Explicit Direct Instruction in Programming Education

Felienne Hermans and Marileen Smit
Delft University of Technology
{f.f.j.hermans, m.i.e.smit}@tudelft.nl

Abstract

In education, there is and has always been debate about how to teach. One of these debates centers around the role of the teacher: should their role be minimal, allowing students to find and classify knowledge independently, or should the teacher be in charge of what happens in the classroom, explaining students all they need to know? These forms of teaching are also called exploratory learning and direct instruction respectively. While the debate is not settled, more and more evidence is presented by researchers that explicit direct instruction is more effective than exploratory learning in teaching mathematics and science. These findings raise the question whether that might be true for programming education too. This is especially of interest since programming education is deeply rooted in the constructionist philosophy, leading many programmers to follow exploratory learning methods, often without being aware of it. This paper outlines this history of programming education and additional beliefs in programming that lead to the prevalence of exploratory forms of teaching. We subsequently explain the didactic principles of direct instruction, explore them in the context of programming, and hypothesize how it might look like for programming.

1. Introduction

Programming education is in fashion: in many countries around the world, programming is mandatory now in the UK, Australia and the US. This is of course, wonderful news for the lovers of programming! In the future, more children will be able to use their power to make computers do their bidding. However, programming is not an easy skill to learn, as evidenced, for example, by high dropout rates in introductory programming courses. While we do not have any numbers on the success rates of courses aimed at younger children, it is likely that there too learners are struggling.

In this paper we examine the form of teaching programming. In other fields like mathematics and language, recent results show that direct instruction is more effective, and benefits weaker students more.

In this paper we examine what programming believes about teaching. What are our didactic and philosophical beliefs and where do they come from. We present a brief history of the didactic beliefs of programming education, and explain why so far they have mainly leaned towards exploratory learning.

The paper also presents explicit direct instruction, since many programmers are not aware of it and might be hesitant for that reason. We close the paper with a detailed look into how we currently implement didactic principles of direct instruction and how we could benefit from its insights.

2. Constructionism and programming

When we program and when we teach programming, we also bring our beliefs about programming into the classroom. Contrary to educational scientists, psychologist and the like however, computer scientists are often unaware of their didactic beliefs, since we are not used to discussing those. There is however one didactic philosophy that many programmers believe in, and that is constructionism. In this section we explain why that is and what the implications of this are.

2.1. LOGO and its constructionist history

Programming education aimed at young children has a long history, starting in the sixties most notable by Seymour Papert and his work on the LOGO programming language. While many programmers are aware of LOGO, and have even learned to program in it, fewer people are informed about the didactic and philosophical ideas behind LOGO. Papert was a multidisciplinary research with a BSc in philosophy and and a PhD in mathematics, and he worked with a variety of researchers.

One of them was Jean Piaget, one of the most well-known psychologist of the last century, who pioneered the idea of *constructivism* (which is not constructionism, we will come to that in a bit!). The basic, simplified idea of constructivism is that children build their own view of the world, among other activities through play. Earlier researchers has seen playful exploration of children as unrelated to learning, but Piaget argued that children *assimilate* knowledge and then change their mental models of the world in a process called *accommodation*. Piaget aimed to describe how children built knowledge, nit not how to teach. This is where Papert comes in. He worked with Piaget for years and Piaget once said that Papers was the one "who understood his ideas best". Papers coined the idea of constructionism, which is a didactic method based on constructivism. Constructionism advocates students exploring the world around them, in a process often called *discovery learning* Students use their existing knowledge to acquire more knowledge via assimilation and accomodation as Piaget described those. The role of the teacher is more of a coach than a lecturer providing step-by-step guidance. Students often engage in project-based learning, making connections between different ideas and scientific fields rather than relying on predefined courses and boundaries of fields (Alesandrini & Larson, 2002). From this philosophy also LOGO was born, in his book Mindstorms (Papert, 1980), Papert describes the great ease with which young children learn French if they live in France. With LOGO, Papers wanted to create a *mathland* in which it is similar easy to learn mathematics and programming. While we do not often discuss this didactic basics of programming education, many programmers do belief this philosophy, thinking that if we just let children program, they will learn how a programming language works with ease. And while modern programming languages for children, for example Scratch, were not directly created by Papert, it was created by the same research group at MIT, and once you know about the constructionist beliefs Papert used to design LOGO, you see the same in the Scratch interface and the accompanying books and lessons. They too emphasize trying and exploring over detailed explanations of how things work in detail.

2.2. No collective memory of programming lessons

A second reason that there is massive buy-in into the constructionist mindset is a different one. Most professional programmers today are in the thirties and forties. That means that, when they learned to program, there were no programming courses, or websites. Most programmers taught themselves programming to a certain extent, and having taught oneself is a source of great pride for many programmers. Some job ads even specifically search for new employees that have been "programming since they were 12". As such, the programming community really lacks a collective memory of how one would give programming lessons. In many cases people that teach programming are programmers themselves, and thus they repeat the childhood experience of those teaching, resulting in very exploratory lessons, or, as was common in the eighties, by giving children large programs to copy, leaving the exploration up to the children.

And while it is great that there are children that can teach themselves, it does not mean that all children can learn programming that way. Like there certainly are some children that teach themselves to write or to bike, the large majority of children need some amount of help in learning various skills.

2.3. Programming skill is innate

In addition to the above, there are additional beliefs that fit the constructionist way of thinking. One of these is the widespread belief that not everyone is born for programming. Many people belief that programming is a skill that is not for everyone. While this seems too obvious to mention, it differs from general beliefs about reading and writing. Almost everyone believes that everyone with normal intelligence can learn how to read and write, even though of course not everyone is the next J.K. Rowling. This already differs from general beliefs about mathematics, where more people think that it is not 'for everyone'. Programming here is more aligned with mathematics. This belief also ties into constructionist views on education, because if some people are born with the programming genes, they can teach themselves in an exploratory fashion.

A closely related belief is the one that people that love programming, love the act and technology programming, rather than loving what programming can do for them. Again the comparison with language and mathematics help us place those beliefs. We learn to read and write for what it allows us to do, write love letters and petitions and dissertations and stories. Teachers stress this role of writing by having children read and write a variety of texts. With mathematics, of course it is also stressed what we can do with it, but, fewer people accept this, and the belied that there are people that love mathematics for mathematics is way more prominent than in language. Here too programming aligns with mathematics. This too aligns with an exploratory learning. If there are people that are born for programming and that love it naturally, we should not have a boring instructor stand in the way of their process!

This, of course, is a matter of 'chicken and egg'. It is hard to know whether people believe this because they were implicitly taught about constructionism and thus believe or because the constructionist beliefs fit will into our community because of these prior beliefs.

2.4. Implications

Because of the constructionist tradition, the lack of a memory of programming lessons and the fact that people belief that some people are naturally drawn to programming leads to the fact that many programming lessons are what we call *implicitly constructionist*. Rather than being designed consciously constructed at exploratory teaching—which can surely have value and is regarded as a superior form of teaching by some, and as equally good as explicit direct instruction by others—people teach in a constructionist fashion by accident, without being aware of its strengths and weaknesses, and without considering alternatives. The role of this paper is not to reject constructionist teaching as a form of teaching, but it is to present direct instruction as precisely such an alternative. Similar to other fields where people argue about these core philosophies, this will happen in programming too, and that is fine. As such, we will present explicit direct instruction, summarize research that has found it beneficial and hypothesize how it could look like for programming.

3. Explicit direct instruction

Like with all teaching methods, are different forms of direct instruction and teachers follow methods with varying levels of strictness. The basic idea of direct instruction is that it is teacher-led, the teacher determines what happens and in what order, and directs information at students, hence the name directed. The lessons are structured and sequenced and have clear learning goals. One of the forms, which is often criticized is Direct Instruction with capitals (DI) as pioneers by Engelmann and Becker (Engelmann, Becker, Carnine, & Gersten, 1988). DI prescribes, among other things, quite rigid teacher scripts and that is probably where the bad name of direct instruction stems from. Scripts read aloud by robot-like teachers, that cannot be right! However more modern forms of direct instruction comprise a lot more than plenary lectures. Most notably Explicit Direct Instruction (EDI) as described by Hollingsworth and Ybara (Hollingsworth & Ybarra, 2009) presents eight didactic principles that strengthen each other: They are clearly expressing the **learning objectives**, **activating prior knowledge developing skills** and **developing concepts**, explaining the **importance of a lesson**, and a sequence of **guided practice** with the teacher, practicing in groups at the **plenary closure**, followed by **independent practice**.

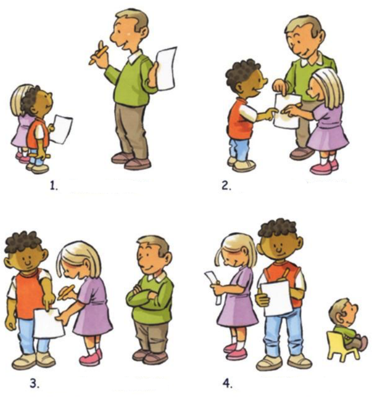


Figure 1 – The 4 steps of an EDI lesson: 1. The teacher explains, 2. the teacher demonstrates, 3. students try together, 4. students do it individually.

EDI lessons can be best summarized by Figure 1, where we see that an EDI lesson is more than the delivery of a canned script, it is a form of teaching that relies on metacognition of the teacher. They have to know what form of teaching they are using, why to use it and when to move to the next step (Hollingsworth & Ybarra, 2009). It is a form of teaching that activates students, and requires them to work together with the teacher in step 2 and together with peers in step 3. By no means does EDI mean that students sit back and listen.

3.1. Comparing direct instruction and exploratory learning

From its inception, proponents of direct instruction have presented its benefit in studies. For an overview of several early studies, we refer the interested reader to Binder and Watkins (Carl & L., n.d.).

The debate still continues, but also more recent recent metastudies have demonstrated direct instruction to be more effective than less structured forms of teaching such as those based on exploratory, discovery or constructionist approaches (Klahr & Nigam, 2004; Kirschner, Sweller, & Clark, 2006).

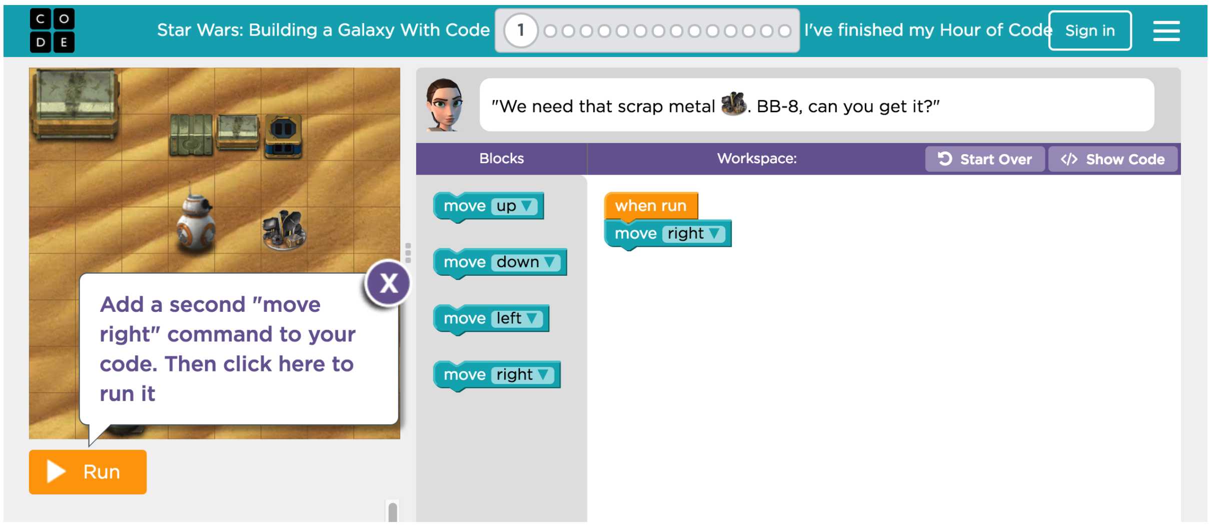
What is more is that direct instruction can be an equalizing from of teaching. Irrespective of prior knowledge, all children are exposed to the same instruction, and practice together, so all students have the same opportunity to gain knowledge and to practice. In exploratory teaching, the students that already have some prior knowledge will be beter able to ask the right questions, and to attach new facts to existing knowledge, or to update mental models. Studies have demonstrates the performance of weaker students within a class especially suffers when exploratory methods are used, while they are the students that enjoy this form of teaching the most!

4. Explicit direct instruction in programming

Knowing that direct instruction seems to be a superior and more equal form of teaching, the final question that this paper aims to answer is 'how would direct instruction for programming lessons look like?' When we explore the seven design principles of EDI, how would we implement those in programming classes?

4.1. How programming differs from other subjects

So far we have (implicitly) assumed that programming and other fields such as language or mathematics are the same, and that teaching methods from those fields like direct instruction will also work for programming. While we absolutely believe the latter, there are some notable ways in which programming differs from other subjects which we will touch upon in the remainder of this section. For starters, when we present a student with a traditional programming environment, they can use all elements of the language. They can try to explore language concepts that have not been covered in class and try this. While of course, students in mathematics can also ask about percentages with you as a teacher are talking about multiplication, in programming students can venture on a path of their own. This inhibits, to a certain extent, the level of control teachers have over the learning trajectory of students. Secondly, compilers and interpreters are relentless. When we are teaching grammar, we might disregard spelling errors for the duration of the lessons to focus on grammar (not all teachers agree they should so unnoticed, but most agree we can give errors outside of the scope of the current lesson or subject lower weight). When we are teaching loops however, and a student has forgotten to indent their code consistently, Python will not comply.



"We need that scrap metal . BB-8, can you get it?"

Add a second "move right" command to your code. Then click here to run it

Figure 2 – The first screen students see when they try the Star Wars Hour of Code lesson.

4.2. EDI design principles applied to programming

Taking the differences between programming and other subjects into consideration, we now inspect the seven EDI design principles and explore how they are currently implements in programming lessons, or how they could be.

4.2.1. Learning Objectives

EDI lessons start with a well-defined statement that explains what students will be able to be after the lesson, in words that they can understand. So something like “after this lesson you will know monads” is not useful when the students have not seen a monad before. Research demonstrates that the presence of learning objectives increased performance (Hattie, 2008). Examining popular programming lessons, we see that learning objective are missing in many cases. For example, consider the programming environment of code.org, of which the first lesson is pictured in Figure 2. It is arguable a quite structured and sequenced learning experience, where a learner is not exposed to all possible programming blocks, but only a small subset. Even this learning environment though has contextual goals “getting the scrap metal”, but does not explain the learning goal of the lesson.

4.2.2. Activating Prior Knowledge

By repeating existing knowledge in the beginning of a lesson, the teacher moves knowledge from the students’ long-term memory to their working memory such that it can be used to connect new knowledge too. Seeing the connection between different parts of programming, between concepts like a method and a class, but also, at a lower level, the difference between and if and and if-else is important, and in this respect we believe that programming education does relatively well. Concepts are presented together and in relation to each other commonly.

4.2.3. Concept Development

Concept development means explaining the concepts that are present in the learning objective. So if the objective is to learn monads, this part of the lesson concerns explaining what a monad is and when to use one. This part is commonly present in programming lessons, in some cases, especially at the university level, it is the only part used in the plenary part of teaching.

4.2.4. Skills Development

Skills development is teaching students the steps or processes that are used to teach the learning objective. It is an interesting aspect, because it raises the question what the skills are that one needs to master programming. One could argue that the most low level skills that children need are basic computer skills: using the mouse, clicking, using copy and paste and undo. In our experience teaching children programming, they rarely have those skills even in high school, and courses do not teach them. At higher levels, such as university, student might lack understanding of an operating system impeding their programming ability. A next step, closer to programming itself is the use of an IDE. That is surely a skill that helps students become more efficient in programming, and yet is not often addressed as topic of lessons or exercises. Looking at programming itself, there are skills (rather than concepts) that we also rarely teach, for example the correct use of syntax. In many programming lessons, as explained above, the concepts play a central role, while syntax is an afterthought. Students are assumed to pick up where colons go and when to use which brackets. Another example of a code skill in programming that we do not teach is reading and interpreting error messages.

At the highest level of skills, we could place strategies for programming. How is a problem approached and what steps does a professional programmer take when addressing a problem. We observe in programming lessons, that this too is something that is rarely vocalized or practiced.

We think skills development is the teaching practice where programming can benefit the most from direct instructional practices.

4.2.5. Lesson Importance

Lesson importance concerns explaining to students why the content of a lesson is important to them. While this might feel a bit boring, research shows that understanding the why of a lessons helps students engage (Hattie, 2008). Here too programming education could use some help. Where lesson importance is addressed, it is often within the scope of programming, for example "a loop is useful because it allows you to execute a computation multiple times". This practice is probably related to the fact that programming believes that children have an interest in programming for the sake of programming, rather than in what it can help them create. Communicating why concepts and skills matter outside of programming (not an easy task!) will help students engage with programming lessons more.

4.2.6. Guided Practice

Guided practice is solving a problem together with students, represented by step 2 in Figure **??**. In programming there is the practice of live coding, where a teacher or instructor programs in front of a classroom, but in many cases the plan for the live coding is made ahead of the lesson rather than collaboratively with the students. Also, problem solving strategies are not always discussed.

It is used in language education more commonly, where a teacher demonstrates, for example, how to write a story, while explaining their approach and asking students for input. A common technique for this is *observational learning*, where a teacher (or peers) demonstrates a task before learners attempt it. In writing education in fact, teacher modeling is the most prevailing way of using models for learning. Usually in the instructional phase (Koster & Bouwer, 2016). In this teaching method the teacher thinks out loud, they explain and demonstrate parts of the writing task. Pupils are expected to adopt the same line of reasoning when they will executing a writing task individually afterwards. It is shown an effective instructional method to teach writing strategies (see e.g. (Fidalgo, Torrance, Rijlaarsdam, van den Bergh, & Álvarez, 2015; Graham, Harris, & Mason, 2005; Koster & Bouwer, 2016)).

This effectiveness of this method is explained by the existence of the mirror neuron system in our brain. This system makes the brain demonstrate identical neural activity when we observe others performing a task as if we perform the task ourselves (see e.g. (Rizzolatti, 2005; Rizzolatti & Craighero, 2004)). In this way, the brain already 'learns' how to perform a task, and primes the execution of similar tasks.

This is definitely a didactic practice where programming could improve in a relatively easy way. From live coding it is not a large step towards making decisions and strategies explicit.

4.2.7. Lesson Closure

At the end of an EDI lesson, the teacher tests whether students master the demonstrated skills, of example with questions or by having them work on small, constraint problems. This has been explored for programming, for example in the form of worked examples (Gray, St. Clair, James, & Mead, 2007), but to say this is a practice that is common would be untrue. Most programming lessons both in schools and universities move towards independent practice soon after a plenary lecture.

It will not be all that easy however to start the practice of having students work on a small task in a collaborative fashion, since we are not used to teach in this way. Programming exercises, especially in the phase when students are still getting familiar with syntax, are most commonly performed alone. Also selecting or creating the right worked examples for the task at hand is not easy. While lots of improvement could be achieved by having students practice (and fail) in a more controlled environment, it will not be easy to convince teachers to do so. Here, the fact that students can use the entire programming language is a confounding factor. Maybe we need more programming systems where teachers can limit the possible statements and thus can exert more control, very much like code.org does it, but for text based languages, and controlled by the teacher.

4.2.8. Independent Practice

Independent practice is the part of a lesson where students work on larger programs independently. This part of learning is common in programming education, both at schools and at universities, but it is commonly the only part of practice that students encounter.

5. Concluding remarks

In this paper we have explored the constructionist history of programming, our lack of a shared memory of programming lessons, and related beliefs about programming, which we argue together have caused our field to teach in an exploratory way, without programmers being aware of this. We also summarize an alternative technique called explicit direct instruction, and present its benefits. We close the paper with a reflection on the current state of the art of programming through the lens of EDI and an outlook on how we could use more principles from direct instruction into our teaching.

6. References

Alesandrini, K., & Larson, L. (2002). Teachers bridge to constructivism. *The Clearing House: A Journal of Educational Strategies, Issues and Ideas*, *75*(3), 118-121. Retrieved from `https://doi.org/10.1080/00098650209599249` doi: 10.1080/00098650209599249

Carl, B., & L., W. C. (n.d.). Precision teaching and direct instruction: Measurably superior instructional technology in schools. *Performance Improvement Quarterly*, *3*(4), 74-96. Retrieved from `https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1937-8327.1990.tb00478.x` doi: 10.1111/j.1937-8327.1990.tb00478.x

Engelmann, S., Becker, W. C., Carnine, D., & Gersten, R. (1988). The direct instruction follow through model: Design and outcomes. *Education and Treatment of Children*, *11*(4), 303–317. Retrieved from `http://www.jstor.org/stable/42899079`

Fidalgo, R., Torrance, M., Rijlaarsdam, G., van den Bergh, H., & Álvarez, M. L. (2015). Strategy-focused writing instruction: Just observing and reflecting on a model benefits 6th grade students. *Contemporary Educational Psychology*, *41*, 37–50.

Graham, S., Harris, K. R., & Mason, L. (2005). Improving the writing performance, knowledge, and self-efficacy of struggling young writers: The effects of self-regulated strategy development. *Contemporary Educational Psychology*, *30*(2), 207–241.

Gray, S., St. Clair, C., James, R., & Mead, J. (2007). Suggestions for graduated exposure to programming concepts using fading worked examples. In *Proceedings of the third international workshop on computing education research* (pp. 99–110). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1288580.1288594` doi: 10.1145/1288580.1288594

Hattie, J. (2008). *Visible learning*. Routledge.

Hollingsworth, J., & Ybarra, S. (2009). *Explicit direct instruction (edi): The power of the well-crafted,* *well-taught lesson*. SAGE Publications. Retrieved from https://books.google.nl/books?id=OiamQ21R8h0C

Kirschner, P. A., Sweller, J., & Clark, R. E. (2006). Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist*, *41*(2), 75-86. Retrieved from https://doi.org/10.1207/s15326985ep4102_1 doi: 10.1207/s15326985ep4102_1

Klahr, D., & Nigam, M. (2004). The equivalence of learning paths in early science instruction: Effects of direct instruction and discovery learning. *Psychological Science*, *15*(10), 661-667. Retrieved from https://doi.org/10.1111/j.0956-7976.2004.00737.x (PMID: 15447636) doi: 10.1111/j.0956-7976.2004.00737.x

Koster, M., & Bouwer, I. (2016). *Bringing writing research into the classroom: The effectiveness of tekster, a newly developed writing program for elementary students* (Unpublished doctoral dissertation). ICO.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY, USA: Basic Books, Inc.

Rizzolatti, C. (2005). The mirror neuron system and imitation. *Perspectives on Imitation: Mechanisms of imitation and imitation in animals*, *1*, 55.

Rizzolatti, C., & Craighero, L. (2004). The mirror-neuron system. *Annual Review of Neuroscience*, *27*, 169-92.