Mining Jewels Together: Debating about Programming Threshold Concepts in Large Classes

Manuel Selva, François Broquedis

Mining Jewels Together: Debating about Programming Threshold Concepts in Large Classes

Manuel Selva
manuel.selva@inria.fr
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
38000 Grenoble, France

François Broquedis
francois.broquedis@grenoble-inp.fr
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
38000 Grenoble, France

ABSTRACT

Computer science has its threshold concepts as any academic discipline that students struggle to understand deeply. In this work, we report on using an active learning method called *scientific debate* to teach programming threshold concepts in large unplugged classes. This method involves students by having them defend their position with arguments and scales up with the number of students by leveraging collective intelligence. We present in detail how we apply scientific debate in a programming class. We also collect and discuss student exchanges during debates and gather feedback after the last debate. Students report they stay more focused and motivated during class with scientific debate compared to traditional transmissive lectures. They also indicate that they understood the goal of this new pedagogical contract. Our experience report could help programming teachers willing to replace/complement transmissive teaching with a methodology involving the students.

KEYWORDS

active learning; threshold concepts; scientific debate; programming

1 INTRODUCTION

Threshold concepts are these jewels that, once crossed, strongly impact learners. Following their initial definition by Meyer and Land [14, 15], these concepts are *troublesome*, *transformative*, *integrated*, and *irreversible*. Said differently, threshold concepts are hard to grasp; they change how students perceive a given discipline; they allow making relations within a discipline, and students cannot forget them once learned. In the context of computer science, particularly in learning to program, the question of identifying threshold concepts has been studied [2, 23, 27, 32] since the introduction of the concept by Meyer and Land. Even if the question is not closed, these works identified and discussed several concepts as being threshold concepts or at least potential ones. For example, the *program dynamics* [27, 29] concept is a threshold one. We refer to the following concise sentence [1] to give the reader an understanding of what this concept is about: "A running program is a kind of mechanism, and it takes quite a long time to learn the relation between a program on the page and the mechanism it describes."

Once teachers identify a threshold concept, the question arises about how to help students cross it. Practicing a lot is part of the answer. That being said, teachers have wondered how to speed up the learning curve regarding these concepts since the early days of teaching programming. As in any discipline with large cohorts of students, programming teachers also faced the question of using pedagogical methods scaling with the number of students. The traditional way of answering this question consists of carefully prepared transmissive lectures.

One of the main criticisms that one can formulate regarding transmissive lectures is that students are not scientifically engaged. Nevertheless, there exist active learning approaches aiming at involving students, making them actors of the class. *Scientific debate* [9] is one of these approaches developed in the context of teaching mathematics. In this socio-constructivist approach, students become actors in the class by having to defend with arguments their position on a problem during a debate. While the debate runs, the teacher's role is only to animate discussions without taking a position on any argument students offer. The debate runs until students no longer bring new arguments or when the amount of time allocated for it has elapsed. At this time, the teacher can give the correct answer and explain why connecting her explanations to the students' correct and wrong arguments. Even if mathematics education researchers have proposed the scientific debate approach before the seminal work on threshold concepts by Meyer and Land, it is a perfect fit for threshold concepts and later work on the method explicitly says that it tackles the "key concepts" of a discipline [10, 18].

In this work, we report and discuss our usage of scientific debate in a programming class at the license 3 level (in the European system). Section 2 presents related work and background. Section 3 lists with details the four problems we use to animate four debates. We then report in Section 4 on the student arguments we collected during the last three years for each of the four debates. Section 5 presents student feedback on this teaching methodology. We conclude in Section 6. We emphasize that we do not tackle the question of pinpointing threshold concepts in programming but focus on reporting our usage of scientific debate to tackle four concepts already identified as potential thresholds by the community.

2 RELATED WORK AND BACKGROUND

Meyer and Land introduced the notion of threshold concepts twenty years ago [14, 15]. Since then, the computer science education community has worked on applying this theory to its field. Many studies focus on listing the threshold concepts of computer science [2, 27, 29, 32] and on defining a methodology to decide whether a computer science concept can be considered a threshold or not [22, 23, 33]. The conclusions suggest that one should take students, teachers, and professionals' points of view into account to understand computer science threshold concepts better.

Some studies go beyond identifying the computer science threshold concepts, focusing on the impact of these concepts on how educators teach computer science. As such, some of them propose teaching material dedicated to specific threshold concepts. Sanders and McCartney listed [23] teaching materials already in use by the community to teach threshold concepts. Sorva suggests, for

example, using visualization software to teach the threshold concept known as *program dynamics* [28]. Shinners-Kennedy proposed classroom activities with mazes to address some computer science threshold concepts [24].

Other more general works focus on the high-level properties a class and associated teachers must have to teach computer science threshold concepts [4, 12, 13, 21, 27, 29]. Several key points emerged from these works. First, students must know the potential threshold concepts' existence and importance [27, 29]. Second, while studying a threshold concept, students oscillate between a state where they think they got the concept and a state where they face facts showing it is not the case [4, 12, 13]. This oscillation period is called the *liminal space*. Because of this liminal space, the teaching approach must consider the affective aspect of threshold concepts [21]. Finally, teachers must find ways to reveal a threshold concept's profound nature [27] or the "underlying game" [16] in Mayer and Land's words.

Without a relationship to threshold concepts, Think-Pair-Share (TPS) [11] and Peer Instruction (PI) [3] are two lightweight active learning methods already in use in the context of large computer science classes. Both methods are similar to scientific debate, requiring students to think and debate on a given problem. In the case of PI, as for scientific debate, students must vote for a given answer, increasing scientific engagement. In the context of teaching programming, TPS can strongly impact learning outcomes, increases student engagement, and is easy to implement [7, 8]. PI fosters deep understanding [20, 25, 26] and helps build technical communication skills [19].

Legrand proposed an active learning teaching method called scientific debate [9, 10] to tackle threshold concepts in the context of mathematics. This method has mainly been used in mathematics [5, 18, 31]. Compared to TPS and PI, scientific debate explicitly targets threshold concepts. Said differently, scientific debate requires more student involvement, preparation on the teacher's side, and more time in class. In our programming class, a single scientific debate occupies a full 90-minute lecture. The method leverages a short, nevertheless conceptually complex (since it relates to a threshold concept) and potentially controversial problem to engage all the students and highlight pre-conception to deconstruct the wrong ones.

3 FOUR SCIENTIFIC DEBATES

In our teaching context, we focus on four concepts identified by the community as candidates for being threshold. This section introduces our programming class and presents the four problems given to students for each of the four threshold concepts.

3.1 Organization of our programming class

The programming class title is *Basis of Imperative Programming (BIP)* and uses Python. It stands as the first programming class in our computer science curriculum. As the title suggests, its primary goal is to focus on concepts rather than mastering Python. While around 95% of the 240 students attending the class already have some programming experience, the audience's level is very heterogeneous. Students with previous programming experience fit into three groups with three different backgrounds. In the group with the most significant programming experience, representing 15% of the 95%, students already followed 200 hours of programming courses, mainly in Python. In the group with the lowest experience, representing 20% of the 95%, students had less than 50 hours of programming courses before reaching our class. In the last group, students attended 120 hours of programming courses before.

The class comprises four sequential chapters corresponding to the four threshold concepts and lasts for a semester made up of 11 weeks. Chapter One lasts three weeks, chapter Two lasts three weeks, chapter Three lasts three weeks, and chapter Four lasts two weeks. Each chapter starts with a 90-minute lecture with all students, including the scientific debate, to introduce the threshold concept. The students then have one lab session and one unplugged session per week in groups of 30 students with a dedicated teacher. These per-group sessions aim to implement real programs on a machine and do paper exercises related to the chapter's threshold concept to help students cross it while simultaneously learning technical skills. Each of the four 90-minute lecture with a scientific debate follows this structure :

(1) an introduction by the teacher followed by the exposition of the problem and the proposed answers ;
(2) enough time (at least 10 minutes) for each student to think about the problem on their own and discuss it with neighbors ;
(3) a first individual vote followed by the posting of the result of the vote for the entire class ;
(4) a debate between students where each one defends her position ;
(5) a second individual vote followed by the posting of the result of the vote for the entire class ;
(6) another discussion time for students who have changed their minds to explain why ;
(7) a restructuring by the teacher, taking up the arguments heard and explaining why they are correct or not, called the *institutionalization* phase.

This organization, respecting the principles of the scientific debate method allows students to think individually about the problem first. The individual vote aims to promote scientific engagement among all students. The debate is the heart of the class, where students are encouraged to explain what they think to others. Its purpose is to let them defend their position with arguments on their terms. The results of the new vote provide feedback on how convincing the students' arguments were. Students are again encouraged to speak up and explain which arguments have convinced them and why. Finally, the teacher's role is to take up all the arguments heard, indicating which are correct and which are not, and then explaining why. For wrong arguments, the teacher explains where can be the origin of the misconception.

3.2 Program dynamics

The first threshold concept tackled in the class is the program dynamics one we presented in the introduction. The problem presented for debate to the students is the following: What does the Ruby program in Listing 1 print on screen? Along with the program and the question, the teacher proposes the following answers to students:

```ruby
def f()
  print "welcome "
  g()
  return
  print "! "
end
print "in BIP "

def g()
  print "for the season "
end

def h(phrase)
  pos = 0
  while pos < phrase.length
    ch = phrase[pos].chr
    pos += 1
    if ch == ")" then break
    elsif ch == "(" then next
    end
    print ch
  end
end

def i(x, y)
  print " which "; j(x, y)
  print "cool"
end

def j(y, x)
  print "gonna be "; y/x
end

print "we'll work"
x = -1
begin
  x += 1
  if not 7 <= (6 + x) then
    print "with fun "
  else print "Repeat: "
  end
  f()
  h("22(23)!")
  i(42, x)
rescue
  retry
end
```

Listing 1: Problem to debate about program dynamics

- A. *welcome in BIP for the season 2223 which gonna be I repeat: welcome for the season 2223 which gonna be cool*
- B. *in BIP we'll work welcome for the season 2223 which gonna be*
- C. *in BIP we'll work with fun welcome for the season 2223 which gonna be*
- D. *in BIP we'll work with fun welcome for the season 2223 which gonna be cool*
- E. *in BIP we'll work with fun welcome for the season 2223 which gonna be Repeat: welcome for the season 2223 which gonna be cool*
- F. *in BIP we'll work with fun welcome for the season 2223 ! which gonna be Repeat: welcome for the season 2223 which gonna be cool*

We decided to use Ruby for two reasons. First, very few attending students know this language[1], so they start with the same level of knowledge regarding this problem's syntax. Second, using a new language should help convince students that program dynamics is a computer science threshold concept since it exists independently of the programming language used.

We deliberately use many control flow features of imperative languages to emphasize the crucial importance of program dynamics for understanding a program. Among these features, some are complex such as exceptions, and have never been seen before by any student.

The goal of a scientific debate is not to have all students have a deep understanding of the threshold at the end of the lecture but to have them realize that they need to work hard on this concept.

In order to have more impact on students, we start the institutionalization phase by running the program to check the correct answer. Then, the teacher executes the program step by step with a visualization tool to see the following line to be executed along with the content of the global area and local variables. Again, during this step-by-step execution, the teacher must connect her explanations to correct and wrong arguments students gave during the debate.

[1]every year before the first debate, we ask students what are the programming languages they already used and very few mention Ruby.

```python
def do_stuff(nb):
    r, m = [], 0
    for _ in range(nb):
        i = randint(0, nb)
        if is_odd(i):
            r.insert(m, i)
            m += 1
        else:
            r.insert(len(r),i)
    return r

def is_odd(integr):
    return bool(integr % 2)

n = input("Enter number: ")
l = do_stuff(int(n))
f = open("out.data", "w")
for elem in l:
    print(elem, file=f)
f.close()
```

Listing 2: Problem to debate about data structures

3.3 Data structures versus abstract data types

The second threshold concept we address is data structures [17, 32]. The question students debate on is how many CPU instructions are required to execute the Python program shown in Listing 2, considering the average overhead induced by executing one Python statement is approximately 100 CPU instructions. We then propose the following answers:

- A. $100 \times (nb \times x)$ with $x < 1000$
- B. $100 \times (nb \times x)$ with $x < constant$
- C. $100 \times (nb \times x)$ with x depending on the results of `randint`
- D. more than the 3 answers above
- E. something else

This simple program allows students to intensely discuss data structures even if they have never heard about the term. Indeed, most have already used Python `list` objects and know they can insert and remove elements from them. Nevertheless, only some have already wondered what is happening behind the scenes. Explicitly asking for counting CPU instructions force them to think about what happens internally and introduces the distinction between the operations available on a Python list (the abstract data type) and how the interpreter implements it (the data structure).

Another advantage of this simple program is that running it in front of students with a not-so-large number, such as 500, 000, shows them the tangible impact of using an inadequate data structure: the program runs for 30 seconds!

During institutionalization, the teacher explains that what Python calls `lists` are dynamic arrays. Again, we use visual representation to support this explanation to have students see the shift of all elements of one position to the right when inserting at index zero.

3.4 References

The third threshold concept we tackle is references [2, 30, 32]. For this one, students must guess the output of the Python program shown in Listing 3. The proposed answers are the following:

- A. *white and blue columns*
- B. *white and blue lines*
- C. *white and blue chessboard*
- D. *something else*

Again, despite its apparent simplicity, this program allows students to dig into the concept of references deeply. Indeed, many students have already encountered the $\star$ operator. However, few have ever been interested in what its use implies and what a Python `list` contains in the interpreter.

```python
def print_stuff(s):
  for i in range(8):
    for j in range(8):
      w = s[i][j]
      if w:
        c = "white"
      else:
        c = "blue"
      r = "\u2588" # rectangle
      t = colored(f"{r}{r}",c)
      print(t, end="")
    print()

def init_stuff():
  res = [[]] * 8
  start_w = True
  for i in range(8):
    white = start_w
    for j in range(8):
      res[i].append(white)
      white = not white
    start_w = not start_w
  return res

print_stuff(init_stuff())
```

Listing 3: Problem to debate about references

```python
def m(i, elems):
    idx = len(elems) - 1 - i
    elems[idx] += 1
    if i == 0: return 1
    return m(i - 1, elems) + \
           m(i - 1, elems)

i = 5
elems = [0] * (i + 1)
print(m(i, elems), elems)
i = 60
elems = [0] * (i + 1)
print(m(i, elems), elems)
```

Listing 4: Problem to debate about recursive functions

Because this program visually displays something, as for the running time of the previous program, showing the result can impact students and facilitates their interactions with the teacher. Indeed, the numerous reactions we can hear from the class just after the teacher presses `Enter` to run the program when the debate is over is an indicator confirming it.

As for the two previous debates, the teacher uses a visualization tool à la PythonTutor [6] during the institutionalization phase to explain the reference concept.

3.5 Recursive functions

The last concept we study is recursive functions, also identified as a potential threshold concept [2, 22] or, more precisely, a transliminal concept [27]. Sorva defines transliminal concepts as "concepts that require an understanding of a threshold concept and can lure students to and across thresholds." In this sense, recursive functions are a transliminal concept of the program dynamics threshold since one can only understand recursive functions with a good understanding of program dynamics.

We nevertheless decided to keep recursive functions apart from program dynamics. We primarily made this decision for pragmatic reasons. Indeed, we find the program dynamics threshold concept hard to cross for many students. Moreover, understanding recursive functions is still a challenge for those who manage to cross the program dynamics threshold, as will be discussed in Section 4.

The question asked and submitted to debate is to guess the output for the code shown in Listing 4 with the following propositions:

- A. *two lines with several* `1` *on each line*
- B. *two lines with several powers of two on each line*
- C. *something else*

This simple program contains the required material to debate about recursive functions. Indeed, to be able to anticipate what will happen when running the program, students must scientifically engage themselves in a deep reflection.

For impacting students, the teacher launches the program at the beginning of the institutionalization phase and returns to it at the end to demonstrate that it is still running. Then the teacher shows, with an approximative computation considering the machine can execute 10^9 calls per second (which is more than the reality), that this program will run for about 73 years! The explanation of program dynamics in the presence of recursive calls is supported again by a visual tool we designed to show the evolution of the call tree on the first call to the recursive function with `i` equals 5.

4 STUDENTS EXCHANGES DURING DEBATES

This section reports and discusses the arguments given by students during each of the four debates. We collected these arguments during three class instances: fall 2020, fall 2021, and fall 2022.

4.1 Program dynamics

Here is the list of the arguments we got from students formulated in our teachers' words:

(1) The character *!* cannot be printed since the corresponding print statement is located right after a `return` statement.
(2) The character *!* must be printed after *for the new season* since the print statement is located right after the call to function `g`.
(3) The answer must start with *welcome* since it is the first `print` statement.
(4) The answer cannot start with *welcome* since the first `print` statement executed is the first one not in a function.
(5) I do not know anything about error handling, but something must repeat because of a division by zero and `retry`.
(6) *with fun* is not printed since `x` is `0` when reaching this line.

These arguments point out that students attending the class already have previous programming experience and know the existence of the program dynamics concept.

Nevertheless, arguments (2) and (3) demonstrate that many still need to cross this threshold. In particular, only one-third of the students provide the correct answer on the first vote each year.

While students may have several other misconceptions, they only reported two wrong arguments, as mentioned above. One reason is that getting student participation in this first debate is challenging. Indeed, this debate occurs during the first class of the first day of our computer science curriculum when students had just arrived. Students barely know each other at this time, and speaking in front of others to defend a position is intimidating. Moreover, the pedagogical method is new for them since they only followed transmissive lectures in the context of large classes before reaching our class.

4.2 Data structures versus abstract data types

Here are the arguments we collected for the second debate:

(1) It depends on what we insert and, consequently, the ratio between even and odd numbers.
(2) The running time of `randint(0, nb)` increases linearly with the value of `nb`.
(3) The running time of `len` increases linearly with the list size.
(4) The running time of `len` is constant.

(5) The answer is **A.** or **B.** or **C.** since there is a single loop iterating nb times.
(6) It depends on the implementation of `insert`.
(7) The running time of `insert` is constant.
(8) The running time of `insert` increases linearly with the list size.

For this debate, the percentage of correct answers after the first vote is usually around one-fourth. We explain this by the fact that even if most students already have programming experience in Python, they have never been introduced to data structures. Moreover, on this point, Python is a dangerous beast since the actual running time of the `insert` method depends on where the insertion happens.

On the other hand, Python lists are a good illustration of the importance of data structures. The last three arguments show that some students identified that `insert` plays a central role in the time complexity of this program. Consequently, the debate here is rich and lets students confront their opposing arguments in their own words.

4.3 References

Here are the arguments we collected for the debate on references:

(1) The answer is *white and blue columns* because the program only alternates colors inside lines.
(2) The answer is *white and blue chessboard* because the program alternates colors between and inside lines.
(3) The answer is *white and blue columns* because a single `list` is shared eight times.

The number of correct answers for the first vote is around one-third for this debate. Nevertheless, as the arguments show, some students have the correct answer, but not for the right reason. Again, the correct arguments impact students significantly more when they come from fellow learners rather than the teacher.

4.4 Recursive functions

The students defended the following arguments:

(1) The answer is **C.** because of a recursion depth error.
(2) The answer is **A.** because the base case returns 1.
(3) The answer is **B.** because each call recurses two times and then sums the two results.
(4) I do not know because I cannot anticipate execution.
(5) The answer is **C.** because `elems = [0] * i` is dangerous.
(6) The answer is **C.** because of computation time problems.

Less than five students provided the correct answer for this debate, even if they all worked a lot on program dynamics during the previous weeks. Hence, it is necessary to spend time on it and to have these few students explain in their own words what happens during the execution of the recursive function.

4.5 What conclusions to draw?

The first conclusion we can draw, considering the students' answers and the arguments they defended during debates, is a confirmation that the four concepts we tackle are thresholds. Hence, in our context, it is mandatory to insist on these concepts even if students already have non-negligible programming experience. Indeed, if most of them had crossed the four thresholds, we would have expected to get a significant fraction of correct answers to each of the four problems, and the debates would have been sterile. That was not the case. Said differently, students need time to cross threshold concepts. That is why we believe the first programming class of a computer science curriculum must insist on these thresholds even if many students already have previous programming experience.

The second conclusion from our experience is that the teacher must propose a well-calibrated problem to debate a given threshold successfully. It must be difficult enough to get different opinions from the audience, but not too difficult either, as students may struggle to have opinions on something too difficult to understand. More importantly, the problem must focus on the threshold and avoid as many details as possible. For example, the feedback we had animating the debate on recursive functions is that we should remove the `*` operator to initialize the `list` with zeros. Indeed, some students got lost on this point they knew to be important from the previous debate on references instead of focusing on the dynamics of recursive functions.

Another evident, nevertheless crucial, element regarding the success or not of a debate is the class's atmosphere. Indeed, it is crucial for students to feel and know they can freely give their arguments without being judged. Here, the teacher plays a central role in ensuring everyone respects every opinion during the debates. While this statement generally concerns the scientific debate method, we believe it is crucial in our programming class context. One reason is that we often have few students genuinely passionate about programmation in front of us that may already have crossed the thresholds. These students may be intimidating to others if they always give the correct arguments during the debate. It happened several times, but thanks to the excellent class atmosphere and the fact that many students did not have already crossed the threshold, these situations led to exciting debates between the few students with the correct arguments and the others, among which wrong arguments convinced some.

	2018	2019	2020	2021	2022
positive	8	7	4	13	4
neutral	44	38	49	37	27
negative	29	21	1	3	0

Table 1: Students global feedback over the years

5 STUDENTS FEEDBACKS

For five years, our university has asked students to evaluate classes when they finish. Students can leave comments about the course in a free text field on the evaluation form. We introduced scientific debate in 2020. Also, because of the pandemic that year, only the first two scientific debates occurred. Table 1 summarizes the results of these evaluations concerning the large classes of the BIP course. We accounted for the number of positive and negative feedback explicitly concerning the large classes. The neutral line is the number of students whose feedback only concerns group lab or unplugged sessions and does not mention large classes.

Negative feedback on the large unplugged classes has significantly decreased since we introduced scientific debate.

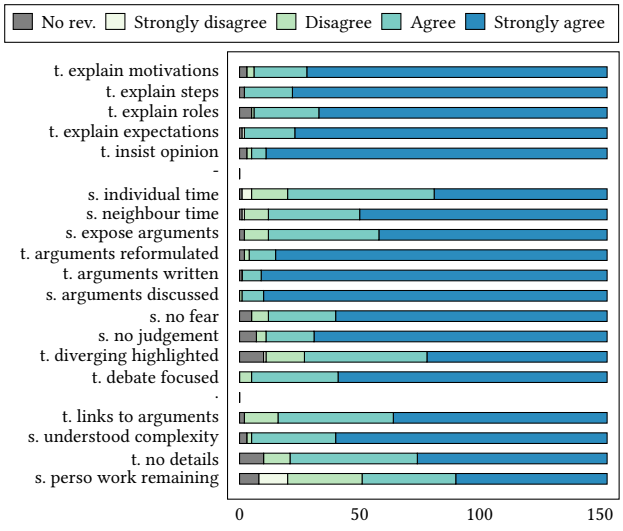

Figure 1: Students feedback on scientific debates

To get more insights from students this year, we asked for explicit feedback on the four debates. We did that at the end of the last debate to collect feedback from the 153 attending students. In this survey, we asked them to numerically indicate their level of agreement with 19 propositions listed on the left part of Fig. 1. A *[t]* at the beginning of a proposition means ***t**eacher*. For example, the first proposition means *the **t**eacher clearly explained motivations*. A *[s]* means *me as a **s**tudent*. For example, the sixth proposition means *I had enough individual time to think about the problem as a **s**tudent*. The numerical agreement values proposed were: (1) No reviews (2) Strongly disagree (3) Disagree (4) Agree (5) Strongly agree.

As shown in Fig. 1, we gathered the 19 propositions into three groups: (1) propositions about teacher explanations before the debate; (2) propositions regarding what happens during the debate; (3) propositions concerning the institutionalization phase after the debate. In the first group of propositions, the survey asked if the teacher gave clear explanations of different things and if he insisted on the importance of thinking individually to have his own opinion. In the second group, the survey asked students if they had enough time to think and discuss, if they could express their arguments without fear and judgment, and if the teacher wrote and discussed all the arguments and kept the debate focused. In the third group, the survey asked if the teacher linked arguments given during the debate with its explanations and if she kept away details. Finally, still in the third group, students were asked if they perceived the complexity of the threshold and if they thought they still had remaining work to cross it.

From the results of the first group propositions and the *[s] understood complexity* one in the last group, it is clear that students understood both the pedagogical purpose of scientific debate and the existence of programming threshold concepts and their complexity. It is a good result since, as stated in Section 2, students must be aware of the threshold concepts' existence and importance [29].

The second conclusion from the answers in Fig. 1 is that students were comfortable with this new pedagogical method. As teachers, one of our biggest fears when introducing the method was having few or no student debates. Nevertheless, except for the first debate, as already mentioned in Section 4, it was easy for us to maintain involved participation from students.

Finally, the results in Fig. 1 suggest that we must take more care to highlight diverging arguments and give more time for individual reflection. Indeed, the two associated propositions have the highest number of disagree and strongly disagree responses.

Another survey part was tree free forms for positive points, negative points, and enhancement proposals. In the positive points form, 42 out of the 153 students say the class format helps them stay focused during the class. In the negative part, 16 said the reflection time before the debate was too short, while 2 said it was too long. In the enhancement proposal, 11 students suggested doing more large classes, which we had never seen before introducing scientific debate.

Finally, the survey asked students to give a global note from 1 (bad) to 6 (excellent) on the scientific debate sessions. The average value from the 153 answers is **5.5/6**, which confirms that students understood the method and found it beneficial.

6 CONCLUSION

This paper presented our usage of the scientific debate method in the context of a programming class. The feedback provided by students up to now encourages us to pursue in that direction since it positively impacts students' motivation and concentration.

Also, even if this could have been identified by other means, students' vote results and arguments during debates indicate that many students have not crossed programming thresholds when they reach our curriculum. It is a logical conclusion of the nature of threshold concepts: they take time to be crossed. From our teacher's perspective, this suggests that we need to keep a closed eye on students' understanding of these concepts.

As a final note, the move to scientific debate has been a stimulating experience from the teacher's point of view. It forced us to sit together to identify and agree on the thresholds. Also, scientific debate requires us to adopt a different role than what we are used to in transmissive lectures. Although it was not easy initially, this was a challenging experience for us teachers.

REFERENCES

[1] Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

[2] Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. 2007. Threshold Concepts in Computer Science: Do They Exist and Are They Useful? *SIGCSE Bull.* 39, 1 (mar 2007), 504–508. https://doi.org/10.1145/1227504.1227482

[3] Catherine H. Crouch and Eric Mazur. 2001. Peer Instruction: Ten years of experience and results. *American Journal of Physics* 69, 9 (09 2001), 970–977. https://doi.org/10.1119/1.1374249

[4] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Kate Sanders, Lynda Thomas, and Carol Zander. 2007. From Limen to Lumen: Computing Students in Liminal Spaces. In *Proceedings of the Third International Workshop on Computing Education Research* (Atlanta, Georgia, USA). Association for Computing Machinery, New York, NY, USA, 123–132. https://doi.org/10.1145/1288580.1288597

[5] Hitt Fernando and Alejandro González-Martín. 2014. Covariation between variables in a modelling process: The ACODESA (collaborative learning, scientific debate and self-reflection) method. *Educational Studies in Mathematics* 88 (02 2014), 201–219. https://doi.org/10.1007/s10649-014-9578-7

[6] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

[7] Aditi Kothiyal, Rwitajit Majumdar, Sahana Murthy, and Sridhar Iyer. 2013. Effect of Think-Pair-Share in a Large CS1 Class: 83% Sustained Engagement. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (San Diego, San California, USA) *(ICER '13)*. Association for Computing Machinery, New York, NY, USA, 137–144. https://doi.org/10.1145/2493394.2493408

[8] Aditi Kothiyal, Sahana Murthy, and Sridhar Iyer. 2014. Think-Pair-Share in a Large CS1 Class: Does Learning Really Happen?. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education* (Uppsala, Sweden) *(ITiCSE '14)*. Association for Computing Machinery, New York, NY, USA, 51–56. https://doi.org/10.1145/2591708.2591739

[9] Marc Legrand. 2001. *Scientific Debate in Mathematics Courses.* Springer Netherlands, Dordrecht, 127–135. https://doi.org/10.1007/0-306-47231-7_12

[10] Marc Legrand, Thomas Lecorre, Liouba Leroux, and Anne Parreau. 2011. *Le principe du "débat scientifique" dans un enseignement.* Technical Report.

[11] Frank T Lyman. 1981. The responsive classroom discussion: The inclusion of all students. *Mainstreaming digest* 109 (1981), 113.

[12] Robert Mccartney, Jonas Boustedt, Anna Eckerdal, Jan Moström, Kate Sanders, Lynda Thomas, and Carol Zander. 2009. Liminal spaces and learning computing. *European Journal of Engineering Education* 34 (08 2009), 383–391. https://doi.org/10.1080/03043790902989580

[13] Robert McCartney, Anna Eckerdal, Jan Erik Mostrom, Kate Sanders, and Carol Zander. 2007. Successful Students' Strategies for Getting Unstuck. *SIGCSE Bull.* 39, 3 (jun 2007), 156–160. https://doi.org/10.1145/1269900.1268831

[14] J.H.F. Meyer and Ray Land. 2003. Threshold concepts and troublesome knowledge: Linkages to ways of thinking and practising within the disciplines. *Improving Student Learning - Ten Years on* (01 2003), 412–424.

[15] Jan Meyer and Ray Land. 2005. Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *High Educ* 49 (01 2005), 373–388. https://doi.org/10.1007/s10734-004-6779-5

[16] Jan Meyer and Ray Land. 2006. *Overcoming Barriers to Student Understanding. Threshold concepts and troublesome knowledge.*

[17] Jan Erik Moström, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Kate Sanders, Lynda Thomas, and Carol Zander. 2008. Concrete Examples of Abstraction as Manifested in Students' Transformative Experiences. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 125–136. https://doi.org/10.1145/1404520.1404533

[18] Yvan Pigeonnat. 2015. Grands auditoires : plus qu'un handicap, une force pour enseigner des concepts difficiles ! (jun 2015). https://doi.org/10.1145/2483710.2483713

[19] Leo Porter, Cynthia Bailey Lee, Beth Simon, Quintin Cutts, and Daniel Zingaro. 2011. Experience Report: A Multi-Classroom Report on the Value of Peer Instruction. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (Darmstadt, Germany) *(ITiCSE '11)*. Association for Computing Machinery, New York, NY, USA, 138–142. https://doi.org/10.1145/1999747.1999788

[20] Leo Porter, Cynthia Bailey Lee, Beth Simon, and Daniel Zingaro. 2011. Peer Instruction: Do Students Really Learn from Peer Discussion in Computing?. In *Proceedings of the Seventh International Workshop on Computing Education Research* (Providence, Rhode Island, USA) *(ICER '11)*. Association for Computing Machinery, New York, NY, USA, 45–52. https://doi.org/10.1145/2016911.2016923

[21] Charles W. Reynolds and Bryan S. Goda. 2007. The Affective Dimension of Pervasive Themes in the Information Technology Curriculum. In *Proceedings of the 8th ACM SIGITE Conference on Information Technology Education* (Destin, Florida, USA) *(SIGITE '07)*. Association for Computing Machinery, New York, NY, USA, 13–20. https://doi.org/10.1145/1324302.1324306

[22] Janet Rountree and Nathan Rountree. 2009. Issues Regarding Threshold Concepts in Computer Science. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95* (Wellington, New Zealand) *(ACE '09)*. Australian Computer Society, Inc., AUS, 139–146.

[23] Kate Sanders and Robert McCartney. 2016. Threshold Concepts in Computing: Past, Present, and Future. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '16)*. Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/2999541.2999546

[24] Dermot Shinners-Kennedy. 2012. *Threshold concepts and teaching programming.* Ph. D. Dissertation. University of Kent. https://kar.kent.ac.uk/86522/

[25] Beth Simon and Quintin Cutts. 2012. Peer Instruction: A Teaching Method to Foster Deep Understanding. *Commun. ACM* 55, 2 (feb 2012), 27–29. https://doi.org/10.1145/2076450.2076459

[26] Beth Simon, Julian Parris, and Jaime Spacco. 2013. How We Teach Impacts Student Learning: Peer Instruction vs. Lecture in CS0. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 41–46. https://doi.org/10.1145/2445196.2445215

[27] Juha Sorva. 2010. Reflections on Threshold Concepts in Computer Programming and Beyond. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '10)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/1930464.1930467

[28] Juha Sorva. 2012. *Visual program simulation in introductory programming education.* Doctoral thesis. School of Science. http://urn.fi/URN:ISBN:978-952-60-4626-6

[29] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 2, Article 8 (jul 2013), 31 pages. https://doi.org/10.1145/2483710.2483713

[30] Lynda Thomas, Jonas Boustedt, Anna Eckerdal, Robert Mccartney, Jan Moström, Kate Sanders, and Carol Zander. 2010. *Threshold Concepts in Computer Science: An Ongoing Empirical Investigation.* 241–257. https://doi.org/10.1163/9789460912078_016

[31] Virginia M. Warfield. 2000. Calculus by scientific debate as an application of didactique. Retrieved January 21, 2022 from https://sites.math.washington.edu/~warfield/articles/Calc&Didactique.html

[32] Lucy Yeomans, Steffen Zschaler, and Kelly Coate. 2019. Transformative and Troublesome? Students' and Professional Programmers' Perspectives on Difficult Concepts in Programming. *ACM Trans. Comput. Educ.* 19, 3, Article 23 (jan 2019), 27 pages. https://doi.org/10.1145/3283071

[33] Bert Zwaneveld, Jacob Perrenet, and Roel Bloo. 2016. *Discussion of Methods for Threshold Research and an Application in Computer Science.* 269–284. https://doi.org/10.1007/978-94-6300-512-8_20