sono rimasti focalizzati sul problema da risolvere (comportamento insolito, a detta dell'insegnante). Tuttavia, la maggioranza non è riuscita a risolverlo pur usando gli array. Alcuni, come detto, avevano perso dettagli fondamentali della spiegazione, altri sono stati ostacolati dall'ulteriore difficoltà di usare il numero estratto come indice dell'array. Questo sottolinea che il design di una sequenza NLD, destinata alla programmazione introduttiva, deve prevedere un solo nuovo concetto/costrutto/pattern.

Anche se da confermare con analisi quantitative, la classe sperimentale ha lievemente migliorato i propri risultati di apprendimento rispetto agli argomenti precedenti.

In generale, la maggioranza degli studenti ha apprezzato la sfida e ne ha compreso il potenziale educativo. Alcuni, tuttavia, si sono sentiti ingannati o troppo frustrati.

È fondamentale che l'insegnante gestisca bene i tempi, evitando che la fase P!S duri troppo, sfruttando invece il "momento giusto per insegnare". Inoltre, per evitare che gli studenti "mangino la foglia", è bene ricorrere a NLD poche volte, solo per introdurre nuovi concetti importanti o difficili. Una delle maggiori difficoltà per l'insegnante è quella di "mantenere il segreto" del concetto target (prima e durante la fase P!S), senza però frustrare eventuali studenti che dovessero già conoscerlo o scoprirlo.

Fondi

Il lavoro di M. Lodi è supporrato dallo Spoke 1 "FutureHPC & BigData" del Centro Nazionale di Ricerca in "High Performance Computing, Big Data and Quantum Computing" (ICSC) finanziato da MUR Missione 4 Componente 2 Investimento 1.4: Potenziamento strutture di ricerca e creazione di campioni nazionali di R&S (M4C2-19) - Next Generation EU (NGEU).

Bibliografia

[1] Jo Boaler. *Mathematical mindsets: unleashing students' potential through creative math, inspiring messages, and innovative teaching*. San Francisco, CA: Jossey-Bass & Pfeiffer Imprints, 2016. ISBN: 9781118418277 9781118415535.

[2] Michael E. Caspersen. ≪Teaching Programming≫. In: *Computer science education: perspectives on teaching and learning in school*. A cura di Sue Sentance, Erik Barendsen e Carsten Schulte. London-New York: Bloomsbury Academic, 2018, pp. 109–130. ISBN: 9781350057111 9781350057104.

[3] Scott Freeman et al. ≪Active learning increases student performance in science, engineering, and mathematics≫. In: *Proceedings of the National Academy of Sciences* 111.23 (2014), pp. 8410–8415. DOI: 10.1073/pnas.1319030111.

[4] Mark Guzdial. ≪Balancing Teaching CS Efficiently with Motivating Students≫. In: *Commun. ACM* 60.6 (mag. 2017), pp. 10–11. ISSN: 0001-0782. DOI: 10.1145/3077227.

[5] Manu Kapur. ≪Examining Productive Failure, Productive Success, Unproductive Failure, and Unproductive Success in Learning≫. In: *Educational Psychologist* 51.2 (2016), pp. 289–299. DOI: 10.1080/00461520.2016.1155457.

[6] Manu Kapur e Katerine Bielaczyc. ≪Designing for Productive Failure≫. In: *Journal of the Learning Sciences* 21.1 (2012), pp. 45–83. DOI: 10.1080/10508406.2011.591717.

[7] Michael Lodi, Simone Martini e Marco Sbaraglia. ≪Crittografia a blocchi al Liceo Matematico≫. In: *Cryptography and Coding Theory Conference 2021*. Vol. 3. Collectio Ciphrarum. Roma, Italy: Aracne, 2022, pp. 103–104. DOI: 10.53136/979125994981340. URL: https://www.aracneeditrice.eu/it/pubblicazioni/estratti/10.53136/979125994981340-crittografia-a-blocchi-al-liceo-matematico-estratto.html.

[8] Michael Lodi, Simone Martini e Marco Sbaraglia. ≪Programmare per imparare la crittografia al Liceo Matematico≫. Italian. In: *Rendiconti del Seminario Matematico* 80.2 (2022). URL: http://www.seminariomatematico.polito.it/rendiconti/80-2/Lodi.pdf.

[9] Michael Lodi, Marco Sbaraglia e Simone Martini. ≪Cryptography in Grade 10: Core Ideas with Snap! and Unplugged≫. In: *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol. 1*. ITiCSE '22. Dublin, Ireland: Association for Computing Machinery (ACM), 2022. ISBN: 978-1-4503-9201-3/22/07. DOI: 10.1145/3502718.3524767.

[10] Katharina Loibl, Ido Roll e Nikol Rummel. ≪Towards a Theory of When and How Problem Solving Followed by Instruction Supports Learning≫. English. In: *Educational Psychology Review* 29.4 (dic. 2017), pp. 693–715. ISSN: 1040-726X, 1573-336X. DOI: 10.1007/s10648-016-9379-x.

[11] Katharina Loibl e Nikol Rummel. ≪The impact of guidance during problem-solving prior to instruction on students' inventions and learning outcomes≫. In: *Instructional Science* 42 (mag. 2014). DOI: 10.1007/s11251-013-9282-5.

[12] Michael Prince. ≪Does Active Learning Work? A Review of the Research≫. In: *Journal of Engineering Education* 93.3 (lug. 2004), pp. 223–231. DOI: 10.1002/j.2168-9830.2004.tb00809.x.

[13] Marco Sbaraglia. ≪Teaching Informatics to Novices: Big Ideas and The Necessity of Optimal Guidance≫. Tesi di Dottorato. Alma Mater Studiorun - Università di Bologna, 2023. URL: http://amsdottorato.unibo.it/10914/.

[14] Marco Sbaraglia, Michael Lodi e Simone Martini. ≪A Necessity-Driven Ride on the Abstraction Rollercoaster of CS1 Programming≫. In: *Informatics in Education* 20.4 (dic. 2021), pp. 641–682. ISSN: 1648-5831. DOI: 10.15388/infedu.2021.28.

[15] Daniel L. Schwartz e John D. Bransford. ≪A Time For Telling≫. In: *Cognition and Instruction* 16.4 (dic. 1998), pp. 475–5223. DOI: 10.1207/s1532690xci1604_4.