UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Corso di Laurea Magistrale in Informatica

Apprendimento della programmazione: analisi delle difficoltà e sviluppo di strategie didattiche

Tesi di Laurea di:
UMBERTO COSTANTINI
Matr. 909001

Relatore:
VIOLETTA LONATI

Correlatore:
ANNA MORPURGO

Anno Accademico 2018-2019

Capitolo 1

Stato dell'arte

In questo capitolo vengono presentati alcuni temi molto studiati nell'ambito della didattica della programmazione. Nella sezione 1.1 sono esaminate le difficoltà tipiche incontrate dagli studenti quando approcciano la programmazione e le loro possibili cause. Nella sezione 1.2 sono presentati vari modelli cognitivi per l'apprendimento della programmazione descritti in letteratura, basati sugli elementi che differenziano i programmatori esperti dai programmatori principianti. Nella sezione 1.3 si discute della relazione tra l'abilità di comprensione del codice e l'abilità di scrittura del codice. Nella sezione 1.4 si illustrano strategie didattiche per insegnare come progettare programmi a partire dalle soluzioni. Nella sezione 1.5 infine, vengono presentate alcune modalità di valutazione dell'apprendimento della programmazione.

1.1 Difficoltà dei principianti

Il corso di programmazione di base, erogato al primo anno dei corsi di laurea in Informatica, è ritenuto un corso difficile e con alti tassi di insuccesso e abbandono in tutto il mondo [7]. L'apprendimento della programmazione è visto come un processo complicato al punto che "le frasi *tasso di insuccesso* e *corso di programmazione* sono quasi sinonimi" [3]. L'alto tasso di insuccesso è citato in numerose altre ricerche [23, 42] ma gli unici ad aver provato a darne una stima sono stati Bennedsen [3] e, successivamente, Watson [59] che ha proposto una rivisitazione del suo lavoro; in entrambi gli studi il tasso di insuccesso rilevato

Scrivi un programma che legga una sequenza di interi e si fermi quando legge 99999. Dopo aver letto 99999, deve stampare la media corretta. Ovvero, non deve contare il 99999 finale.

Figura 1.1: Rainfall Problem

è di circa 33%. Lo stesso Bennedsen, constatando che l'argomento era ancora centrale per molti ricercatori e che non erano presenti dati a riguardo oltre ai due lavori appena citati, nel 2019 ha ripetuto lo studio su un campione più ampio, riscontrando un tasso di insuccesso del 28% [4]. Al confronto con quelli di altri corsi, i tassi di insuccesso misurati non sono ritenuti allarmanti ma sicuramente possono essere ridotti, soprattutto al fine di migliorare la percezione del corso di programmazione di base, e più in generale dei corsi di informatica, e quindi attrarre un maggior numero di studenti.

Uno dei primi ad investigare le difficoltà incontrate dagli studenti è stato Soloway in una serie di studi [51, 6, 52]. L'esempio maggiormente utilizzato da Soloway nelle sue analisi è il problema chiamato *Rainfall Problem*, presentato in Figura 1.1.

Soloway ha dimostrato che questo problema, seppur apparentemente banale, risulta difficile per i programmatori principianti: in uno degli studi, solamente il 14% degli studenti di Yale riuscì alla fine del corso di programmazione a risolvere il problema correttamente. Come riportato da Guzdial [19], lo stesso problema è stato riproposto in numerosi altri studi che hanno confermato le difficoltà degli studenti, ottenendo risultati simili a Soloway.

I libri di testo di programmazione di base si concentrano tipicamente sulla sintassi e sulla semantica dei costrutti. Soloway suggerisce invece che il vero ostacolo per i principianti non siano questi aspetti, quanto il *mettere insieme tutti i pezzi*, comporre e coordinare le varie componenti di un programma. Questa capacità è ciò che secondo Soloway differenzia maggiormente un programmatore esperto da un principiante.

Nel 2001, un gruppo di lavoro internazionale inter-istituzionale guidato da Michael McCraken [31], si è posto l'obiettivo di investigare le capacità sviluppate dagli studenti al termine del loro primo corso di programmazione; si sono chiesti in particolare se le competenze acquisite dagli studenti rispecchiassero

le aspettative e gli obiettivi iniziali di un corso di programmazione. È stato assegnato un insieme di esercizi di scrittura di programmi di varie difficoltà a studenti di diverse nazioni che avevano appena completato il corso di programmazione. Il punteggio medio degli studenti è stato approssimativamente di 23 punti su 110. Gli autori affermarono che "il primo e più significativo risultato è che gli studenti hanno avuto risultati molto inferiori alle nostre aspettative" e che la risposta alla domanda che si erano posti, ovvero se gli studenti dei corsi di programmazione di base sapessero programmare al livello di abilità atteso, è "No!". Nella loro indagine è emerso inoltre che molte delle soluzioni proposte dagli studenti presentavano addirittura problemi in fase di compilazione a causa di errori di sintassi, evidenziando che molti studenti non avevano nemmeno acquisito le abilità tecniche necessarie per far sì che un programma possa essere compilato.

Alla base degli errori e delle difficoltà incontrate dagli studenti spesso ci sono quelle che in letteratura vengono definite *misconcezioni* (traduzione dall'inglese *misconception*). Qian riprendendo [9, 46, 50] definisce le misconcezioni come "idee imperfette, spesso fortemente radicate negli studenti, che sono in conflitto con il consenso scientifico comunemente accettato". Sorva [54] utilizza il termine misconcezioni in ambito informatico con un'accezione molto ampia ad indicare "idee carenti o inadeguate per molti contesti di programmazione".

Qian [40], in una rassegna di studi precedenti, distingue le cause alla base delle misconcezioni e di altre difficoltà degli studenti:

- `Complessità del compito e carico cognitivo` - la complessità del compito può contribuire alle difficoltà degli studenti nel programmare, incrementando il loro carico cognitivo, portandoli a confondersi.

- `Linguaggio naturale` - essendo i comandi dei linguaggi di programmazione basati sul linguaggio naturale, la conoscenza degli studenti del linguaggio naturale può interferire con la loro comprensione del significato dei comandi di programmazione.

- `Conoscenze pregresse` - ad esempio, conoscenze pregresse dello studente in matematica possono portarlo a confondere assegnamenti di variabili in

un programma con espressioni algebriche, che sembrano simili ma hanno significati differenti.

- `Modelli mentali errati` - modelli mentali incompleti o imprecisi dell'esecuzione del codice e del computer possono risultare in misconcezioni. Queste idee sono solitamente le più nascoste e difficili da cambiare.

- *`Pattern`* `e strategie inadeguate` - studenti che hanno appreso la sintassi e capito i concetti possono comunque incontrare difficoltà se posseggono una conoscenza frammentaria invece di aver sviluppato *pattern* ben organizzati, strategie di programmazione, e capacità di ragionare ad un livello astratto.

- `Fattori di contesto` - fattori di contesto, come le caratteristiche del linguaggio e dell'ambiente di programmazione, possono anche essere di ostacolo al successo degli studenti nell'apprendimento della programmazione di base.

- `Istruzioni e conoscenze degli insegnanti` - qualche volte l'insegnante stesso può contribuire alla formazione di modelli mentali inaccurati negli studenti; per esempio le strategie didattiche possono includere l'utilizzo di analogie, modelli e metafore inappropriate.

Du Boulay [14] per primo ha utilizzato il termine *notional machine* ad indicare "le proprietà generali della macchina che si sta imparando a controllare". Imparare a programmare, dice Du Boulay, "è come imparare a utilizzare un set per la costruzione di un giocattolo per costruire dei meccanismi, ma come se si fosse in una stanza buia con solo pochi modi di vedere i meccanismi dell'oggetto che si sta costruendo". La *notional machine* è un modello mentale di un computer idealizzato utilizzato dal programmatore per eseguire il codice; serve allo scopo di capire cosa avviene durante l'esecuzione del programma, è associata ad uno specifico paradigma o linguaggio di programmazione, dà una prospettiva particolare sull'esecuzione dei programmi e riflette correttamente cosa fanno i programmi mentre vengono eseguiti [55]. Prendendo ad esempio l'assegnamento `a = b` in Java, alcuni principianti, a causa di un errato modello di *notional machine*, vedono l'assegnamento come un collegamento tra a e b,

così che ciò che avverrà in futuro ad *a* varrà anche per *b*. Un qualche tipo di *notional machine* è presente in ciascun programmatore anche inesperto, sia che sia esplicitata sia che non lo sia [54]. Molte misconcezioni, se non la maggior parte di esse, hanno a che fare con aspetti non facilmente visibili, ma nascosti all'interno del mondo dell'esecuzione della *notional machine*: riferimenti, oggetti, aggiornamenti automatici alle variabili di controllo dei cicli, e così via [54]. Un'altra indagine [33] riporta che i concetti più difficili lo sono in quanto gli studenti non sono in grado di comprendere cosa stia succedendo in memoria durante l'esecuzione del programma, non riuscendo a creare un chiaro modello mentale della sua esecuzione.

Associata all'idea di *notional machine* è il concetto di *superbug* introdotto da Pea [36]: il *superbug* può essere definito come "l'idea che ci sia una mente nascosta da qualche parte nel computer che ha poteri intelligenti e interpretativi. Sa cosa è successo o cosa sta per succedere su tutte le righe del programma oltre a quella che sta eseguendo; può andare oltre le informazioni che gli vengono fornite per aiutare lo studente a raggiungere i suoi obiettivi nello scrivere il programma". Questa mente nascosta, secondo Pea, fornisce una profonda spiegazione ad alcune misconcezioni che affliggono i programmatori principianti.

1.2 Modelli cognitivi della programmazione

In uno studio del 1979 [49] Shneiderman e Mayer hanno presentato un *framework* cognitivo per descrivere i comportamenti che riguardano la scrittura e la comprensione di programmi e l'acquisizione di nuove abilità di programmazione. Ogni modello che riguarda il comportamento del programmatore deve essere in grado di considerare cinque differenti compiti fondamentali (l'ordine in cui sono riportati non implica una progressione): *composizione*, scrivere un programma; *comprensione*, capire un dato problema[1]; *debugging*, trovare gli errori in un dato programma e correggerli; *modifica*, modificare un programma affinché si adatti ad un nuovo compito; *apprendimento*, acquisizione di nuove abilità e conoscenze di programmazione. In aggiunta, il modello deve poter descrivere le *strutture cognitive* che il programmatore possiede o sviluppa, e i *processi cognitivi* coinvolti

[1] Comprensione del problema non è comprensione del codice, che verrà trattata nella sezione 1.3

nell'uso della conoscenza legata alla programmazione. Il programmatore esperto deve acquisire quindi diversi tipi di conoscenze. Una parte di questa conoscenza, chiamata *conoscenza semantica*, "consiste in concetti di programmazione generali indipendenti dallo specifico linguaggio; la conoscenza semantica va da nozioni di basso livello, per esempio cosa fa la istruzione di assegnamento [...] a nozioni di più alto livello, come la strategia per la ricerca binaria". Con l'esperienza nello scrivere i programmi lo studente astrae questo tipo di conoscenza. La *conoscenza sintattica* è il secondo tipo di conoscenza che fa parte del bagaglio di un programmatore; "comprende i dettagli che riguardano aspetti come il formato dell'iterazione, istruzioni di assegnamento o condizionali, o il nome delle librerie". Una volta imparato un linguaggio di programmazione, imparare un secondo linguaggio che ne condivide almeno parzialmente la semantica sarà più facile. Al contrario, imparare un secondo linguaggio con una semantica radicalmente differente potrebbe invece essere difficile come o più che imparare il primo linguaggio.

Linn e Dalbey [24], confrontando il comportamento di programmatori esperti e non, descrivono il processo di apprendimento della programmazione come una catena di realizzazioni cognitive. La catena è composta di tre passi: (i) acquisizione della conoscenza delle caratteristiche del linguaggio; (ii) apprendimento delle abilità di progettazione, ovvero sviluppo di un repertorio di *template* e dell'abilità di combinarli per risolvere problemi; (iii) sviluppo di abilità generali di risoluzione di problemi. Il primo passo della catena corrisponde approssimativamente all'acquisizione del livello sintattico nel modello di Shneiderman e Mayer; il secondo passo copre in parte gli aspetti all'acquisizione delle conoscenze semantiche del precedente modello. Linn e Dalbey hanno però compiuto un'ulteriore distinzione tra le abilità di progettare soluzioni per problemi simili (passo ii) e l'abilità di sviluppare soluzioni a problemi nuovi e più complessi: la capacità di adattare e generalizzare le abilità di risoluzioni di problemi infatti richiede una padronanza più solida e profonda della programmazione. Nello studio da loro condotto su un campione di oltre cinquecento studenti, emerge che solo pochi studenti riescono ad andare oltre al passo (i) della catena. Viene mostrato che l'insegnamento esplicito della progettazione di programmi può essere di grande aiuto, favorendo la progressione delle conoscenze degli studenti

lungo la catena delle realizzazioni cognitive. Insegnamenti che si concentrano soprattutto su aspetti sintattici della programmazione e si basano su tecniche di progettazione semplificate, ad esempio la tecnica top-down, difficilmente riescono ad andare oltre al passo (i) della catena cognitiva. In questo caso, solamente un numero ristretto di studenti, in grado di inferire le abilità di progettazione autonomamente, avrà successo [25].

Bayman e Mayer [2] hanno sviluppato un modello simile a quello di Linn e Dalbey che descrive i tipi di conoscenza della programmazione sotto tre aspetti: sintattico, concettuale e strategico. La *conoscenza sintattica* è definita come la conoscenza della grammatica dei linguaggi; questa conoscenza è necessaria per scrivere programmi che compilano ma non è sufficiente per progettare e implementare programmi per risolvere problemi. La *conoscenza concettuale* riguarda la comprensione della semantica dei vari costrutti e delle azioni eseguite dal programma. La *conoscenza strategica* è l'abilità di utilizzare la conoscenza sintattica e concettuale nella maniera più appropriata ed efficace per risolvere nuovi problemi di programmazione; consiste nell'essere in grado di identificare le tecniche più appropriate per risolvere uno specifico problema.

Un modello che integra i tre diversi tipi di conoscenza presentati (sintattica, concettuale, e strategica) con le tre forme distinte di conoscenza (dichiarativa, procedurale, e condizionale) proposte dalla letteratura della psicologia cognitiva è stato sviluppato da McGill e Volet [32]. I loro modello si articola su cinque diversi livelli:

- `conoscenza dichiarativa-sintattica` - rappresenta la conoscenza degli aspetti sintattici legati ad un particolare linguaggio. Questa forma di conoscenza è di solito introdotta all'inizio dei corsi di programmazione, può essere presentata a lezione ed imparata sui libri. Non richiede alcuna comprensione concettuale né presuppone l'abilità di utilizzarla in un programma.

- `conoscenza dichiarativa-concettuale` - questo livello di conoscenza include la comprensione e l'abilità di spiegare la semantica delle azioni che accadono durante l'esecuzione di un programma. Questa forma differisce da entrambe le forme di conoscenza procedurale nel fatto che la sua presenza non assicura che lo studente sappia applicarla. Può esse-

re insegnata a lezione e con tutorial o derivata osservando programmi ed eseguendoli. Uno studente può possedere la comprensione concettuale di come un particolare programma funzioni senza aver la capacità di scrivere il programma da solo.

- `conoscenza procedurale-sintattica` - si riferisce all'abilità di applicare le regole della sintassi quando si programma, ovvero essere in grado di produrre istruzioni corrette in un linguaggio di programmazione. Questa conoscenza è sviluppata mediante laboratori pratici durante i quali gli studenti risolvono esercizi di programmazione.

- `conoscenza procedurale-concettuale` - abilità di utilizzare la conoscenza semantica per scrivere programmi. Questo tipo di conoscenza frequentemente non è insegnata esplicitamente, piuttosto ci si aspetta che gli studenti l'apprendano come risultato della conoscenza dichiarativa presentata a lezione. Tuttavia, negli studi in cui è stato esplicitato l'insegnamento di questa conoscenza, è stato registrato un consistente miglioramento dei risultati degli studenti [2, 35, 48, 58].

- `conoscenza strategica/condizionale` - si riferisce all'abilità di utilizzare le conoscenze sintattiche e concettuali efficacemente (sapere quando e in quali condizioni utilizzarle, da ciò "condizionale") per progettare, programmare e testare un programma che risolve un nuovo problema. Lo studente è anche in grado di spiegare la semantica delle azioni eseguite dal programma. Nuovamente, questo tipo di conoscenza solitamente non è insegnata esplicitamente.

Dati empirici di un precedente studio dello stesso Volet [58] supportano il modello proposto. Negli studenti a cui era stato proposto un metodo d'insegnamento sperimentale basato su moduli didattici basati su questa classificazione, rispetto a quelli sottoposti al metodo di insegnamento tradizionale, era significativo lo sviluppo dei due tipi di conoscenze procedurali, così come anche lo sviluppo della forma di conoscenza più ampia, quella strategica/condizionale. Le conoscenze dichiarative, come atteso, erano invece quasi invariate tra i due gruppi.

Più recentemente, Xie et al. [62], proseguendo sullo stesso filone di ricerca ha proposto una teoria che identifica quattro diverse abilità che il programmatore

principiante impara incrementalmente: (i) lettura della semantica - esaminando il codice il programmatore sa determinare gli stati intermedi e finali del programma (sa tracciare il codice); (ii) scrittura della semantica - data una descrizione non ambigua il programmatore sa tradurla in codice; (iii) lettura di *template* - il programmatore sa riconoscere un *template* ed utilizzarlo per capire cosa fa il codice; (iv) scrittura di *template* - data una descrizione di un problema il programmatore sa utilizzare il *template* al fine di risolvere il problema. Insegnando esplicitamente e progressivamente queste abilità si riduce il carico cognitivo, ottenendo un migliore tasso di completamento degli esercizi pratici e un minore tasso di errori nei programmi degli studenti.

1.3 Comprensione e scrittura del codice

L'insegnamento della programmazione è classicamente incentrato sulla scrittura di programmi per risolvere problemi. La capacità di comprendere il codice (in letteratura, *program comprehension*) è comunque un'importante abilità da acquisire, che spesso i principianti non padroneggiano adeguatamente anche alla fine del corso di programmazione [26]. Riprendendo la definizione proposta in un gruppo di lavoro ITiCSE del 2019 [22], il processo di comprensione del codice è "un processo nel quale l'individuo costruisce il proprio modello mentale del programma". In un compito di comprensione del codice "lo studente incontra un artefatto che rappresenta il programma [...] Tramite l'interazione con l'artefatto lo studente è stimolato a costruire, elaborare e raffinare il proprio modello mentale" del programma.

Un altro gruppo di lavoro ITiCSE [26], qualche anno prima, aveva studiato le prestazioni degli studenti su esercizi che non richiedevano la scrittura di codice. Erano state proposte due tipologie di esercizi: la prima testava la capacità degli studenti di predire il risultato dell'esecuzione di un piccolo pezzo di codice; la seconda verificava, dato lo scopo desiderato di un programma, l'abilità di selezionare da un piccolo insieme di possibilità il corretto completamento del programma presentato in forma parziale. Rispondere a queste domande richiedeva che lo studente avesse compreso tutti i costrutti presenti nel codice e sapesse tracciare il codice. L'idea alla base di questo studio era che se uno stu-

dente è in grado di svolgere correttamente attività di comprensione del codice ma fatica a scrivere autonomamente programmi, allora è ragionevole concludere che il problema sia una difficoltà nella risoluzione di problemi. Molti studenti non furono in grado di risolvere questo tipo di esercizi, in particolare quelli della seconda tipologia, dimostrando una scarsa padronanza di queste abilità. Gli autori indicarono queste due abilità come pre-requisiti dell'abilità di scrittura: in ogni esempio proposto a lezione o nei libri infatti, anche se l'obiettivo inteso è mostrare come scrivere programmi, sono richieste abilità di comprensione del codice.

Altre ricerche su ampia scala come il progetto BRACELet, un'indagine multinazionale multi-istituzionale sull'insegnamento e l'apprendimento della programmazione [8], hanno studiato le abilità degli studenti nel tracciare e nel leggere il codice, e la loro correlazione con il saper scrivere codice. La serie di studi facenti parte del progetto BRACELet suggerisce che:

- ciò che differenzia un programmatore esperto da un principiante è la capacità di astrazione per cui l'esperto sa riassumere il significato di un programma senza soffermarsi sulle singole righe di codice
- uno studente che ha sviluppato la capacità di tracciare è anche in grado di comprendere la relazione tra i vari blocchi di codice
- c'è una correlazione tra il sapere descrivere cosa faccia una porzione di codice e il saper scrivere codice corretto e funzionante.

Whalley et al. [60] hanno esaminato le risposte degli studenti a domande che richiedevano di spiegare in linguaggio naturale la funzione di un segmento di codice Java. Alcuni studenti sono riusciti a riassumere correttamente il significato complessivo del pezzo di codice, altri invece hanno dato una spiegazione linea per linea. Gli autori, confermando quanto emerso negli altri studi del progetto BRACELet, hanno riscontrato una correlazione tra il successo nelle domande di comprensione del codice e in altri quesiti che richiedevano di scrivere codice.

Lister et al. [28] riscontrarono che quando le stesse domande "spiega in linguaggio naturale" venivano sottoposte ad accademici, quasi sempre la risposta era un riassunto generale della funzione del blocco di codice e non una descrizione riga per riga, aspetto che mostra la capacità di riuscire a "vedere la foresta e non solamente gli alberi".

Anche in [30] è supportata la tesi che ci sia una forte correlazione tra l'abilità di tracciare e quella di scrivere codice, in particolare quando la tracciatura riguarda i cicli. La correlazione esiste, anche se in misura minore, tra la capacità di lettura del codice (domande del tipo "Spiega in linguaggio naturale...") e il saper scrivere codice.

Philpott, Robbins e Whalley [38] hanno presentato ulteriori dati che suggeriscono che tracciare il codice sia un pre-requisito per riuscire a considerare il programma nel suo complesso (aspetto a cui ci si riferisce utilizzando il termine *pensiero relazionale*): "la luce verde per il pensiero relazionale sembra essere una completa padronanza degli esercizi di tracciatura del codice [...] Invece, se l'abilità di tracciare è debole [...] la luce è decisamente rossa per quanto riguarda il pensiero relazionale".

In uno studio successivo Lister [27], benché non proponga una gerarchia stretta, afferma che una capacità iniziale di tracciare il codice viene prima della capacità di spiegare il codice e che queste due capacità insieme, almeno a livello elementare, sono necessarie per poter scrivere codice. Queste capacità, una volta sviluppate anche solo parzialmente, migliorano in parallelo rafforzandosi vicendevolmente. Buone capacità di tracciatura per esempio sono fondamentali anche quando si scrive codice e occorre trovare possibili errori o controllare che il programma faccia ciò che ci si aspetta.

Similmente in [10] si sostiene che "un bilanciato mix di scrittura e lettura del codice è il modo più efficace per gli studenti per sviluppare buone abilità di programmazione".

Questi studi sottolineano tutti il fatto che ci sia una stretta correlazione tra la capacità di lettura del codice e quelle di risoluzione di problemi, inclusa la scrittura di codice. In seguito a queste considerazioni, numerosi studi hanno esaminato come aiutare gli studenti a sviluppare capacità di comprensione del codice e molti hanno valutato gli effetti di attività progettate appositamente per stimolare le capacità di tracciatura e lettura sulle capacità di scrittura degli studenti.

Schulte ha sviluppato un framework per aiutare gli insegnanti a valutare il carico cognitivo degli esercizi di comprensione del codice, il Block Model [47]. Nell'idea di Schulte il framework doveva servire da modello per pianificare e

MACROSTRUTTURA	Comprensione della struttura complessiva del testo del programma	Comprensione dell'*algoritmo* del programma	Comprensione del goal/scopo del programma (nel suo contesto)
RELAZIONI	Riferimenti tra blocchi (ad es. chiamate di metodi, creazione di oggetti, accesso ai dati...)	Sequenza di chiamate a metodi	Comprensione di come i sotto-goal sono in relazione con i goal, come la funzione è raggiunta dalle sotto-funzioni
BLOCCHI	*Regioni di interesse* che sintatticamente o semanticamente costituiscono un'unità	Esecuzione di un blocco, un metodo, o di una regione di interesse	Funzione di un blocco (eventualmente visto come sotto-goal)
ATOMI	Elementi del linguaggio	Esecuzione di un'istruzione	Funzione di un'istruzione
	CODICE STATICO	ESECUZIONE	FUNZIONE

Figura 1.2: Matrice del Block Model

analizzare i contenuti delle lezioni e degli esercizi, in modo semplice e facilmente adattabile ad ogni contesto. Il Block Model considera il programma su due diverse dimensioni. La prima dimensione riguarda la granularità con cui viene esaminato il codice: è possibile considerare la singola riga (*atomo*), un *blocco* di codice, la *relazione* tra due o più blocchi di codice, o l'intero programma (*macro-struttura*). La seconda dimensione individua tre diversi modi di guardare al codice: in maniera statica, come codice (*codice statico*); in maniera dinamica, quando lo si esegue (*esecuzione*); in relazione al suo scopo/funzione (*funzione*). La matrice risultante è presentata in Figura 1.2.

In [45] il Block Model è stato utilizzato per classificare, a seconda del loro contenuto cognitivo, oltre seicento domande a scelta multipla di programmazione di base e avanzata. In [22] è presentata e classificata secondo il Block Model un'ampia collezione di attività di apprendimento, con oltre 60 tipologie differenti di esercizi mirati esplicitamente a stimolare l'abilità di comprensione del codice. Gli autori evidenziano che l'effetto più sorprendente dell'approccio tramite Block Model sia "un cambiamento di prospettiva sui compiti di apprendimento basato su una semplice domanda: cosa vuol dire se un principiante non è in grado di risolvere un esercizio di comprensione del codice?". L'analisi degli esercizi tramite Block Model permette di comprendere meglio quali siano le difficoltà insite in ogni domanda e di correlarle con eventuali lacune e difficoltà dello studente.

1.4 *Goal* e *plan*

Come già evidenziato nella sezione 1.1, una delle principali difficoltà dei principianti sta nel *mettere i pezzi insieme*, comporre e coordinare la varie componenti di un programma [53]. Soloway [52] ha suggerito di analizzare i problemi da risolvere in termini di *goal*, cioè compiti da svolgere, e di *plan*, ovvero "soluzioni tipiche pronte all'uso" (dall'espressione originale inglese *stereotipical canned solution*) con cui vengono raggiunti i *goal*. Nel resto della trattazione si è scelto di mantenere i termini *goal* e *plan* proposti da Soloway, senza tradurli. Altri autori hanno utilizzato differenti termini per presentare concetti analoghi, come *schema* [41], *pattern* [15, 39, 34], o *strategie* [11, 12, 13]. Soloway come esempio riprende il Rainfall Problem presentato nel paragrafo 1.1. Possiamo identificare cinque *goal*: (i) lettura dei numeri in input; (ii) calcolo della somma dei numeri passati in input (escluso il valore di terminazione); (iii) conteggio dei numeri letti (escluso il valore di terminazione); (iv) divisione tra la somma e il conteggio dei numeri letti, controllando che il divisore non sia zero; (v) stampa dell'output. Essendo *goal* comuni, un programmatore esperto conoscerà sicuramente soluzioni standard pronte all'uso. L'implementazione più tipica (Figura 1.3) presenterà un ciclo con sentinella con una variabile accumulatore per sommare tutti i valori; nella terminologia di Soloway questo prende il nome di *plan per il goal della somma* (*running total loop plan*). Il terzo *goal* richiede un *plan iterativo di conteggio* (*counter loop plan*) per contare quanti numeri sono sommati, mentre per il calcolo della media viene implementato un *plan della divisione protetta* (*guarded division plan*), in modo da evitare eventuali divisioni per zero.

Nell'implementazione della soluzione i vari *plan* sono combinati tra loro. Soloway individua tre modalità di combinazione dei *plan*: concatenazione (*abutment*), in cui i *plan* sono sviluppati uno dopo l'altro (per esempio nell'esempio precedente, i *plan* per la somma e il conteggio prima e quello per la divisione protetta dopo); annidamento (*nesting*), dove un *plan* è contenuto in un altro *plan* (il *plan* per stampare l'output dentro al *plan* per la divisione protetta per il calcolo della media); fusione (*merging*), in cui i *plan* vengono fusi insieme (*plan* per la somma e *plan* per il conteggio nello stesso ciclo). Spesso i *plan* inoltre necessitano di essere adattati (*tailoring*) allo specifico *goal* a cui devono rispon-

```
inizializza una variabile accumulatore
inizializza un contatore
chiedi all'utente di inserire un valore
se l'input non è il valore sentinella allora
    aggiungi il nuovo valore alla variabile accumulatore
    incrementa il contatore
    torna indietro all'inserimento dell'input
se il contatore è maggiore di 0 allora
    divido la variabile accumulatore per il contatore
    stampa l'output
altrimenti riporta all'utente l'errore
```

Figura 1.3: Implementazione tipica del Rainfall problem

dere. Soloway, citando il pensiero di Benjamin Whorf secondo cui "il linguaggio determina il pensiero", sottolinea l'importanza di fornire agli studenti un linguaggio appropriato; il lessico dei *goal* e *plan* assolve anche a questo compito, dando una terminologia sia per sviluppare che per analizzare i programmi.

Questo approccio con *goal* e *plan* è stato usato da De Raadt nell'ambito della valutazione degli studenti [11]. Ai partecipanti di un corso di programmazione di base è stato chiesto di risolvere il Rainfall Problem, in modo anonimo e senza poter utilizzare il computer. In ogni consegna veniva poi valutata la presenza/assenza dei vari *plan*. Nonostante l'esercizio fosse considerato basilare dallo stesso De Raadt, i risultati mostrarono importanti lacune nelle strategie di programmazione degli studenti. Solamente una soluzione includeva tutti i *plan* richiesti; se si esclude il *plan* per la divisione protetta, il 23% delle soluzioni erano complete. Considerando come promossi gli studenti che avevano implementato almeno la metà dei *plan*, il 62% degli studenti risultava promosso.

Lo stesso autore, in un successivo studio ha valutato i risultati dell'insegnamento esplicito delle strategie di programmazione [13]. I risultati mostrano evidenze che: (i) sia possibile insegnare esplicitamente queste strategie; (ii) esplicitare le strategie nell'insegnamento contribuisce a formare un linguaggio utile sia nell'analisi dei programmi che nell'insegnamento; (iii) è possibile includere

queste strategie tra gli aspetti valutati negli esami.

Ginat ha proposto agli studenti un esame volto a testare la loro capacità di utilizzo dei *plan* (nella terminologia di Ginat prendono il nome di *pattern*) [18]. La metà degli esercizi richiedeva di combinare due di questi *pattern* (per esempio, era richiesto di trovare il numero di occorrenze del minimo in una sequenza). Agli studenti era richiesto di rispondere ad ogni domanda nel modo più efficiente possibile. Solo il 37% delle risposte erano corrette. Ginat classificò gli errori più frequenti in tre diverse categorie: soluzioni in cui i singoli *pattern* erano errati; soluzioni che presentavano una concatenazione di *pattern* invece della fusione; soluzioni nelle quali la composizione tra *pattern* aveva condotto a output errati. Gran parte degli errori erano dovuti alla difficoltà di combinare i pattern correttamente. Ginat afferma che ciò è un effetto della tecnica di progettazione top-down proposta classicamente nei libri e a lezione: i pattern invece "non devono essere percepiti come unità atomiche ma piuttosto come unità formate da componenti più piccole, che possono essere manipolate in vario modo".

Così come i *plan* descrivono soluzioni stereotipiche per la risoluzione di *goal*, analogamente è possibile definire degli usi tipici delle variabili. Sajaniemi [43]ha analizzato i programmi presentati in tre libri di testo di programmazione di base in Pascal, individuando otto ruoli che coprivano il 99% delle variabili incontrate. La classificazione risultante è riportata di seguito.

- `Fixed-value` - variabile il cui valore non cambia durante l'esecuzione del programma
- `Stepper` - variabile che cambia valore in una qualche maniera sistematica
- `Follower` - variabile che assume il vecchio valore di un'altra variabile
- `Most-recent-holder` - variabile che assume l'ultimo valore incontrato in una sequenza di valori
- `Most-wanted-holder` - variabile che assume il miglior valore, secondo un qualche criterio, incontrato in una sequenza di valori
- `Gatherer` - variabile che accumula in qualche senso i valori incontrati in una sequenza di valori (somma, prodotto, concatenazione...)

- `One-way-flag` - variabile booleana che può cambiare valore solo una volta, al verificarsi di una determinata condizione
- `Temporary` - variabile che assume il valore di una qualche altra variabile per un breve tempo

L'84% dei casi è coperto dalle tre variabili più frequenti (costanti, stepper, most-recent-holder). La definizione dei ruoli è basata "sulla natura dei valori successivi che assumono le variabili, senza prestare attenzione al modo in cui sono poi utilizzate". Riprendendo l'esempio del Rainfall Problem, verrà istanziata una variabile di tipo *stepper* per contare i numeri inseriti e un *gatherer* per l'accumulo della somma.

Sajaniemi e Kuittinen valutarono l'effetto dell'insegnamento esplicito dei ruoli delle variabili [44]. Gli studenti di un corso di programmazione di base in Pascal vennero divisi in due gruppi: il primo gruppo partecipò a lezioni tradizionali mentre il secondo utilizzò i ruoli delle variabili per tutta la durata del corso. In seguito venne chiesto agli studenti di rispondere a tre domande che riguardavano rispettivamente la tracciatura, la descrizione della funzione di un programma, e la scrittura di codice. I risultati mostrarono che nelle ultime due tipologie di esercizi, gli studenti che avevano utilizzato i ruoli delle variabili durante il corso (secondo gruppo) ottennero risultati nettamente migliori di quelli del primo gruppo; sugli esercizi di tracciatura i risultati furono simili.

1.5 Valutazione degli apprendimenti

La capacità di saper programmare è valutata in molte maniere diverse. Gli esami tipici includono una prova scritta, una prova orale, progetti/compiti di laboratorio individuali o di gruppo, esami pratici al computer. La scrittura di programmi è uno dei più comuni tipi di compito che ritroviamo negli esami di programmazione di base [37]. I programmi scritti dagli studenti sono generalmente valutati considerando sia gli aspetti funzionali (ovvero se l'output del programma corrisponde a quello atteso su una serie di test significativi) sia la qualità del codice (per esempio la leggibilità del codice, la modularità, l'uso idiomatico dei costrutti, le scelte dei nomi, concisione e consistenza e così via) [56, 21, 20, 31]). Questi aspetti sono importanti non soltanto nel proces-

so di valutazione e di assegnazione dei voti, ma anche per dare *feedback* agli studenti, specialmente ai principianti.

In letteratura sono stati descritti tre approcci per la valutazione dei programmi [16]: nella valutazione *olistica* il valutatore si fa un'impressione generale della qualità del programma e dà un voto complessivo; nella valutazione *analitica* viene definito un insieme di criteri ampio e dettagliato, i punteggi relativi ai vari criteri vengono assegnati indipendentemente l'uno dall'altro e sono poi sommati per ottenere il voto finale; nella valutazione per *tratti primari* sono considerati solo un numero molto piccolo di aspetti mentre gli aspetti secondari vengono ignorati.

In generale, la valutazione è basata sulla definizione di alcuni criteri, di una scala per ciascun criterio, e di pesi o priorità per combinarli. Al fine di favorire un'equità di valutazione, la scala dell'università di Jacksonville [20] elenca sette criteri e fornisce una scala pesata per assegnare punti a questi criteri. McCraken et al. [31] considerano nella valutazione sia l'esecuzione dei programmi e la correttezza dell'output che lo stile di scrittura del codice, utilizzando un indicatore che misura quanto il codice della soluzione sia vicino ad una soluzione corretta. Una rubrica per valutare specificamente la qualità del codice è proposta in [56].

In [16] sono messi a confronto metodi diversi di valutazione di quattro insegnanti di programmazione, due basati su approcci analitici e due su approcci olistici, e viene rilevato che, nonostante i diversi metodi di valutazione, è presente una forte correlazione tra i voti assegnati dai quattro istruttori. In particolare l'accordo è pressoché totale sulle valutazioni molto alte e sulle valutazioni molto basse. Un altro recente studio [1] mostra al contrario quanto gli istruttori possano essere in disaccordo nel valutare domande di scrittura di codice negli esami di programmazione di base.

La tassonomia SOLO (*Structure of Observed Learning Outcome*), formulata da Biggs e Collis nel 1982, fornisce un metodo per analizzare e valutare le risposte degli studenti in ambito multi-disciplinare [5]. La tassonomia è formata da cinque categorie, che riflettono il livello di complessità e qualità della risposta: (i) *pre-strutturale* - lo studente non ha capito l'esercizio o la risposta non ha alcuna rilevanza; (ii) *mono-strutturale* - lo studente si concentra su un solo aspetto o dimostra di aver capito un solo aspetto; (iii) *multi-strutturale* - lo stu-

dente descrive tutti gli aspetti ma senza riuscire a metterli in relazione l'uno con l'altro, dimostrando una conoscenza nozionistica; (iv) *relazionale* - lo studente riesce a mettere in relazione i vari aspetti e applicare il concetto a problemi già visti; (v) *astratta* - lo studente riesce ad astrarre e ad applicare i concetti a problemi mai visti. Thompson [57] ha applicato la tassonomia SOLO alla programmazione, utilizzandola nella valutazione degli esami. In particolare ha definito una serie di criteri per ogni categoria SOLO e li ha utilizzati per la valutazione, al fine di aiutare gli studenti a comprendere meglio il voto assegnato. Ginat ha utilizzato la tassonomia SOLO per valutare le abilità di progettazione degli studenti [17]. Studi simili sono stati compiuti anche all'interno del progetto BRACELet [61, 29, 38].

Bibliografia

[1] Ibrahim Albluwi. A closer look at the differences between graders in introductory computer science exams. *IEEE Transactions on Education*, 61(3):253–260, 2018.

[2] Piraye Bayman and Richard E Mayer. Using conceptual models to teach basic computer programming. *Journal of Educational Psychology*, 80(3):291, 1988.

[3] Jens Bennedsen and Michael E. Caspersen. Failure rates in introductory programming. *SIGCSE Bull.*, 39(2):32–36, June 2007.

[4] Jens Bennedsen and Michael E Caspersen. Failure rates in introductory programming: 12 years later. *ACM Inroads*, 10(2):30–36, 2019.

[5] J Biggs and K Collis. Evaluating the quality of learning: the solo taxonomy new york: Academic pres. 1982.

[6] Jeffrey Bonar and Elliot Soloway. Uncovering principles of novice programming. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 10–13, 1983.

[7] Richard Bornat, Saeed Dehnadi, et al. Mental models, consistency and programming aptitude. In *Proceedings of the tenth conference on Australasian computing education-Volume 78*, pages 53–61. Australian Computer Society, Inc., 2008.

[8] Tony Clear, J.L. Whalley, Phil Robbins, Anne Philpott, Anna Eckerdal, and M. Laakso. Report on the final bracelet workshop: Auckland uni-

versity of technology, september 2010. *Journal of Applied Computing and Information Technology*, 15(1), 2011.

[9] John Clement. Using bridging analogies and anchoring intuitions to deal with students' preconceptions in physics. *Journal of research in science teaching*, 30(10):1241–1257, 1993.

[10] Malcolm Corney, Sue Fitzgerald, Brian Hanks, Raymond Lister, Renee McCauley, and Laurie Murphy. "explain in plain english" questions revisited: Data structures problems. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, page 591–596, New York, NY, USA, 2014. Association for Computing Machinery.

[11] Michael de Raadt, Mark Toleman, and Richard Watson. Training strategic problem solvers. *ACM SIGCSE Bulletin*, 36(2):48, oct 2004.

[12] Michael de Raadt, Richard Watson, and Mark Toleman. Chick sexing and novice programmers: Explicit instruction of problem solving strategies. *Conferences in Research and Practice in Information Technology Series*, 52:55–62, 2006.

[13] Michael de Raadt, Richard Watson, and Mark Toleman. Teaching and assessing programming strategies explicitly. *Conferences in Research and Practice in Information Technology Series*, 95:45–54, 2009.

[14] Benedict du Boulay. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2(1):57–73, may 1986.

[15] J Philip East, S Rebecca Thomas, Eugene Wallingford, Walter Beck, and Janet Drake. Pattern-based Programming Instruction *. In *ASEE Annual Conference Proceedings*, 1996.

[16] Sue Fitzgerald, Brian Hanks, Raymond Lister, Renee McCauley, and Laurie Murphy. What are we thinking when we grade programs? In *SIGCSE 2013*, page 471. ACM, 2013.

[17] David Ginat and Eti Menashe. Solo taxonomy for assessing novices' algorithmic design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 452–457, 2015.

[18] David Ginat, Eti Menashe, and Amal Taya. Novice difficulties with interleaved pattern composition. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7780 LNCS, pages 57–67, 2013.

[19] Mark Guzdial. From science to engineering. *Communications of the ACM*, 54(2):37–39, 2011.

[20] R. Wayne Hamm, Kenneth D. Henderson, Marilyn L. Repsher, and Kathleen M. Timmer. A tool for program grading: The Jacksonville University Scale. *ACM SIGCSE Bulletin*, 15(1):248–252, 1983.

[21] James W. Howatt. On Criteria for Grading Student Programs. *ACM SIGCSE Bulletin*, 26(3):3–7, 1994.

[22] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, et al. Fostering program comprehension in novice programmers-learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, pages 27–52. 2019.

[23] Tony Jenkins. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, volume 4, pages 53–58. Citeseer, 2002.

[24] Marcia Linn and John Dalbey. Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational Psychologist - EDUC PSYCHOL*, 20:191–206, 09 1985.

[25] Marcia C Linn and Michael J Clancy. The case for case studies of programming problems. *Communications of the ACM*, 35(3):121–132, 1992.

[26] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda 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, New York, NY, USA, 2004. ACM.

[27] Raymond Lister, Colin Fidge, and Donna Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Acm sigcse bulletin*, 41(3):161–165, 2009.

[28] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L Whalley, and Christine Prasad. Not seeing the forest for the trees: novice programmers and the solo taxonomy. *ACM SIGCSE Bulletin*, 38(3):118–122, 2006.

[29] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L Whalley, and Christine Prasad. Not seeing the forest for the trees: novice programmers and the solo taxonomy. *ACM SIGCSE Bulletin*, 38(3):118–122, 2006.

[30] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*, pages 101–112, 2008.

[31] Michael McCracken, Tadeusz Wilusz, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, and Ian Utting. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, 33(4):125, dec 2001.

[32] Tanya J. McGill and Simone E. Volet. A conceptual framework for analyzing students' knowledge of programming. *Journal of Research on Computing in Education*, 1997.

[33] Iain Milne and Glenn Rowe. Difficulties in learning and teaching programming—views of students and tutors. *Education and Information technologies*, 7(1):55–66, 2002.

[34] Orna Muller. Pattern oriented instruction and the enhancement of analogical reasoning. *ICER 2005*, pages 57–67, 2005.

[35] Ron Oliver and John Malone. The influence of instruction and activity on the development of semantic programming knowledge. *Journal of Research on Computing in Education*, 25(4):521–533, 1993.

[36] Roy D Pea. Language-independent conceptual "bugs" in novice programming. *Journal of educational computing research*, 2(1):25–36, 1986.

[37] Andrew Petersen, Michelle Craig, and Daniel Zingaro. Reviewing CS1 exam question content. In *SIGCSE 2011*, page 631. Association for Computing Machinery (ACM), mar 2011.

[38] Anne Philpott, Phil Robbins, and J Whalley. Assessing the steps on the road to relational thinking. In *Proceedings of the 20th Annual Conference of the National Advisory Committee on Computing Qualifications*, volume 286, 2007.

[39] Viera K. Proulx. Programming patterns and design patterns in the introductory computer science course. *ACM SIGCSE Bulletin*, 32(1):80–84, jul 2004.

[40] Yizhou Qian and James Lehman. Misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Educ.*, 18(1):1:1–1:24, October 2017.

[41] Robert S. Rist. Schema creation in programming. *Cognitive Science*, 1989.

[42] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer science education*, 13(2):137–172, 2003.

[43] Jorma Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 37–39. IEEE, 2002.

[44] Jorma Sajaniemi and Marja Kuittinen. An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, 15(1):59–82, 2005.

[45] Kate Sanders, Marzieh Ahmadzadeh, Tony Clear, Stephen H Edwards, Mikey Goldweber, Chris Johnson, Raymond Lister, Robert McCartney, Elizabeth Patitsas, and Jaime Spacco. The canterbury questionbank: building a repository of multiple-choice cs1 and cs2 questions. In *Proceedings of the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports*, pages 33–52, 2013.

[46] Michael J Sanger and Thomas J Greenbowe. Common student misconceptions in electrochemistry: Galvanic, electrolytic, and concentration cells. *Journal of Research in Science Teaching: The Official Journal of the National Association for Research in Science Teaching*, 34(4):377–398, 1997.

[47] Carsten Schulte. Block Model: An Educational Model of Program Comprehension As a Tool for a Scholarly Approach to Teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pages 149–160, New York, NY, USA, 2008. ACM.

[48] Steven Schwartz, DN Perkins, Greg Estey, John Kruidenier, and Rebecca Simmons. A "metacourse" for basic: Assessing a new model for enhancing instruction. *Journal of Educational Computing Research*, 5(3):263–297, 1989.

[49] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3):219–238, 1979.

[50] John P Smith III, Andrea A Disessa, and Jeremy Roschelle. Misconceptions reconceived: A constructivist analysis of knowledge in transition. *The journal of the learning sciences*, 3(2):115–163, 1994.

[51] E Soloway, K Ehrlich, and J Bonar. Greenspan, j.(1982). what do novices know about programming. *Directions in Human. computer Interaction. Norwood, NJ" Ablex Publishing Corporation*, 1982.

[52] Elliot Soloway. Learning to Program = Learning to Construct Mechanisms and Explanations. *Communication of the ACM*, 29(9):850–858, 1986.

[53] Elliot Soloway and James C Spohrer. Studying the novice programmer. hillside, 1989.

[54] Juha Sorva. Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13(2):1–31, jul 2013.

[55] Juha Sorva et al. *Visual program simulation in introductory programming education*. Aalto University, 2012.

[56] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. Towards an empirically validated model for assessment of code quality. pages 99–108, 2014.

[57] Errol Thompson. Holistic assessment criteria: applying solo to programming projects. In *Proceedings of the ninth Australasian conference on Computing education-Volume 66*, pages 155–162. Australian Computer Society, Inc., 2007.

[58] Simone E Volet. Modelling and coaching of relevant metacognitive strategies for enhancing university students' learning. *Learning and Instruction*, 1(4):319–336, 1991.

[59] Christopher Watson and Frederick WB Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 39–44, 2014.

[60] Jacqueline L Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, PK Ajith Kumar, and Christine Prasad. An australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies. In *Conferences in Research and Practice in Information Technology Series*, 2006.

[61] Jacqueline L Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, PK Ajith Kumar, and Christine Prasad. An australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies. In *Conferences in Research and Practice in Information Technology Series*, 2006.

[62] Benjamin Xie, Dastyni Loksa, Greg Nelson, Matthew Davidson, Dongsheng Dong, Harrison Kwik, Alex Tan, Leanne Hwa, Min Li, and Amy Ko. A theory of instruction for introductory programming skills. *Computer Science Education*, pages 1–49, 01 2019.