Abstraction and decomposition in tinkering tasks in visual programming environments

Andreas Eckert, Anna Sjödahl

Abstraction and decomposition in tinkering tasks in visual programming environments

Andreas Eckert[1] and Anna Sjödahl[2]

[1]Örebro University, Faculty of Science and Technology, Örebro, Sweden; andreas.eckert@oru.se

[2]Örebro University, Faculty of Science and Technology, Örebro, Sweden; anna.sjodahl@oru.se

This study examines how a tinkering task in visual programming environments can provide opportunities for developing problem solving skills. We pursue this by analyzing eleven students in year 8 working with a tinkering task in a visual programming environment during a mathematics class. They were asked to create repeating patterns. Their work and discussions were analyzed through a lens of abstraction and decomposition, two elements of computational thinking. The analysis reveals how some students became thoroughly engaged in problem solving while others had a shallower experience of randomly manipulating the pre-made code. Since it was not the difficulty of the task but rather the random outcome of their manipulation of the code that determined whether they became engaged or not suggests that there is a need for support structures to fully tap into the potential of tinkering tasks to elicit problem solving.

Keywords: Problem solving, tinkering, computational thinking.

Introduction

Coding is now, by the year 2020, incorporated in elementary school curricula in many countries around the world. Sweden, amongst others, has incorporated it into the mathematics curriculum and transforming the curriculum into an integrated science, technology, engineering and mathematics (STEM) curriculum in the process. Feurzeig et al. (2011) envisioned a school system where coding could be viewed as a means toward learning mathematics and developing problem solving skills. This cross-section of coding and mathematics situates coding in a STEM-context in Sweden. Coding has since then become framed within the bigger idea of computational thinking (CT), problem-solving by drawing on the concepts fundamental to computer science (Wing, 2006). The educational research literature does not adhere to one conceptualization of CT, nor how one can operationalize CT in the classroom practice for young pupils (Brennan & Resnick, 2012).

Problem solving is highlighted in both mathematics education research and the classroom practice as an important skill and a means for developing understanding (Hiebert & Grouws, 2007). It can be viewed as moving from the world we live in, to the world of symbols, that can be shaped and manipulated (Freudenthal, 1991). Problems are interpreted in the language of mathematics, creating models that has potential to say something about the real world. CT implies that this can be done by applying technical solutions through a process of breaking down the problem into decontextualized subproblems (abstracting and decomposing) which are interpreted in the language of digital technology (code). There are several similarities between problem solving skills in mathematics and programming, and a potential for mutual benefits in terms of learning (Feurzeig et al., 2011). Bråting and Kilhamn (2021) discuss this by pointing out intersections between algebraic thinking and CT with tasks for young children, e.g., that both contexts use variables, but also points out that they are

not necessarily understood in the same ways in both contexts and therefor introduce a potential conflict for learners.

Programming has a steep learning curve as it entails learning a new language to express oneself in. Kotsopoulos et al. (2017) suggests tasks where students modify pre-made code to make the learning experience more accessible and there are also visual programming environments (VPE) where code is represented as building blocks (Figure 4). A recent review of research on CT in the VPE Scratch[1] concluded that there is need for studies that make sense of the quality of understanding that students develop while programming (Fagerlund et al., 2021). However, problem solving that requires creative reasoning is hard and requires careful design of tasks and instruction (Jäder, 2019). There is a need to better understand design-elements of tasks to design instruction to engage primary schools' youngest students in tasks with potential to develop their CT. Our guiding question is *how can tinkering tasks in visual programming environments provide opportunities for developing problem solving skills?*

We will pursue the question by analyzing students' CT in a tinkering task which was a part of a design research project in mathematics education. We will use a lens of abstraction and decomposition to analyze students' actions in a VPE and discuss opportunities to develop CT from a STEM perspective.

Abstraction and decomposition

Solving a problem is a process of analyzing the problem, devising a plan and carrying out the plan (Polya, 2004). A way to approach this process is to make the problem more accessible (ibid.), e.g. by drawing a picture, find an analogous or auxiliary problem or break down the problem into more manageable subproblems and later recombine the solutions to a solution for the original problem. Burke (2012) describe similar processes in how middle school students can be taught to solve programming problems. The whole concept of CT can be thought of as "thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms" (Aho, 2012, p. 832). Kalelioglu et al. (2016) propose a framework for CT based on a systematic literature review that starts in Abstraction and Decomposition. Abstracting is the ability to focus what is essential and ignoring what is not (Wing, 2008) and thereby making the problem more accessible. What is then abstraction when speaking about young students' ability to abstract? Let us imagine a student that want to include a fish swimming from point A to point B in a VPE project. The student realize that the code needs to include the movement of the fish and the direction of the movement, but it doesn't really matter if the fish is green or purple. The problem to solve is how to get from point A to point B. The student must manage to see what is important and what is not (abstracting), to be able to solve the problem. Decomposing into

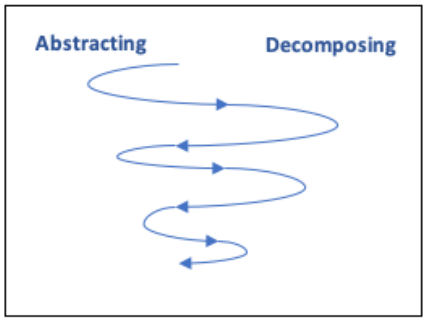

Figure 1. A spiral representing how we conceptualize the computational thinking processes abstraction and decomposing

[1] Scratch is a visual block-based programming environment developed by MIT. It is developed to be suitable for introduction to programming for young students since the pre-made blocks of code lowers the threshold of learning syntax to be able to code.

subproblems is the ability to solve problems by breaking them down into smaller more manageable problems. The student with the swimming fish needs to break this problem down into smaller pieces in order to solve it. One part of the problem is to make the fish move just far enough to get from point A to point B. The other part of the problem is to figure out which direction will get it to the right spot. Two subproblems to solve one problem. Whether one views these two processes as one, e.g. the CT practice (Brennan & Resnick, 2012), or as two, e.g. two tools when attacking large and complex tasks (Wing, 2008), they are closely intertwined. We choose to visualize this in a downward spiral, see figure 1, as we envision it having a direction towards the solution of the problem.

Solving a mathematical problem involves analyzing the problem, devising a plan and carrying out the plan (Polya, 2004). We connect this to our visualization of CT in three practices, visualizing an idea, formulating a plan, and translating into code, see figure 2. Steps of abstracting and decomposing moves the student forward towards a solution of the problem within a computer environment. The student with the fish starts in an idea, wanting to create a scene from the ocean. Acts of abstraction enables the student to figure out what parts of the idea are essential to create the impression of an ocean and every essential part has to be decomposed into solvable sub-problems to devise a plan. This plan, as an intermediary step towards writing an instruction that a computer could understand, is sometimes called a pseudo code (Weintrop et al., 2016). The plan is then carried out within the syntax of the coding environment, i.e., code-blocks in the VPE case.

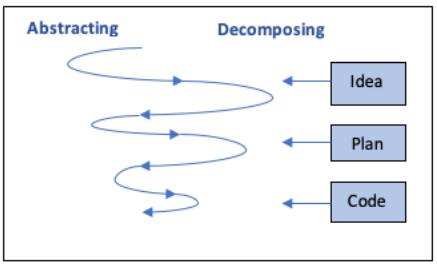

Figure 2. The computational thinking spiral with three stages of problem solving written in, Idea, Plan and Code

This spiral movement visualize the student getting deeper into the problem-solving process, working towards solving subproblems that eventually adds up to a complete solution. The student shift back and forth between the practices, modifying the *idea*, formulating and revising a *plan*, and produce *code*. The student in the example had an idea to create an ocean. The ocean was abstracted into a few essential parts that would make the viewer understand it is an ocean, for example a couple of moving fishes. Those essential parts constitute the plan. As the student start to deal with a problem one at the time, abstracting and decomposing in an iterative process, the processes lead to coding.

The type of subproblems that students engage in depends on the type of task (Eckert & Hjelte, 2021) and whether the subproblem constitutes a problem or not depends on the student. We adhere to the definition of a problem as when the solution method is not known in advance to the learner but require creative reasoning (Lithner, 2008). Tasks where students program for motion, typical to VPEs, tend to put programming subproblems in the foreground and mathematical subproblems in the background (Eckert & Hjelte, 2021). Bråting and Kilhamn (2021) question whether VPEs developed for the purpose of developing students' CT is the future of mathematics education. Even though there are similarities between algebraic thinking and CT, objects such as variables exists in both worlds, how we make sense of those objects differs. Benton et al. (2017) provide evidence for how carefully designed tasks have potential to develop both algebraic thinking and CT, e.g. the idea of the 360° total turn. The risk is though that one or the other gets lost in the process of creating code (Bråting & Kilhamn, 2021; Eckert & Hjelte, 2021).

Method

The present paper focuses on reaching an understanding of how tinkering tasks in VPEs provide opportunities for developing CT. The analytical process of this paper is that of an abductive approach (Alvesson & Sköldberg, 2009). We propose an analytical lens to view the data through, and then allow for the emerging insights into the students' CT process to influence the initial theorization. The idea of the CT spiral is meant to inspire cycles of future design experiment (Cobb, Confrey, diSessa, Lehrer, & Schauble, 2003) on programming in mathematics classes in the first years of primary school. The CT spiral is used here as an analytical framework in a retrospective analysis using constant comparisons (Gravemeijer & Cobb, 2013). Instances of data are compared to find similarities and differences related to abstraction and decomposition in the data. Looking for indications for when students relabel an object by focusing on certain aspects (abstraction) and when students try solving a subproblem by focusing on one aspect of the object (decomposition).

The data set used in this analysis are video screen captures of student work in the VPE and recordings of their verbal communication, generated within a larger design project concerned with digitally enhanced mathematics education. The tasks utilize the idea of engaging students in actual programming tasks and asking them to reason about the results. The design hypothesis was that the tasks would trigger students to develop their ability to solve mathematical problems using computers. The empirical study was performed with a class of 20 students in year 8 (aged 13/14) in Sweden using a tinkering task. For this particular task, eleven out of those 20 students' screens and discussions were captured through a screen capturing tool for analysis. The students were given a pre-made code in scratch (https://scratch.mit.edu/projects/250218077/) that drew a shape we called a snowflake. Figure 3 and 4 depict the code and result. The inner loop, that repeats twice, contain moves and turns that create the shape of each point of the snowflake. The outer loop, that repeats 8 times, creates the number of points and the angle between them. The task was to create their own repeating pattern based on the pre-made code. This type of task is called a tinkering task (Kotsopoulos et al., 2017) and is an established task design as students immediately are confronted with the intended problem-solving process. Students are meant to make simple modifications to an existing computer program to elicit questions such as "What if I change this part of the code?". Visual programming environments are especially suited for such tinkering because of its low threshold regarding programming syntax (Resnick et al., 2009) and its focus on exploration (Kotsopoulos et al., 2017). The result section in this paper includes a summary of the analysis of all eleven students as well as an in-depth analysis of one student's work. Transcripts are translated from Swedish to English by the authors. Names used are fictional.

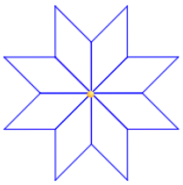

Figure 3. The snowflake

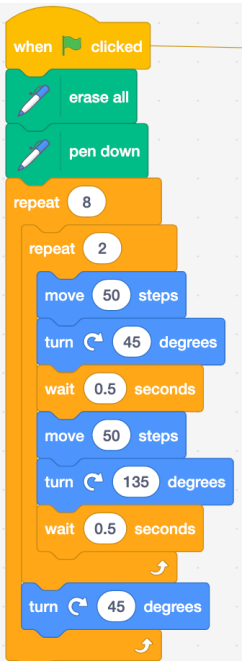

Figure 4. The tinkering task's pre-made code

Results

Our analysis indicates that tinkering tasks in visual programming environments provide both opportunities and challenges for developing CT in terms of abstraction and decomposition.

Reviewing all eleven students showed that in cases where we could detect a formulation of both an idea and a plan in the students' work, the task managed to engage the students in CT during the allotted time. In cases where either an idea, plan or both were missing, there were few indications of CT. The task type seemed to encourage CT in cases where random changes in the code produced new unfinished patterns that interested the students. In cases where the random changes either produced a finished pattern, or a pattern that was not pleasing to the student, the process came to a stop and did not elicit further CT. Following is Nevin's work, who together with Jasper produce interesting unfinished patterns in their random changes that they pursue and try to complete. Their random changes of the turn-blocks of the original code (see figure 3) resulted in a open snowflake shape (see figure 5). Their actions thereafter indicate that they have an idea what they want to accomplish. The idea seems to be to create a closed version of the open new snowflake shape.

Nevin: It has more repetitions.
Jasper: You need more repetitions.
Nevin: I know.
Jasper: 12, pick 12.
Nevin: Do you think 12 is enough.
Jasper: I think 12.

Nevin and Jasper analyze their snowflake shape and we interpret Nevin's utterance It has more repetitions as him identifying that the new shape has more repeating shapes than the original snowflake. We interpret Jasper's call for more repetitions as abstracting the notion of points and decompose their idea into the subproblem of filling the gap in the shape with more points. Nevin agrees that this is their plan. Together with their actions on the screen when they place the marker in the outer loop, the plan of adding more points to the snowflake is abstracted into being about repeating particular chunks of code more times, and they settle on twelve. Twelve repetitions are however too much so they try with ten. The change to ten repetitions does not close the shape completely, and this exchange follows.

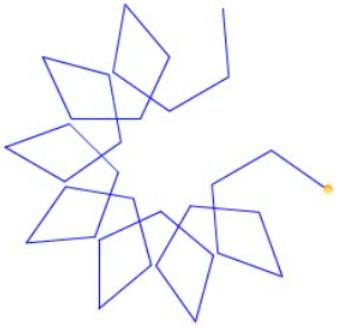

Figure 5. An open snowflake shape that requires more points to close

Jasper: There is one more needed.
Nevin: No … we can try but I don't think so.
Jasper: But, like this. It is one line that is not used. This line. Get it Nevin.
Nevin: They must connect at the end.
Jasper: Yes.
Nevin: But it isn't possible.

Jasper is set on their original plan of adding repetitions to close the shape. However, Nevin has spotted that even if they do so, the lines will not line up perfectly. To make the lines connect after one lap, the rotation-block in the outer loop needs to be a devisor to 360. Nevin seems to have spotted this visually from the picture and insinuate that they need to revise the plan.

Jasper: Maybe have to … do more steps. No, nothing will happen. Maybe more degrees or something.
Nevin: No, I don't think I will change anything.
Jasper: But I think so because 65 65 65 does not add up to 360. That's why.
Nevin: What is it then? How should I change it then?

To revise the plan, Jasper searches for aspects of the idea of a closed snowflake shape to abstract. He rejects the aspect of size and instead abstract the aspect of angles between each point of the snowflake.

Nevin does not agree but Jasper has decomposed the aspect to a subproblem of divisibility. In a previous task with simpler geometrical shapes, the key feature was to make the turn-blocks add up to 360. Their new plan is to use angles that are divisors to 360. Jasper and Nevin start by changing all the turn-blocks to 120 with the argument three times 120 is 360 but change their minds.

Jasper:	120 … you have to split it.
Nevin:	Yes. [whispers: I thought] … but what does 120 become then … 60 60 60. Otherwise it doesn't add up to 120. Does it add up to 120.
Jasper:	No
Nevin:	No. If I do 60. No I need to have the same or.
Jasper:	Yes but this must be higher.
Nevin:	Why.
Jasper:	It must be. What must it be … it must be … 90 it will be then. Wait. It was 45 from the start. Then that one was 135. So I think you need to have it to 180. Yes try 180.
Nevin:	How about this one?
Jasper:	It should be 60.

We see how the CT process elicited by the tinkering task has engaged the students in a mathematical reasoning. Their plan is to use divisors to 360, i.e. 60, 120 and 180 degrees. However, their understanding of the code is lacking which shows when the attempt to abstract elements of the plan into code. Based on the original code, they use 120 and 60 degrees in the inner loop since it adds up to 180 just as in the original code. The turn-block in the outer loop is set to 60 degrees. The end result is a shape very similar to the original snowflake, but at least it is closed which was in line with their idea and the time for the task runs out.

Analyzing the eleven students resulted in three different categories of actions, 1) students who maintained a CT process throughout the activity, similar to that above. Six students ended up in this category. In the other two categories did students not engage in a CT process, but those are difficult to showcase with transcripts. However, they both had in common that the task did not elicit a formulation of either an idea or a plan or both but their reasons why were different. 2) Students who, by mere chance, generated what was deemed by the student as an acceptable repeating pattern never got the chance to formulate an idea to continue working with. Two students were in this category and figure 6 shows an example of how such shape looked like. The engagement with the task stopped when the student achieved this shape from the first manipulation of the code. 3) Students who generated a shape that had potential to start a CT process but did not take time to abstract what were key aspects of their idea of the emerging shape that could be decomposed into manageable subproblems. Those did not formulate plans. Working without plans resulted in aimless manipulation of the code which either made the students give up or resulted in completely different shapes and a reformulation of the idea. Three of the analyzed students were in this category.

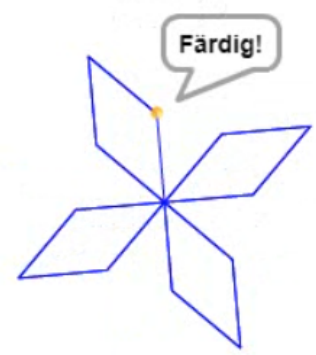

Figure 6. A closed shape produced on a student's first try

Discussion

Tinkering tasks in visual programming environments provide both opportunities and challenges for developing CT in terms of abstraction and decomposition. On the one hand, the pre-made code provided limited freedom to create a clear idea of what to achieve and the accessibility encourage a

planless revising of code did not elicit abstraction and decomposition. On the other hand, the low threshold of working with a pre-made code provided opportunities for some students to quickly engage in solving subproblems of both programming reasoning and mathematical reasoning. Such opportunities for integrated STEM learning merits further discussion.

Tinkering as a pedagogical activity in a VPE can be summarized as exploration (Kotsopoulos et al., 2017). It aligns with our findings, that students explore different randomly generated shapes. However, there were big differences in their engagement in problem solving. Students who immediately arrived at a shape they were happy with, hindered them from formulating problems which solutions could be manifested in code, essential to CT processes (Aho, 2012). Students who got stuck in unreflected exploration, who never formulated a plan connected to an idea, were not supported enough to decompose their idea into more manageable subproblems. Only students who were lucky enough to arrive at acceptably unfinished shapes were engaged in abstraction and decomposition. Those 6 students in the analysis created subproblems related both to coding and mathematics and thus had similar experiences as those in the Benton et al. (2017) study. The remaining 5 students never engaged in problem solving or mathematical reasoning and thus had the lesser experience pointed out by Bråting and Kilhamn (2021).

This tinkering task's advantage and disadvantage was the low threshold to coding. It was not the difficulty or novelty of the task that usually elicit creative reasoning (Lithner, 2008) but rather the random outcome of the low threshold exploration. The task itself seemed to fulfil the criteria of not having a known solution method and was engaging to the students. It leads us to believe that it is not the task itself that needs the careful designing to elicit creative reasoning proposed by Jäder (2019) in similar settings, or the positioning of mathematics as proposed by Eckert and Hjelte (2021) but rather support strategies that can help those students that for one reason or the other never engages in the problem-solving process. There are more aspects of STEM related problem solving activities in the cross section between coding and mathematics to explore. We invite researchers to use our model of analyzing a CT process to evaluate how support structures connected to tinkering tasks in VPEs can elicit STEM related problem solving experiences to a higher degree in the classroom.

References

Aho, A. V. (2012). Computation and computational thinking. *The computer journal, 55*(7), 832–835.

Benton, L., Hoyles, C., Kalas, I., & Noss, R. (2017). Bridging primary programming and mathematics: Some findings of design research in England. *Digital Experiences in Mathematics Education, 3*(2), 115–138.

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. *Proceedings of the 2012 annual meeting of the American educational research association*, Vancouver, Canada,

Bråting, K., & Kilhamn, C. (2021). Exploring the intersection of algebraic and computational thinking. *Mathematical Thinking and Learning, 23*(2), 170–185.

Burke, Q. (2012). The markings of a new pencil: Introducing programming-as-writing in the middle school classroom. *Journal of Media Literacy Education, 4*(2), 121–135.

Eckert, A., & Hjelte, A. (2021). Positioning of programming in mathematics classrooms–a literature review of evidence based didactical configurations. *Sustainable mathematics education in a digitalized world. Proceedings of MADIF12. The twelfth research seminar of the Swedish Society for Research in Mathematics Education*, Växjö, Sweden.

Fagerlund, J., Häkkinen, P., Vesisenaho, M., & Viiri, J. (2021). Computational thinking in programming with scratch in primary schools: A systematic review. *Computer Applications in Engineering Education, 29*(1), 12–28.

Feurzeig, W., Papert, S. A., & Lawler, B. (2011). Programming-languages as a conceptual framework for teaching mathematics. *Interactive Learning Environments, 19*(5), 487–501.

Freudenthal, H. (1991). *Revisiting mathematics education : China lectures.* Dordrecht: Kluwer Academic.

Hiebert, J., & Grouws, D. (2007). The effects of classroom mathematics teaching on students learning. In F. Lester (Ed.), *Second handbook of research on mathematics teaching and learning: A project of the national council of teachers of mathematics* (pp. 371–404). Charlotte: Information Age Pub.

Jäder, J. (2019). Task design with a focus on conceptual and creative challenges. *Eleventh Congress of the European Society for Research in Mathematics Education (CERME11)*, February 6–10, 2019, Utrecht, Netherlands.

Kalelioglu, F., Gulbahar, Y., & Kukul, V. (2016). A framework for computational thinking based on a systematic research review. *Baltic Journal of Modern Computing, 4*(3), 583–596.

Kotsopoulos, D., Floyd, L., Khan, S., Namukasa, I. K., Somanath, S., Weber, J., & Yiu, C. (2017). A Pedagogical Framework for Computational Thinking. *Digital Experiences in Mathematics Education, 3*(2), 154–171. https://doi.org/10.1007/s40751-017-0031-2

Lithner, J. (2008). A research framework for creative and imitative reasoning. *Educational Studies in Mathematics, 67*(3), 255–276.

Polya, G. (2004). *How to solve it: A new aspect of mathematical method*. Princeton: Princeton university press.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., & Silverman, B. (2009). Scratch: programming for all. *Communications of the ACM, 52*(11), 60–67.

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology, 25*(1), 127–147.

Wing, J. (2006). Computational thinking. *Communication of the ACM, 49*(3), 33–35.

Wing, J. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 366*(1881), 3717–3725.