Debugging, Tracing, and Fading: Improving the Mental Model

Helena Rasche

2022-03-21

Students have a poor working model of code which leads to numerous issues; they struggle to debug, to adjust copy-pasted code, and to predict runtime behaviour.

Existing research provides a number of avenues for improving our coding education and improving the student experience. Here we discuss a number of strategies to enhance student learning, provide a deeper understanding of code, and improve the student mental model of code execution. This should allow students to thrive as budding coders.

The Problem

Currently students finish the course with a moderate working understanding of code, not sufficient to write their own programs or adapt new code to their situation. This is an extremely suboptimal outcome, after 8 weeks they should be able to write moderately complex python programs.

Showing what happens live on the screen is received well by students, if they can manage to watch what we type and try to type it themselves simultaneously. We know at least that our examples give the correct result, but students never see anything other than correct, working code, and never have to formulate an internal model for how to write code. They end up copying and pasting and not understanding *why*. As of now nothing is going "well", and there is room for improvement in all aspects, but specifically this discussion will focus on improving mental models of code and the potential effects on understanding code execution.

Predicting code behaviour without running it is a key component of work as a programmer, and a lot of the time we spend debugging relies on us emulating the computer in our head. Without a solid mental model of code behaviour one cannot predict how it will function in one situation, much less other or non-standard situations. Planning for code to handle both good and bad inputs requires some creativity and mentally planning around expected values at various points throughout the execution.

This situation leaves students unprepared for incorrect or buggy

code, either (un)intentionally included in homework assignments, or, generated by themselves, if they cannot identify where code will fail without executing it.

The Conclusion

Students must develop a coherent and strong internal model of code execution to allow them to understand code flow, fix broken code, and debug issues.

My Hypothesis

Augmenting lessons with:

- Tracing - Stepping through the internal state
- Faded examples
- Debugging intentionally broken examples

Will give students enough tools to respond dynamically to failure states with informed experience to resolve issues they encounter as programmers.

Opportunities for Improvement

Mental Model

The student's mental model of the code underlies everything they do as a programmer, from conception to implementation to debugging to their self efficacy:

> This study shows that a well-developed and accurate mental model directly affects course performance and also increases self efficacy, the other key element in course performance. Given this double impact, helping students develop good mental models should remain a goal in introductory programming courses. [Ramalingam et al., 2004]

This is a foundational skill to be able to *think* through a program, step by step, and understand how the code executes and which variables exist when, and what their values should be. This mental modelling allows students to predict the behaviour of a system, and when it diverges from their prediction, recognise any potential bugs.

Tracing

While there is no bug in fig. 1, when there *is* a bug present, having students produce a table like fig. 2 significantly improves their understanding of code flow and execution [Hertz and Jump, 2013]. "Tracing" is a valuable and easy to complete exercise, and the results can even

```python
# Initialise our accumulator
x = 1 + 1
# Loop over our input data
for i in range(10): # 0..9
    # In-loop temporary variable
    tmp = x * 2 + i
    # Update our accumulator
    x = tmp + 1
# Output our result
print(f'The final value is {x}')
```

Figure 1: Example Python Code. While not representative of a real world workflow, it is useful as an illustrative example, and an example which can be given directly to beginners for them to attempt.

Line	i	x	tmp
2	-	2	-
4	0	2	-
6	0	2	4
8	0	5	4
4	1	5	-
6	1	5	11
8	1	12	11
4	2	12	-
6	2	12	26
8	2	27	26

Figure 2: Example of a student's process tracing execution flow

be checked automatically leading to good scalability of the exercise across larger classes.

Faded Examples

Given that the students taught by Avans are primarily novice programmers who have not read or written a programming language before, we need to take significant care of their cognitive load. Both learning based on problem-solving and worked examples may cause high cognitive loads for different audiences, and exploring alternatives is important [Retnowati, 2017]. Faded examples such as what is seen in fig. 3 are exactly such an alternative, starting with a fully worked example and removing successive components until we reach a problem description requiring a full solution. This leads to fewer unproductive learning events [Renkl et al., 2004].

Faded examples however, do come at a higher cost of implementation than worked out examples [Zamary and Rawson, 2018]. They require writing the correct worked out example and then determining which components to remove, which presents an additional cost during course updates that if examples are changed they need to be double checked to ensure they are still valid, whereas worked examples can be checked more automatically.

```python
# Write a function that multiplies two numbers
def multiply(a, b):
    c = a * b
    return c
```

```python
# Write a function that adds two numbers
def add(___):
    _____
    return c
```

```python
# Write a function that subtracts two numbers
```

Figure 3: Example of fading in coding exercises, these let students work on a continuum from worked out examples to self-devised solutions.

Debugging

Debugging is the act of identify and resolving "bugs" or defects within code, a term popularly attributed to my personal hero Admiral Grace Hopper:

> While she was working on a Mark II computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system [Wikipedia contributors, 2022]

Debugging also functions as a reinforcement method we can use once students have an ok mental model of code execution, a necessary pre-requisite for this activity, which can be further developed through debugging [Ramalingam et al., 2004] alongside their self-efficacy [Michaeli and Romeike, 2019]. Debugging activities can take many forms but most commonly the task is to correct incorrect code, an activity that works best if they are primed with a number of methods of debugging [Murphy et al., 2008] such as the "Wolf Fence" [Gauss, 1982], commenting out code, or breakpoints.

```python
# Fix me!
for number in range(10):
    # use a if the number is a multiple of 3,
    # otherwise use b
    if Number % 3 == 0:
        message = message + a
    else:
        message = message + "b"
print(message)
```

Figure 4: A debugging exercise featuring code with numerous issues from type confusion, variable typos, and failure to initialise a variable.

Pair Programming

Complementary to the efforts of Mental Model development via Tracing/Debugging/Fading, pair programming or "pairing" (fig. 5)

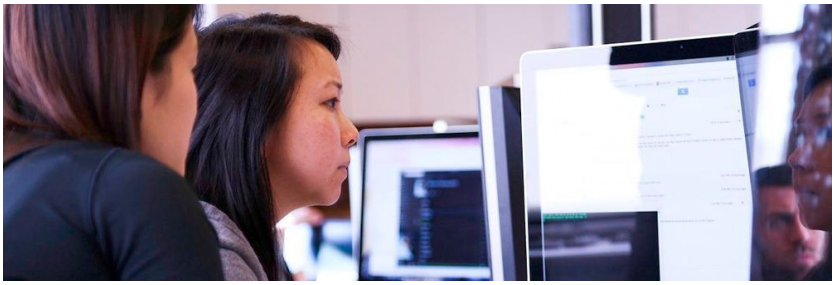

Figure 5: Pair programming has two participants join forces to develop solutions, one leading the effort, one writing the code. (Image: Startup-StockPhotos.com / CC0)

provides a reinforcement activity where they utilise similar skills. As one person writes and executes code, the other person 'drives' the experience, telling them what to write [Williams, Williams and Upchurch, 2001]. It has become a common learning model in introductory courses due to its benefits to students [Mendes et al., 2005, 2006, Hannay et al., 2007]. Specifically this technique has also been shown to be beneficial for women in computer science and gives them better chances for success in future programming endeavours [Werner et al., 2004]. Adopting this technique already (Assignment 6) has shown initially promising results, provided we adhere to principles outlined by [Mentz et al., 2008].

The Intervention

The intervention will consist of overhauling the formative assessments that students complete during class to include these new types of problems incorporating tracing exercises, faded examples, and debugging problems. These assessments work in unison and cannot be extracted, so they will be executed together. The example code listings provided (figs. 1, 3 and 4) are representative of the updates to formative assessments that we will make. The choice for this specific intervention instead of pair programming[1]

- This intervention is well backed by research showing improvements in student proficiency and grades [Hertz and Jump, 2013, Renkl et al., 2002] leading to better learning outcomes [Renkl et al., 2004].
- These skills significantly improve self efficacy [Ramalingam et al., 2004, Michaeli and Romeike, 2019].

The second point, lack of self-efficacy has been a significant cost for TOAs and teachers alike in terms of interruptions for questions students should be able to answer on their own, and by applying these interventions hopefully students will find themselves and peers a more reliable source of discussion and answers. Self empowerment in this field sets the students on a good path for their future.

[1] Additionally pair programming is already being trialled in one section.

Assessment

It is difficult to conduct a sufficiently controlled experiment given individual teacher preferences and teaching styles, so assessment will be carried out via review of student homework solutions. The system we use that collects student homework submission tracks student code entry over time, and will let us review how they attempted solutions and let us make qualitative observations.

The Awaited Results

Unfortunately there was not sufficient time to trial all of these strategies and evaluate the sections. Instead, all of these are currently being trialled in Group 32MBI02, please check back at the end of P3 for results. It is expected that these research backed teaching strategies will significantly improve student learning.

However, given that existing similar models, incorporating all of these activities, is used in K-12 teaching [Sentance and Waite, 2017], this intervention is expected to produce good results. Their model, PRIMM, starts with a good mental model which is required to predict, tracing during investigation, and debugging to modify code, all building towards students making things themselves.

References

Edward J. Gauss. The wolf fence algorithm for debugging. *Communications of the ACM*, 25(11):780, nov 1982. DOI: 10.1145/358690.358695. URL https://doi.org/10.1145%2F358690.358695.

Jo E. Hannay, Dag I.K. Sjoberg, and Tore Dyba. A systematic review of theory use in software engineering experiments. *IEEE Transactions on Software Engineering*, 33(2):87–107, feb 2007. DOI: 10.1109/tse.2007.12. URL https://doi.org/10.1109%2Ftse.2007.12.

Matthew Hertz and Maria Jump. Trace-based teaching in early programming courses. In *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13*. ACM Press, 2013. DOI: 10.1145/2445196.2445364. URL https://doi.org/10.1145%2F2445196.2445364.

Emilia Mendes, Lubna Basil Al-Fakhri, and Andrew Luxton-Reilly. Investigating pair-programming in a 2nd-year software development and design computer science course. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 296–300, 2005.

Emilia Mendes, Lubna Al-Fakhri, and Andrew Luxton-Reilly. A replicated experiment of pair-programming in a 2nd-year software development and design computer science course. *ACM SIGCSE Bulletin*, 38(3):108–112, 2006.

E. Mentz, J. L. van der Walt, and L. Goosen. The effect of incorporating cooperative learning principles in pair programming for student teachers. *Computer Science Education*, 18(4): 247–260, dec 2008. DOI: 10.1080/08993400802461396. URL https://doi.org/10.1080/08993400802461396.

Tilman Michaeli and Ralf Romeike. Improving debugging skills in the classroom. In *Proceedings of the 14th Workshop in Primary and Secondary Computing Education*. ACM, oct 2019. DOI: 10.1145/3361721.3361724. URL https://doi.org/10.1145%2F3361721.3361724.

Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging. *ACM SIGCSE Bulletin*, 40(1):163–167, feb 2008. DOI: 10.1145/1352322.1352191. URL https://doi.org/10.1145%2F1352322.1352191.

Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. Self-efficacy and mental models in learning to program. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '04*. ACM Press, 2004. DOI: 10.1145/1007996.1008042. URL https://doi.org/10.1145%2F1007996.1008042.

Alexander Renkl, Robert K. Atkinson, Uwe H. Maier, and Richard Staley. From example study to problem solving: Smooth transitions help learning. *The Journal of Experimental Education*, 70(4): 293–315, jan 2002. DOI: 10.1080/00220970209599510. URL https://doi.org/10.1080%2F00220970209599510.

Alexander Renkl, Robert K. Atkinson, and Cornelia S. Große. How fading worked solution steps works – a cognitive load perspective. *Instructional Science*, 32(1/2):59–82, jan 2004. DOI: 10.1023/b:truc.0000021815.74806.f6. URL https://doi.org/10.1023%2Fb%3Atruc.0000021815.74806.f6.

E Retnowati. Faded-example as a tool to acquire and automate mathematics knowledge. *Journal of Physics: Conference Series*, 824: 012054, apr 2017. DOI: 10.1088/1742-6596/824/1/012054. URL https://doi.org/10.1088%2F1742-6596%2F824%2F1%2F012054.

Sue Sentance and Jane Waite. PRIMM. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*. ACM,

nov 2017. DOI: 10.1145/3137065.3137084. URL https://doi.org/10.1145%2F3137065.3137084.

Linda L Werner, Brian Hanks, and Charlie McDowell. Pair-programming helps female computer science students. *Journal on Educational Resources in Computing (JERIC)*, 4(1):4–es, 2004.

Wikipedia contributors. Debugging — Wikipedia, the free encyclopedia, 2022. URL https://en.wikipedia.org/w/index.php?title=Debugging&oldid=1069955193. [Online; accessed 9-February-2022].

L. Williams. Integrating pair programming into a software development process. In *Proceedings 14th Conference on Software Engineering Education and Training. 'In search of a software engineering profession' (Cat. No.PR01059)*. IEEE Comput. Soc. DOI: 10.1109/csee.2001.913816. URL https://doi.org/10.1109/csee.2001.913816.

Laurie Williams and Richard L. Upchurch. In support of student pair-programming. *ACM SIGCSE Bulletin*, 33(1):327–331, mar 2001. DOI: 10.1145/366413.364614. URL https://doi.org/10.1145/366413.364614.

Amanda Zamary and Katherine A. Rawson. Are provided examples or faded examples more effective for declarative concept learning? *Educational Psychology Review*, 30(3):1167–1197, mar 2018. DOI: 10.1007/s10648-018-9433-y. URL https://doi.org/10.1007%2Fs10648-018-9433-y.