[ITADINFO]

CONVEGNO ITALIANO
SULLA DIDATTICA DELL'INFORMATICA

Dalle foglie alle radici: imparare il *Debugging* dalle sue componenti fondamentali

Gabriele Pozzan - gabriele.pozzan@phd.unipd.it
Tullio Vardanega - tullio.vardanega@unipd.it

4 ottobre 2025, Salerno

Introduzione

I materiali di questo laboratorio (dispensa formato PDF, esercizi, ecc.) sono disponibili all'indirizzo: https://github.com/cornacchia/debugging-workshop-itadinfo-2025

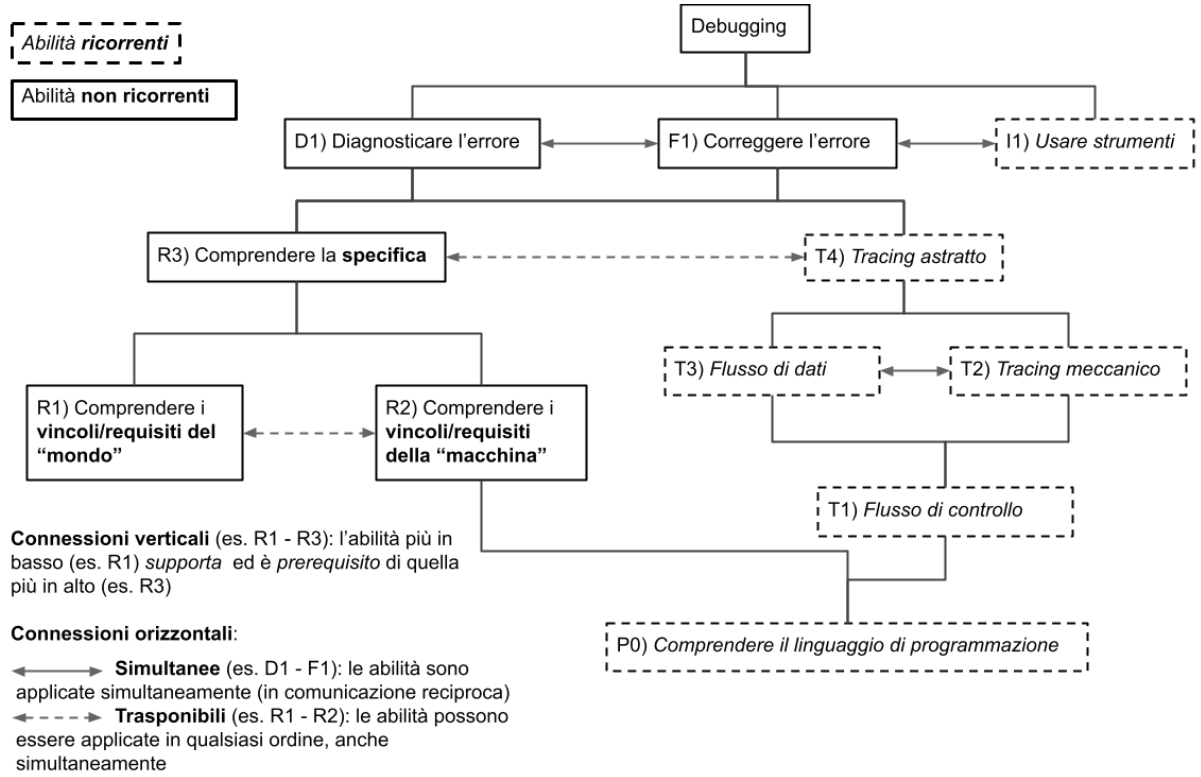

Fig. 1 - Scomposizione del *Debugging* in sottoabilità collegate da relazioni di supporto (le abilità in basso sono supporto e prerequisito di quelle in alto), basata sulla notazione spiegata da: Van Merriënboer, J. J., Clark, R. E., & De Croock, M. B. (2002). **Blueprints for complex learning: The 4C/ID-model.** *Educational technology research and development*, *50*(2), 39-61.

Per *Debugging* intendiamo qui sia il processo (sequenza di azioni necessarie a fare una cosa), sia la capacità/abilità di diagnosticare e correggere errori di programmazione (i cosiddetti *bug*). Questa abilità è una componente fondamentale del pensiero Informatico e la ricerca scientifica invita spesso al suo insegnamento diretto e sistematico.

La scomposizione del *Debugging* in sottoabilità illustrata in Fig. 1 è fondata sulla teoria del **carico cognitivo**, formulata principalmente da John Sweller e Jeroen J. G. van Merriënboer. Quest'ultimo, in particolare, ha stabilito la notazione alla base della nostra scomposizione e la divisione fondamentale delle sottoabilità in ricorrenti/non ricorrenti, di grande utilità nel progettare attività didattiche. Per approfondire, vedere: Sweller, J., Van Merriënboer, J. J., & Paas, F. (2019). **Cognitive architecture and instructional design: 20 years later.** *Educational psychology review*, *31*(2), 261-292.

Dal punto di vista del carico cognitivo, il *Debugging* è un'abilità complessa che richiede il coordinamento di diverse sottoabilità. Chi fa *Debugging* deve essere in grado di 1) diagnosticare l'errore e 2) applicare una soluzione. Per poter diagnosticare un errore (sia logico che di implementazione) è necessario comprendere quale sia il programma **atteso**, ovvero la composizione di algoritmi che idealmente soddisfa i requisiti, e quale sia il programma **concreto**, ovvero l'effettivo codice eseguito dalla macchina (e l'effetto della sua esecuzione). Bisogna inoltre comprendere pienamente i **requisiti**, ovvero le aspettative che si hanno sul programma e su quello che può/deve "fare".
Risolvere un errore di tipo logico (cioè un errore di ragionamento nel "pensare" a livello di algoritmo la soluzione ai requisiti) significa far convergere il programma atteso verso i requisiti. Risolvere un errore di implementazione (cioè un errore nel tradurre algoritmi in linguaggi di programmazione eseguibili) significa far convergere il programma concreto verso il programma atteso.
La comprensione del programma concreto richiede capacità di lettura, comprensione, analisi del codice (chiamata *Tracing*). La comprensione del programma atteso richiede la comprensione dei requisiti del "mondo" (degli utenti, del problema da risolvere), dei requisiti della "macchina" (limiti e contesto di esecuzione) e della specifica, che combina i primi due creando una mappa che collega aspetti della macchina ad aspetti del mondo.

Come leggere il grafico

Il grafico rappresentato in Fig. 1 mostra una possibile scomposizione di tutte le abilità che vengono messe in campo quando si "fa *debugging*". È importante tenere a mente che *tutte* queste abilità vengono applicate durante un processo di *debugging*, in misura maggiore o minore a seconda del tipo di errore da risolvere, della complessità del programma, ecc.

Le connessioni verticali del grafico rappresentano *prerequisiti*: le abilità che stanno più in basso sono prerequisito di quelle che stanno più in alto. Per esempio: sapere come un certo linguaggio di programmazione implementa il concetto di iterazione (P0) è prerequisito a capire come il relativo costrutto (per esempio un ciclo *while* o un ciclo *for*) influenza il flusso di controllo (cioè l'ordine in cui le operazioni del programma sono eseguite, rappresentato da T1). Quando più abilità "in basso" sono collegate a una stessa abilità "in alto", quest'ultima deve *coordinare* quelle più in basso: per esempio, comprendere un programma a livello di *tracing* astratto (cioè comprenderlo a partire da vincoli e relazioni logiche, T4) richiede di essere in grado di comprenderlo a livello meccanico (cioè capire esattamente l'effetto delle sue istruzioni a partire da una situazione iniziale "fissa", T2) e di capirne il flusso di dati (ovvero, come le sue diverse parti si influenzano a vicenda, T3).

Le connessioni orizzontali suggeriscono come le abilità con ruolo di prerequisito vengono coordinate dalle abilità cui sono collegate verticalmente:

- Le abilità in connessione **simultanea** devono essere applicate allo stesso tempo ed essere in "dialogo" reciproco per supportare l'abilità più in alto.
- Le abilità in connessione **trasponibile** possono essere applicate in qualunque ordine e in momenti diversi.

Le abilità sono infine categorizzate per tipologia:

- Le abilità **ricorrenti** sono *consistenti* a prescindere dal contesto specifico di applicazione. Sono frutto di un pensiero veloce, inflessibile, automatico. Per esempio, abilità ricorrenti sono risolvere semplici operazioni di aritmetica (addizioni, sottrazioni, ecc. con numeri "piccoli") e riconoscere il significato di vocaboli di una lingua. Queste abilità richiedono *molto* esercizio, per lunghi periodi, per essere automatizzate in modo efficace. Gli esercizi che meglio le supportano sono "piccoli" e forniscono un riscontro immediato su eventuali errori. Per esempio, per imparare vocaboli di una lingua straniera è efficace un sistema automatico che li riproponga a intervalli basati sulla capacità e facilità nel riconoscerli correttamente (vedere per esempio il software Anki: https://apps.ankiweb.net/).
- Le abilità **non ricorrenti** sono *variabili* e dipendono dal contesto di applicazione. Sono frutto di un pensiero lento, flessibile, creativo. Per esempio, abilità ricorrenti sono l'analisi di equazioni e la scrittura di testi. Queste abilità richiedono soprattutto *varietà* e *spazio di riflessione (metacognizione)*. L'obiettivo non è automatizzare ma *generalizzare* queste competenze al di là di contesti specifici di applicazione. Per esempio, per allenare la scrittura creativa può essere efficace scrivere/leggere racconti secondo i canoni di diversi "generi" (horror, fantasy, avventura) e poi analizzarli e confrontarli per individuare affinità e divergenze.

Dettagli sulle abilità

Le fondamenta (P0)

L'abilità fondamentale che è un prerequisito di (quasi) tutte le altre è **comprendere il linguaggio di programmazione (P0)**. Questa comprensione include la *sintassi* (regole di scrittura di un linguaggio), la *semantica* (significato delle istruzioni di un linguaggio) e la *pragmatica* (effetto dell'esecuzione delle istruzioni). Questa abilità non implica che si debba conoscere la totalità di un linguaggio di programmazione prima di poter affrontare le successive. L'approccio suggerito per "scalare" il modello è quello cosiddetto *a spirale*: dopo aver affrontato alcuni costrutti a livello di un'abilità (per esempio P0), si può passare ad affrontarli a livello di una successiva (per esempio T1) e così via. Quando si vorranno introdurre nuovi costrutti e concetti si potrà ripartire da P0 e "scalare" nuovamente il modello.

Esercizi suggeriti

Nota: Essendo questa una abilità **ricorrente**, gli esercizi più indicati per allenarla dovrebbero essere molti, "piccoli" e fornire riscontro immediato su eventuali errori. Per esempio, si potrebbero regolarmente tenere delle sessioni di una decina di minuti con domande "botta e risposta", stile quiz. Queste indicazioni valgono anche per le altre abilità ricorrenti.

- Per la **sintassi**: individuare ed indicare errori di sintassi in brevi programmi; tradurre uno schema di programma (es. un diagramma di flusso) in codice eseguibile.
- Per la **semantica**: dividere i costrutti di un linguaggio in categorie (quelli che "stampano" un valore, quelli che alterano il flusso di controllo, ecc.); individuare ed

indicare usi impropri degli operatori (per esempio, in certi linguaggi, la “somma” tra una stringa e un intero).

- Per la **pragmatica**: collegare operatori a descrizioni degli effetti della loro esecuzione; dato uno stato iniziale (poche variabili inizializzate con qualche valore) e una istruzione, prevedere come questa modificherà lo stato (nota: all’aumentare delle istruzioni questo esercizio si sposta verso il *tracing*).

Il programma atteso (R1, R2, R3)

Il programma **atteso** è la composizione di istruzioni, algoritmi e moduli che, idealmente, soddisfa i requisiti, ovvero il “problema” che il programma cerca di risolvere. Il programma atteso è diverso dal programma effettivo (ovvero il codice effettivamente eseguibile) perché uno stesso algoritmo può essere implementato da diversi linguaggi di programmazione (ed esiste quindi a un livello più astratto). Un errore nel programma atteso è un errore **logico**, l’obiettivo del *debugging* in questo caso è correggere un passo falso nella soluzione algoritmica al “problema” affrontato dal programma. Un errore nella traduzione da programma atteso a programma effettivo è invece un errore di **implementazione**, l’obiettivo del *debugging* in questo caso è far convergere il programma effettivo verso il programma atteso.

L’abilità **comprendere i vincoli/requisiti del “mondo” (R1)** rappresenta la capacità di capire il “problema” che il programma deve risolvere. Questo “problema” può essere il testo di un esercizio (es. “Scrivi un programma che calcoli la media di tutti gli interi da 1 a 100”) o una serie di requisiti che il programma deve soddisfare. È fondamentale verificare che chi deve fare *debugging* di un programma capisca il **dominio** del “problema” che il programma affronta.

Esercizi suggeriti

Nota: essendo questa abilità **non ricorrente**, gli esercizi più indicati per allenarla sono quelli in cui coordinare varie competenze, che possono essere anche attività lunghe, svolte in dialogo con chi insegna. È importante la *varietà* degli esercizi (con l’obiettivo di generalizzare le competenze) e la *riflessione* durante e dopo le attività (per consolidare le competenze). Eventuali materiali di supporto alle attività (es. spiegazione di concetti fondamentali o del dominio del problema) vanno condotte *prima* delle esercitazioni, in modo da non creare sovraccarico cognitivo. Queste indicazioni valgono anche per le altre abilità non ricorrenti.

- Dopo aver letto il testo del “problema” che un programma deve risolvere (es. “Scrivi un programma che, dati due numeri interi, calcoli la loro media”), dati uno o più valori in ingresso (es. coppie di numeri interi), scrivere il risultato atteso dal programma (es. la media delle coppie di interi). Questo esercizio verifica la comprensione del dominio del “problema” e di quanto è richiesto al programma. Per una discussione più approfondita dell’esercizio vedere: Prather, J., Pettit, R., Becker, B. A., Denny, P., Loksa, D., Peters, A., ... & Masci, K. (2019, February). **First things first: Providing metacognitive scaffolding for interpreting problem prompts.** In *Proceedings of the 50th ACM technical symposium on computer science education* (pp. 531-537).

- Descrivere una “macchina” incontrata nella vita quotidiana (es. una macchinetta del caffè) a partire dai suoi requisiti (es. la macchinetta deve poter accettare pagamenti, le utenti devono poter selezionare un prodotto, ecc.).
- Discutere diversi domini di “problemi” risolti da software (es. che differenza c’è tra il software di una applicazione “sveglia” per il telefono e quello che gestisce processi critici in una centrale nucleare?)

L’abilità **comprendere i vincoli/requisiti della “macchina” (R2)** rappresenta la conoscenza delle assunzioni e dei limiti legati all’ambiente di esecuzione del programma. L’ambiente di esecuzione, che chiamiamo “macchina”, è una combinazione e astrazione del linguaggio di programmazione e dell’hardware specifico su cui è eseguito. Conoscere i vincoli della “macchina” vuol dire, per esempio, sapere che i risultati delle operazioni aritmetiche su numeri decimali sono quasi sempre approssimati (questo è dovuto a come i numeri decimali sono rappresentati internamente dalla “macchina”) oppure sapere che un determinato linguaggio di programmazione (es. JavaScript) non è adatto a creare software che reagisca ad eventi in “tempo reale” al contrario di altri (es. Ada).

Esercizi suggeriti

- Analizzare (parte di) un linguaggio di programmazione elencando esplicitamente vincoli e regole (come sono salvati i valori in memoria? quali sono le regole dei costrutti fondamentali?)
- Comparare diversi linguaggi di programmazione: quali vincoli hanno in comune (es. qual’è il valore massimo che può assumere una variabile contenente un numero intero?), quali sono i loro ambienti di esecuzione (es. JavaScript esegue “sul browser”, Java sulla Java Virtual Machine)?

L’abilità **comprendere la specifica (R3)** rappresenta la capacità di capire come i requisiti/vincoli del “mondo” vengono soddisfatti entro i requisiti/vincoli della “macchina”. Vuol dire capire qual è “il progetto” che un programma cerca di implementare. Porre l’accento su questa abilità serve a rinforzare la comprensione del fatto che fare *debugging* non vuol dire riscrivere un programma da zero ma riconoscere la divergenza tra le aspettative e il risultato e cercare di colmare la distanza. È importante ricordare che i programmi sono raramente (quasi mai) *completamente errati* e che la creazione di soluzioni eseguibili Informatiche è spesso un processo di continuo raffinamento di un ragionamento e della sua implementazione.

Esercizi suggeriti

- Data una lista di requisiti del “mondo” e una lista che descriva componenti della “macchina” (procedure e moduli di un programma), collegare i primi alle seconde.
- Discutere come un algoritmo possa essere implementato in modi diversi o con diversi linguaggi di programmazione.

Il programma effettivo (T1, T2, T3, T4)

Il programma **effettivo** è il codice eseguibile che esiste concretamente e che può contenere gli errori per risolvere i quali si fa *debugging*. La letteratura scientifica ha spesso osservato che la capacità di scrivere programmi non implica necessariamente quella di leggerli e

comprenderli. Al contrario, una buona competenza nel leggere e comprendere programmi (anche scritti da altre persone) è un importante supporto alla capacità di crearne di nuovi. Per una discussione approfondita vedere: Lister, R. (2020, October). **On the cognitive development of the novice programmer: and the development of a computing education researcher.** In *Proceedings of the 9th computer science education research conference* (pp. 1-15).

Programma da analizzare

```
A)  i = 1
B)  somma = 0
C)  while i < 4:
D)    somma = somma + i
E)    if i è pari:
F)      somma = somma * 2
G)    i = i + 1
```

Quali istruzioni scrivono/leggono quali variabili? (T3)

Variable	Letta	Scritta
i	C D E G	A G
somma	D F	B D F

Scrivere la sequenza di esecuzione (T1)

Sequenza di esecuzione:
ABCDEGCDEFGCDEGC

Tabella di *tracing* (T2)

Sequenza:	ABCDEGCDEFGCDEGC
i:	1111122222333344
somma:	x001111336669999

Scrivere invarianti per il costrutto di selezione, righe E-F (T4)

```
(i >= 1) AND (i < 4) AND (i % 2 == 0)
```

Diagramma di flusso (T1)

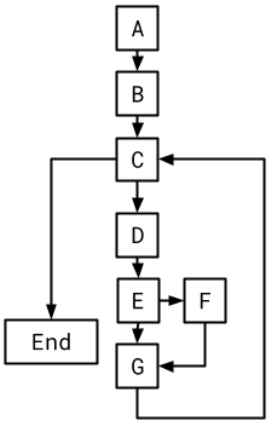

Fig. 2 - Esempi di esercizi relativi alla comprensione del programma effettivo

Comprendere il **flusso di controllo (T1)** vuol dire essere in grado di prevedere la sequenza di esecuzione delle istruzioni di un programma, che può essere influenzata per esempio da costrutti come l'iterazione (cicli *while*, *for*, ecc.) o la selezione (istruzioni condizionali).

Esercizi suggeriti

- Dato un semplice programma, etichettare (con lettere o numeri) le sue istruzioni e poi scrivere la sequenza in cui vengono eseguite. Nota: nel caso più semplice (come in Fig. 2) questo si può ridurre a etichettare le righe di un programma, tuttavia una stessa riga può contenere più istruzioni!
- Dato un semplice programma, disegnare un diagramma di flusso che schematizzi l'ordine in cui le istruzioni vengono eseguite (ed eventuali strade alternative dovute per esempio a costrutti di selezione).

La capacità di fare ***tracing* meccanico (T2)** unisce l'analisi del flusso di controllo (T1) all'analisi di come lo stato di esecuzione (i valori delle variabili di un programma) viene influenzato dall'esecuzione. Un classico esempio di *tracing* meccanico è costruire una **tabella di *tracing***, le cui colonne rappresentano la sequenza di istruzioni (ottenuta applicando T1) e le cui righe (una per variabile) contengono i valori che le variabili assumono mano a mano che le istruzioni vengono eseguite.

Esercizi suggeriti

- Dato un semplice programma che manipoli una o più variabili (con valori di partenza predefiniti), prevedere quale sarà il valore finale di una variabile.
- Costruire una tabella di *tracing* per tutte le variabili di un programma.

Comprendere il **flusso di dati (T3)** vuol dire capire come e da quali istruzioni vengono manipolate le variabili di un programma. Questa abilità è importante nel *debugging* perché permette, a volte, di trovare gli errori facendo *tracing* "a ritroso": se si scopre che una variabile ha un valore errato, si possono indagare le istruzioni che modificano il valore di quella variabile (ignorando le parti del programma non coinvolte e riducendo quindi il carico cognitivo).

Esercizi suggeriti

- Costruire una tabella che elenchi, per ogni variabile di un semplice programma, quali istruzioni la modificano (scrivono) o la leggono

La capacità di fare ***tracing* astratto (T4)** coordina le abilità precedenti (T1, T2, T3) per descrivere un programma a partire da asserzioni logiche che ne descrivono proprietà generali. Questa asserzioni possono essere, per esempio, precondizioni (cose che sappiamo essere sempre vere *prima* dell'esecuzione di un blocco di codice), postcondizioni (cose che sappiamo essere sempre vere *dopo* l'esecuzione di un blocco di codice) e invarianti (cose che sappiamo essere vere *sempre* in un punto qualsiasi del programma). Per esempio: un programma che contiene una istruzione di selezione che controlla se una certa variabile, divisa a metà, ha resto zero, avrà due rami, uno in cui quella variabile contiene sicuramente un valore pari e uno in cui quella variabile contiene sicuramente un valore dispari.

Esercizi suggeriti

- Analizzare un programma per individuare quante più precondizioni, postcondizioni e invarianti possibili.

Il processo di debugging (D1, F1, I1)

Le tre abilità direttamente collegate alla "radice" del modello rappresentano il **processo** di *debugging*, ovvero la sequenza ciclica di diagnosi e soluzione degli errori, possibilmente accompagnata dall'utilizzo di strumenti.

Diagnosticare l'errore (D1) vuol dire "provare" l'esistenza di un errore in modo inconfutabile. Questo può essere fatto in vari modi: si può mostrare (facendo leva sul *tracing* meccanico) che alcuni valori in ingresso a una programma portino a risultati non attesi; si

può dimostrare (facendo leva sul *tracing* astratto) come intere *classi* di errori portino *sempre* a risultati non attesi, ecc. Diagnosticare l'errore vuol dire anche essere in grado di riprodurlo nell'esecuzione di un programma, di conseguenza capendo esattamente quali sono le condizioni che portano a una esecuzione non riuscita. Porre l'enfasi sul questa abilità serve a ostacolare l'istintivo approccio al *debugging* basato sull'"andare a tentoni" modificando per tentativi pezzi del programma fino ad ottenere un risultato accettabile.

Esercizi suggeriti

- Dato un programma e una descrizione della sua "specifica", ovvero una sorta di "contratto" che descriva cosa quel programma dovrebbe fare, costruire delle verifiche (test) di questo contratto. Per esempio coppie di dati in ingresso e valori attesi che verifichino la soddisfazione dei requisiti.
- Dato un programma (con specifica nota) contenente un errore di implementazione o un errore logico, indicare qual è l'errore e fornire quante più "prove" possibili della sua esistenza. Nota: l'obiettivo qui non è *indicare una soluzione* ma solo *provare la presenza del bug*.

Correggere l'errore (F1) vuol dire modificare il programma *effettivo* per farlo convergere verso il programma *atteso* (oppure, nel caso di errori logici, aggiustare il programma atteso e, di conseguenza, quello effettivo). A valle di tutte le abilità discusse fino ad ora, questa abilità coincide largamente con le competenze di programmazione.

Esercizi suggeriti

- Dato un programma con un errore *esplicitamente evidenziato*, modificare il programma per risolvere l'errore (questo può essere il secondo passo di un esercizio che inizialmente chieda di provare la presenza di un errore, vedere D1).

Usare strumenti (I1) è la competenza legata all'utilizzo di tutti gli strumenti che vengono coinvolti in un'attività di *debugging*. Questi includono eventuali editor di testo, terminali, debugger, ecc. È importante ricordare che l'utilizzo di qualsiasi strumento crea carico cognitivo e può quindi essere di ostacolo all'apprendimento di una nuova abilità. Il suggerimento è quindi di dedicare delle sessioni di apprendimento all'utilizzo degli strumenti prima di utilizzarli in attività educative indirizzate all'apprendimento di altre abilità. Per esempio: può essere opportuno costruire degli esercizi mirati unicamente a imparare a usare un debugger che esegue un programma passo per passo *prima* di utilizzare il debugger in una attività in cui si deve diagnosticare o correggere un errore in un programma.

Esercizi suggeriti

Gli esercizi per questa competenza dipendono in larga parte dagli strumenti specifici che si vogliono utilizzare. Il suggerimento generale è di costruire delle attività i cui obiettivi di apprendimento siano specificamente legati all'*utilizzo degli strumenti*. Quindi per esempio un esercizio dovrebbe avere come obiettivo insegnare come utilizzare la funzione di esecuzione passo-passo di un debugger e *non* (almeno inizialmente) quello di usare il debugger per individuare un errore o costruire una tabella di *tracing*.