Problem Solving and Algorithmic Development with Flowcharts

Smetsers-Weeda, R.; Smetsers, S.

2017, Article in monograph or in proceedings (Barendsen, E. (ed.), WiPSCE '17: Proceedings of the 12th Workshop on Primary and Secondary Computing Education, Nijmegen, Netherlands — November 08 - 10, 2017, pp. 25-34)

Doi link to publisher: https://doi.org/10.1145/3137065.3137080

Problem Solving and Algorithmic Development with Flowcharts

Renske Smetsers-Weeda
Radboud University
Nijmegen, The Netherlands
Renske.Smetsers@science.ru.nl

Sjaak Smetsers
Radboud University
Nijmegen, The Netherlands
S.Smetsers@cs.ru.nl

ABSTRACT

Programming, where problem solving and coding come together, is cognitively demanding. Whereas traditional instructional strategies tend to focus on language constructs, the problem solving skills required for programming remain underexposed.

In an explorative small-scale case study we explore a "thinking-first" framework combined with stepwise heuristics, to provide students structure throughout the entire programming process.

Using unplugged activities and high-level flowcharts, students are guided to brainstorm about possible solutions and plan their algorithms before diving into (and getting lost in) coding details. Thereafter, a stepwise approach is followed towards implementation. Flowcharts support novice programmers to keep track of where they are and give guidance to what they need to do next, similar to a road-map.

High-level flowcharts play a key role in this approach to problem solving. They facilitate planning, understanding and decomposing the problem, communicating ideas in an early stage, step-wise implementation and evaluating and reflecting on the solution (and approach) as a whole.

CCS CONCEPTS

• **Social and professional topics** → **Professional topics**; **Computing education**; *K-12 education*;

KEYWORDS

Flowcharts, Unplugged, Algorithmic Thinking, Problem Solving, Novice Programming, Think-then-Act, Plans, Algorithmic Development

1 INTRODUCTION

Novice programmers suffer from a wide range of difficulties and deficits [33]. They have to juggle with new programming language constructs, syntax, paradigms, tooling (compilers and debugging) and problem solving strategies simultaneously [7]. Programming, where problem solving and coding come together, is cognitively demanding. With most traditional introductory programming courses focussing on teaching how to code, the coupled problem-solving skills remain underexposed.

In particular, novice programmers lack guidance in the problem solving process. Problem solving requires careful reasoning about how to get things done, involving abstraction, decomposition, generalization, evaluation and planning and ordering. From a heuristics point of view, they tend to skip steps in the process [27]. Novice programmers spend little time thinking-first: brainstorming, tinkering and planning [33]. As a result, when writing code, students dive straight into details and seem to lose sight of the big picture [7].

Understanding how programming related problem-solving skills can be enforced might bring educators a step closer to helping learners become more effective and efficient computational problem solvers.

Sources of programming difficulties

Sources of programming difficulties (or 'misconceptions') which novice programmers encounter have been extensively researched and documented. In their meta-analysis, Sheard et al. [35] investigated 164 relevant full refereed papers about programming education. Rountree gives a general account of misconceptions [33]. Others focus on specific concepts such as variables [24], loops [11], boolean conditions and control structures [1, 20]. Spohrer and Soloway [38] analyze difficulties related to *plan composition* problems, and argue that these have even more impact on programming errors than construct-based misconceptions.

Problem solving difficulties

Computer science learners often experience difficulties during problem analysis, planning and design of solutions [19]. Kirschner, Sweller and Clark [22] argue that learning a complex task like programming, not only requires conceptual knowledge, but also *explicit procedural guidance*. It is essential to understand which steps must be taken in order to solve a problem, as well as how to recognize an acceptable solution.

In their research, McCracken et al. [27] define 5 iterative steps to problem solving. These steps are a slight (almost textual) modification to Polya's steps to solving a mathematical problem [32]. The five steps, are:

(1) Abstract the problem from its description - Identify the relevant aspects from the problem description, then model the elements in an abstract framework.
(2) Generate sub-problems - Decompose the design (determine methods and sub-methods).

WiPSCE '17, November 8–10, 2017, Nijmegen, Netherlands

ACM ISBN 978-1-4503-5428-8/17/11…$15.00
https://doi.org/10.1145/3137065.3137080

(3) Transform sub-problems into sub-solutions - Determine an implementation strategy for individual classes, methods, appropriate language constructs as well as data structures and programming techniques.
(4) Re-compose the sub-solutions into a working program - Combine the sub-solutions from the previous step into a working program / complete solution. This step generally involves the creation of an algorithm that controls a sequence of events.
(5) Evaluate and iterate - Determine if the previous steps have lead to an adequate solution to the problem and take appropriate action if not.

McCracken's study shows that problems in programming assignments arise when students skip steps or execute them incorrectly.

Thinking first: unplugged

Unplugged activities can be used to introduce new topics, analyze problems and brainstorm about solutions, away from the distracting computer [3, 16]. Students use the skill of decomposition more successfully in situations where they understand the problem very well [34]. Using an unplugged approach and visualizing the solution prior to implementation assures that all students have made a start and are 'on-board', increasing growth-mindset [9] and the necessary self-efficacy to tackle the problem at hand.

Coding strategies: plans

Students need strategies to turn a conceptual design into a program [33, 38]. Becoming a competent programmer involves, among other things, establishing a collection of problem patterns together with their standard solutions [33].

Plans are frequently-used (expert) coding strategies to accomplish a task. Examples are finding the maximal value of several objects or the use of a counter controlled loop to repeat tasks.

Choosing and applying such solutions (or *plans*) results in pieces of a solution that must be integrated together in a correct algorithmic manner. In his research de Raadt [12, 13] describes how teaching programming strategies *explicitly* improves student learning outcomes [14], showing both an increase in the use of strategies and a significant improvement in the completeness of solutions.

The big picture: flowcharts

As the number of errors increases with the complexity of the problem [38], decomposition is the key to beating cognitive-load issues. Students suffer from cognitive overload when they receive too much information and are unable to process it [30].

Considering the taxonomy of educational objectives, decomposition is a prerequisite for abstraction, algorithm design, and evaluation [34]. Although students understand the concept of breaking-down a problem, learners struggle with the process of decomposition (McCracken's step 2 and 3): decomposition is perceived to be the most difficult programming skill to master [34].

Successful *plan composition* (McCracken's step 4) is strongly dependent on algorithmic thinking. Firstly, the correct plans must be selected [12] for use as a basis for the solution. Secondly, they must be correctly combined. Novice programmers spend little time planning [33]. Spohrer and Soloway [36] conclude from an analysis of 101 bugs that the majority are caused during plan abutment and merging [38]. As a result, when writing code, students dive straight into details and seem to lose sight of the big picture [7]. McCracken et al. [27] conclude a cause to be a lack of structure when tackling a programming problems.

Flowcharts enable the visualization, communication and evaluation of proposed solutions at an early stage, comparable to how (UML) designs are used as a high-level description to communicate with stakeholders and team members in the IT-industry. Students could benefit from some kind of similar guidance through the process of design, implementation and evaluation; see [6].

Feedback and evaluation

Success-experience and (concrete, specific and timely) feedback are essential factors for motivating students to exert their utmost to understand and learn [2]. Students that are actively participating in exploring ideas are more engaged (constructivism approach), leading to an increased learning potential.

An incomplete or incorrect mental model makes it difficult for students to envision and comprehend the behavior of programs [37], [7]. Inspired by Papert's Mindstorms philosophy (Papert, 1980), visualisation tools (such as Scratch [26], Blockly [29], Greenfoot [23], AppInventor [39] or Alice [8]) have been developed to allow students to analyze the effects of language constructs and algorithms, in addition to increasing motivation [7]. Novice students prefer a visual algorithm development environment to a textual programming language [5]. The visualization capabilities of these tools unquestionably have the potential to aid the understanding and concretizing of abstract concepts of computing science, particularly for novices [19] [35]. By linking algorithmic thinking to an implementation (programming), pupils obtain direct visual feedback on their algorithms.

The high-level concepts of logical thinking and design are the most difficult for students to develop, yet they receive little feedback. Syntax is perceived as less difficult but receives large amounts of feedback [4]. Being able to communicate an idea at an early stage allows for scaffolding as students discuss the task, monitor what they learn as they resolve conflicts, build on each other's ideas, and correct mistakes [17, 21].

The Guide

In a joint project with the Radboud University Nijmegen, new educational material has been developed based on a "thinking-first" pedagogy, with a primary focus on algorithms-first. The framework **guides** the students through the process of programming. Based on a stepwise heuristic it facilitates planning, understanding and decomposition of the problem, communicating ideas at an early stage, and evaluating and reflecting on the implemented solution as a whole.

To structure problem solving, we propose:

(1) flowcharts to guide students through the iterative steps of using problem solving heuristics (based on McCracken [27] and Polya [32]);
(2) application of 'standard' *plans* (based on the research of de Raadt [12]);
(3) unplugged activities to get students into the "thinking-mode" and tinkering about the problem and its possible solutions;

(4) coding the solution using a visually compelling environment (Greenfoot) in order to receive feedback and facilitate the evaluation of the solution.

The paper is organized as follows. In Section 2 we describe our pedagogical philosophy and the educational material used. Our research question is formulated in Section 3. The methodology used in this research is discussed in Section 4. Section 5 presents the results of the case study. In Section 6 we answer our research question, and discuss improvements for the material. Finally, we formulate the conclusions of the study and plans for future research in Section 7.

2 EDUCATIONAL MATERIAL

Pedagogy

The material development was originally inspired by Karel The Robot [31]. It is distinctive because it focusses on problem solving skills, rather than concept-first. Unplugged activities and devising flowcharts put algorithmic development in the spotlight. Students learn to think about algorithms away from the distracting computer and explicitly sketch their proposed solutions using flowcharts. Students develop, analyze, and evaluate solutions, prior to, during, and after implementation. Theoretical concepts are taught when required to solve the problem at hand. The programming language (Java) is used as a tool for implementing algorithmic solutions. The conceptual aspects of programming languages are thus taught as a vehicle for creating solutions, not as a primary goal on its own. The visually compelling programming environment Greenfoot [23] engages students to test and evaluate their (algorithmic) solutions. In each of these steps, flowcharts are the linking-pin, helping students make a plan and guiding them through the different steps of problem solving.

The main idea is that establishing an appropriate approach has far more potential of leading to a correct solution than an incorrect approach which is implemented faithfully. The latter will most likely introduce logical runtime errors. As these are far more difficult to locate and fix, than for example syntax errors, they are bound to have a negative effect on students' self-efficacy.

The time-frame is as follows:

week	main concepts
1	methods, results, flowcharts, algorithms, classes, inheritance
2	conditional expressions, correctness, structured problem solving approach, sentinel-controlled loop
3	problem decomposition, sub-methods, re-use, nesting, algorithmic evaluation
4	variables, tracing tables, operators, parameters, counter-controlled loops
5	plans, generic algorithms
6	lists, for-each-loop, sorting, Java API
7	solving complex problems (travelling salesman, minimal spanning tree)

Material time-line

The following time-line shows the material has been tailored throughout each iteration:

- **Objects-first approach:** During the first three runs we experienced that the material had potential. Students were learning, having fun and after instruction could complete the assignments rather autonomously. However, only moderate results were being achieved. Especially weak students got stuck on elementary OO-design principles, losing focus. As a result, the appropriate level algorithmic design wasn't being achieved.
- **Think-first approach:** OO-design was removed to make room for unplugged activities. During the following four runs we experienced that students, away from the distracting computer were able to compose surprisingly complex high-level algorithms (for example, travelling-salesman problem). The flowchart's underlying task decomposition gives way to modularized and working implementations. This version has been used by other teachers, including being taught as part of a CPD. In addition to our own observations, feedback from students and other teachers is concretely what encouraged this systematic research. We wish to determine if algorithmic design and evaluation is indeed being achieved at substantially high levels and what else can be done to further improve learning results.

Guidance and support

The primary goal of the educational material is to give support in the problem solving process by guiding them to work in a structured manner. This is done by uncoupling design (thinking about how to tackle a problem) from coding (mastering and applying conceptual knowledge), and in addition incorporating *plans* explicitly.

1. Abstract the problem from its description "First get those youngsters to think" [32]. The (distracting) computer seems to suck students into an immediate act-first-mode instead of a thinking-first-mode. We propose that when students are able to split the problem solving process from specific implementation details it becomes less cognitively demanding [38]. The key is to do this *away from the computer*.

During unplugged activities we focus on the development of CT skills such as algorithmic thinking, abstraction, and evaluation as well as on brainstorming for alternative solutions prior to implementation. Students become actively engaged and are challenged to think creatively about complex problems. They describe their solutions using high-level flowcharts as a manner of structuring and communicating their thoughts and solutions.

2. Generate sub-problems The flowcharts help students focus on the design, (temporarily) evading the nitty-gritty details of coding. Selecting appropriate *plans* (i.e. standard solutions) reduces details and makes the problem easier to think about. In this step, students focus on thinking procedurally (identifying and ordering (sub)steps), thinking logically (in terms of decisions and conditions), and thinking ahead (in terms of inputs and outputs). Preconditions and postconditions are formulated explicitly.

3. Transform sub-problems into sub-solutions With a high-level abstract sketch of the solution in their hands, each flowchart component (subproblem) is tackled independently through successive decomposition. Students implement their design using the appropriate language constructs, data structures and programming techniques. Greenfoot supports simple object instantiation and

method invocation, facilitating testing and evaluation of each sub-solution in isolation. If necessary, design or implementation adjustments are made.

4. Re-compose the sub-solutions into a working program As the flowchart specifies the algorithm for the complete solution, it is to be used as a roadmap, guiding students through the composition of the sub-solutions into a complete program. The pre and post conditions defined in step 2 can be used to reason about sub-solution abutment details.

5. Evaluate and iterate As the proposed solution (including initial and final states) has been described prior to implementation, it can be used to evaluate that the solution adheres to the requirements specified. If the solution is incorrect or incomplete, appropriate action is taken (iterate back to a previous step).

Course objectives

The main course objective is to able to create programming solutions which incorporate the following:

- Computational Thinking: decomposition, abstraction, generalization, algorithmic thinking and evaluation.
- Construct-based knowledge: variables (and types), (assignment, comparison and arithmetic) operators, methods (signatures, parameters and results), boolean statements, control structures (`if..then..else`, `while` and `for`-each), lists, and basic OO principles.
- Plans: triangular swap, initialization, primed sentinel loop and counter controlled loop, min/max, average and sorting.

Previous experiences

The material has been used and tested in a successful pilot with undergraduate non-computer science students in the spring of 2015 and a high school course in 2016. Successful is to say, in comparison with the previous year, the science students achieved higher academic results (similar tests and learning objectives), were more satisfied about the level and manner of education, and more enthusiastic about computer science as a whole. The year before, similar course material based on Greenfoot was used, however without the focus on problem solving skills and computational thinking which is now explicitly taught.

The material is freely available under a Creative Commons licence. It has been reviewed by several high school teachers. Together with its underlying pedagogy it formed the basis for two CPD courses at the Radboud University Nijmegen and the Delft University of Technology. In addition to receiving constructive feedback, the overall responses are both positive and enthusiastic.

3 AIM OF THE STUDY

We explore the use of flowcharts in combination with unplugged activities, instructing the use of 'plans' and Greenfoot's visualisation techniques as an approach to elevate students' programming skills to a higher level.

We investigate the educational implications of our findings and propose a set of improvements for both the material and the underlying pedagogy to be further researched in a follow-up study.

Research question

Which learning difficulties pertaining to novice programmers does our approach, with a strong focus on algorithmic design and thinking-first, overcome?

4 METHOD

In an explorative small-scale case study we investigated our pedagogical approach.

Research population

The research population consisted of high school students who chose the course as an elective masterclass. It was announced as an introductory course. Participants were not expected to have any prerequisite programming knowledge or experience. Any previous exposure to CS was minimal or self-taught. There was one class with 11 Dutch high school students, of which 2 females and 9 males. The group was heterogeneous, with students from different academic levels and age-groups. Each student's level and previous experience with CS was determined a priori using a pretest and an interview. The pretest (borrowed from [28], with some slight modifications) was at an abstract algorithmic level.

The class time comprised of one two-hour periods per week for 13 weeks. In addition, students were expected to complete at least one hour of homework each week.

The two teachers had ample knowledge about the educational material and its pedagogy. In addition to being the authors, they had previously used the material in a pilot in an undergraduate science course.

Data collection

Due to the participant group size our research is qualitative. To answer our research question, a variety of data was collected: individual formal written assessments, (homework) assignments, think-aloud sessions, semi-structured interviews and in-class observations.

Taxonomy

In their research, Meerbaum-Salant et. al. propose a hierarchical taxonomy scale describing how student's performance grows in complexity. It is based on a combination of the revised Bloom and Solo taxonomies [28]. Three super-categories (derived from the revised Bloom taxonomy) were chosen:

Understanding: The ability to summarize, explain, exemplify, classify, and compare CS concepts, including programming constructs.

Applying: The ability to execute programs or algorithms, to track them, and to recognize their goals.

Creating: The ability to plan and produce programs or algorithms (constructing, analysing, evaluating and formulating).

Each of these categories were further subdivided into three levels (derived from the Solo taxonomy):

Unistructural: The ability to create very small scripts doing one thing, or adding an instruction with a local effect to an existing script.

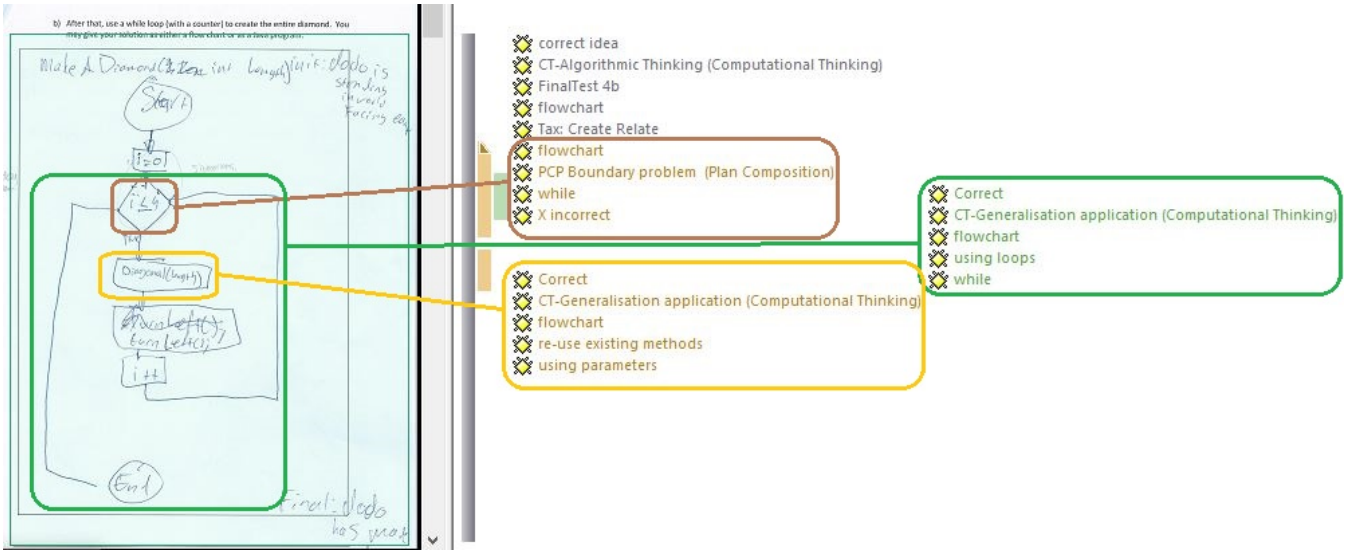

Figure 1: An example of coded student work

Multistructural: The ability to track a program and grasp various parts but not the entity they form as a whole.

Relational: The ability to fully understand a complex concept (such as concurrency) and to coherently explain its various facets.

The result is a nine-level taxonomy with the highest cognitive level being relational-creating, and the lowest level being unistructural-understanding. It's aim is to pinpoint (the development of) programming skills: both problem solving (such as plan composition) and (construct-based) language aspects.

Questions (in tasks, quizzes and tests) have been classified according to the following categories:

Procedures

Assessments We conducted six assessments (including the pretest) in which students worked individually on quizzes, tests and tasks. All six assessments were completed on paper under testing conditions, without any support from a teacher or computer. These pen-and-paper assessments were chosen to get a grip on what a student can do on his own.

Two 20-minute quizzes (each counting towards 15% of the final grade) and one 100-minute summative test (counting towards 70% of the final grade) were held with the intention to assess programming skills: both problem solving skills and the Java implementation aspects taught throughout the course. Specifically, a variant of the Rainfall problem, which has been used over decades in many researches [15, 25, 38] has been incorporated in the final test.

Furthermore, there were two tasks that focussed on assessing problem solving capabilities. They were not assessed for a grade. Inspired by related research [28] and [10], these were designed to indicate which progress the students were making on Computational Thinking and problem-solving aspects, independent of a programming language.

The questions (in tasks, quizzes and tests) were designed in such a way that all categories of the taxonomy were covered, with an emphasis on categories corresponding to higher cognitive levels; see the atlas encoding for the classification of the questions.

Generally students were asked to produce flowcharts, pseudocode or code (if they were capable of writing code) as a vehicle for communicating and explicating their thoughts and strategies. The intention here was to establish whether producing code or flowcharts made a difference in the development of problem-solving strategies.

Assignments Students were given a week to complete programming assignments. Students were advised to work in pairs, and initially did so. However, for practical reasons some pairs split-up and continued to work individually. The assignments were done on the computer, using the Greenfoot IDE. After completion, students handed in their (working) code for teacher feedback.

Here, we will use the results of the assignments to examine the difference between solutions that were devised solely by using pan-and-paper and solutions that were coded and evaluated using Greenfoot's IDE.

Data analysis

We analyzed the written assessments (tasks, quizzes and tests) using a coding procedure, followed by a more in-depth qualitative analysis, using the tags as pointers to relevant text segments. Table 1 summarizes the most important categories of codes used. Each error has been labelled using one or more of 162 different specific codes. We identified the *correctness* of a solution with respect to its requirements as follows:

Correct: semantically and syntactically correct.

Correct idea: correct on an abstract level (correct plan has been selected), but contains implementation errors. Minor adjustments will solve the problem successfully.

Code group	Description
Solution correctness	correct, correct idea, incorrect.
Construct Based	(variable) assignment, control structures (selection, loops), Boolean statements, method-related, OO-design, types.
Plan Composition	summarization, transient-experience, boundary, cognitive load, abstraction and generalisation, loops.
Translation flowchart to code	Incorrect translation from correct flowchart, or accurate translation from incorrect flowchart.
Solution format	flowchart, code, pseudo-code.

Table 1: Summary of codes used

Incorrect: incorrect on a conceptual level. Even major improvements will not result in a workable solution.

Whenever the *correct idea* code was used, another subcode was linked indicating the specific issue. As to *misconceptions*, we distinguished between two categories of programming problems:

Construct-Based errors: language specific errors which often, but not always, lead to syntax errors (see Table 2);

Plan Composition Problems: related to problem-solving and algorithmic thinking, lead to logical errors (see Table 3).

Programming errors were coded accordingly, and for each of these errors, we zoomed in and tagged them with additional subcodes specifying which specific misconception the student had made. Each response format was also tagged as *code*, *flowchart* or *pseudo-code* accordingly. Figure 1 illustrates an example coding of student's work.

5 RESULTS

The written assessments (tasks, quizzes and tests) were coded (in *Atlas.ti*[1]) and the 981 tags were subjected to a qualitative analysis. We summarize our findings below.

Construct-based errors

The most predominant construct-based misconceptions are related to methods (definition, use), variables (declaration, initialization, assignments), and boolean conditions. The majority of these errors can be detected with the assistance of a compiler. Some cause logical errors, such as missing out entire cases when conjugate boolean statements (combined with negation) were used. These were evident on written tests, but not in the compiler-assisted assignments. The number of mistakes improved over time, but did not disappear.

Plan Composition problems

Plans. Section 2 summarizes the plans taught. Each of these were each assessed in quizzes or tests:

- Triangular swap plan and comparison-based sort algorithm: Every student was able to correctly trace and summarize the goal of each.
- Counter-controlled loop: Application was assessed in the final test while creating a generic multi-structural or relational solution. Only one student, a student lagging behind on homework, was not able to select the appropriate plan. Three students were able to generate correct solutions. The others were able to select the appropriate plan and recall all the details of the plan, but made some implementation errors.
- Min/Max plan: Through all students were able to select the appropriate plan for application, many construct-based errors were made during implementation and details were omitted.
- Guarded exception plan (divide-by-zero, a variant on the rainfall problem): This plan was not taught explicitly nor implicitly. We surprised the students just to see how far they could get on their own. Without explicit instruction, no student guarded against the unexpected divide-by-zero in the Rainfall-problem.

To summarize, students understood how plans work and selected them appropriately. However, a plan must be taught (either explicitly or implicitly) in advance. Students successfully related the given problem to a previous problem they solved (generalisation), and tried to either apply the plan directly, or fine-tune the plan to solve the problem at hand, but made (mainly syntax) mistakes in the process.

The condition in the loop-plan was the largest source of boundary-errors. With one exception, students were capable of coming up with a generic solution to a complex (relational-level problem) requiring an adequate control variable and updating it appropriately. The errors were caused when choosing the specific boundary. For example, a student correctly applied the traverse-list plan and recalled that a `while` loop must terminate when a list is empty (correct idea for a plan), but a *type* misconception is evident as the student tried to compare a list to the integer value '0' instead of using the method `isEmpty()`. These mistakes pertain to paper tests and prevail throughout the entire course, however, were not evident in computer-assisted homework assignments.

Flowcharts First. In two assessment questions, students were given the choice whether to answer using code or flowchart. Novice students always chose flowcharts over writing code. On a unistructural level task, the students were rather uniformly distributed, about half chose to answer using a flowchart. However, on a more complex relational level task, substantially more students (three quarters) choose flowcharts over code.

As problems become more complex students tend to make careless mistakes (putting detailed steps in an incorrect order), forgot steps in a plan or just can't seem to focus on small details at a time (choosing correct boundaries). These errors are found in both flowcharts and code, but noticeably less frequent when they were asked to decompose the problem and focus on a specific subtask (subsiding to unistructural level).

Summarization problems (omitting secondary requirements such as guarding for an error) increased with complexity. During coding,

[1] http://atlasti.com/

Bug token	Description
Assignment	Inverted assignment or declaring a variable when trying to use variable.
Control structures	Incorrect branch selection, misconception that 'then' branch is always executed, or misunderstanding the role of a loop control variable.
Boolean statements	Confusing an assignment with a comparison operator, misconceptions about negation or conjugation of statements
Method-related	Executing a method instead of defining it, failing to appropriately use arguments or store return values, or misunderstanding scope/life-span.
OO design	Not capable of creating a second object of a class, making a copy of a reference instead of a new object, or confusing local variables with instance variables.
Types	Incorrect assignments, such as an integer value to an (compound data) object.

Table 2: Construct-Based bug tokens

Bug token	Description
Summarization	Secondary requirement of a program omitted (such as guarding for an error, returning or printing a result).
Transient-experience	Incorrectly fine-tuning a plan to the specific situation (omit a specification not appearing in the general plan or a previous similar problem).
Boundary	Inappropriate selection of boundary points for a specific plan.
Cognitive load	Omit small but significant parts of a plan, or overlooking plan entirely (being capable of performing a task at a unistructural level but not on a multistructural or relational level where a similar task is interleaved or intertwined with other tasks).
Abstraction and generalisation	Failing to re-use existing components(recognizing patterns or using parameters).
Loops	Omit loops entirely or confuse an **`if..then..else`** with a **`while`**-loop.
Optimization	Introducing an error while trying to optimize (attempting to reducing redundancy or define sub-methods).

Table 3: Plan Composition Problem bug tokens

students often omitted the last specified step, such as returning or printing a specified value. Less summarization errors are made using flowcharts than with code. It appears that flowcharts help maintain a bird's-eye view and keep goals in mind.

Students have been trained to initially abstract from unnecessary details and describe their algorithm at a high-level, leaving the details to be elaborated on at a later point in time. Using flowcharts students decompose problems into subtasks. The sub-methods themselves then became rather straightforward, allowing for direct implementation into code.

Figure 2 illustrates the work of a student who is competent at producing syntax-error free code (on paper). He has previously shown to be capable in creating correct flowcharts on a multistructural level, but here shows ample PCP issues while writing his solution directly in code (without using a flowchart first). This code snippet contains no less than 7 errors (incorrect parameters, initialization error, wrong boundary choice, **`else`** without **`if`**,...).

Inspection of other direct-code implementations show similar misplaced, omitted or superfluous method calls. These types of errors are less abundant when students sketch a flowchart first. When directly coding, students also fail to link subtasks together properly (algorithmic thinking), and in doing so omit loops (using an **`if..then..else`** where a **`while`** is expected), omit or misplace method calls or call a method unnecessarily. These are cognitive overload errors which the same students do not make on a unistructural level.

Analysis of the final test and homework assignments indicates that the students have absolutely no trouble creating and applying *their own* sub-methods appropriately (multistructural or relational level). Apparently, training to design a flowchart first, helps students identify subtasks and modularize their solution. Moreover, decomposition into subtasks reduces a multistructural problem into multiple unistructural tasks. In turn, this diminishes overload as each is tackled individually.

Translating flowchart to code

Students correctly translate their flowcharts into code. Students trust their initial design and translate this accordingly. Mistakes in flowchart design are incorporated directly into their resulting code. In only two cases, students introduced Plan Composition errors during translation from flowchart to code: (1) omitting a specific method call, and (2) mistaking the scope of a for-loop. On both occasions students were dealing with a rather complex relational-create level question. Both students had previously revealed to be capable of adequately making a translation in lower-level tasks.

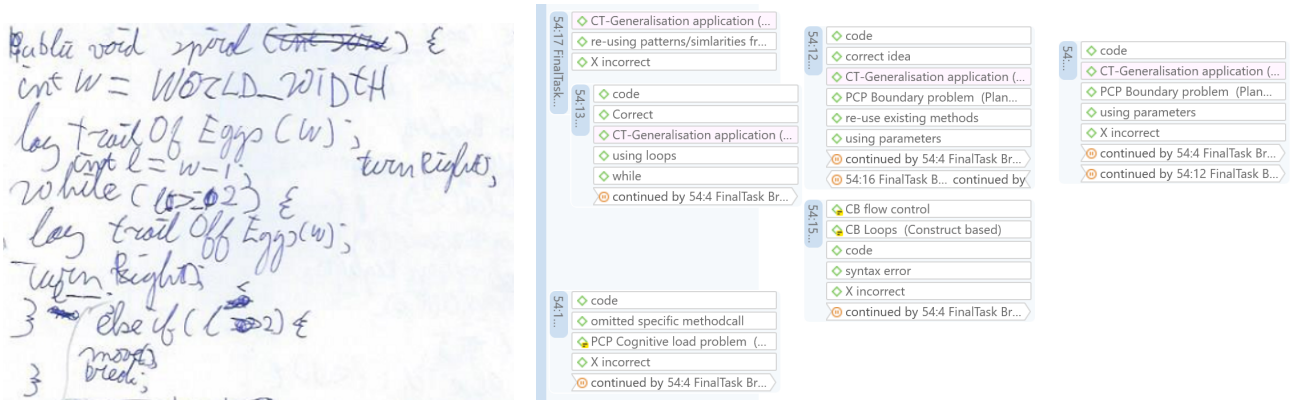

Figure 2: Flowchart skeptic (left: code with evidence of cognitive load errors; right: coding in Atlas)

6 DISCUSSION

Flowcharts are used to structure and guide the problem solving process of programming. We summarize our findings according to the problem solving steps presented in Section 2:

1. Abstract the problem from its description. Unplugged activities help to understand the problem. Flowcharts are used to identify the relevant aspects in the problem description and model these in an abstract framework. This process supports communication in an early stage, facilitating brainstorming and tinkering. We see that having a plan increases self-efficacy, and gives students the confidence necessary to get started on a complex problem. A sketched flowchart provides means for peer-reviews and teacher feedback on the proposed solution. In addition, the high-level description helps abstract from details and focus on dealing with the problem instead of implementation details.

Using the combination of unplugged activities and flowcharts, students seem more able to deal with more complex problems, come up with more creative solutions and run into less troubles in the process. It seems to help structure, develop and communicate thoughts and problem solving strategies prior to implementation.

When dealing with more complex problems, students generally prefer creating high-level flowcharts to code. Our results suggest that students with prior programming experience are generally skeptical of using flowcharts. However, when flowcharts are introduced at an early stage and students are nurtured to use them habitually, we observe that students tend to have relatively less cognitive-load issues when dealing with more complex problems, despite their shorter programming experience. In our results, more summarization errors were made when directly implementing code than when using flowcharts. Omitting the last specified step in a goal (such as returning or printing a specific value) indicates that the students fail to maintain a bird's-eye view and loose sight of their goals.

2. Generate sub-problems. As high-level flowcharts summarize the problem at hand, they seem to naturally facilitate design decomposition. Drawing a flowchart helps identify and recognize subproblems and divide the solution into more manageable chunks. For each subcomponent, the solution strategy and selection of plans is made and evaluated prior to implementation. Patterns and (obsolete) repetitions are recognized at an early stage. We claim that sketching an idea as a flowchart and thereby removing the burden of dealing with specific language constructs (syntax) lowers the cognitive load. It helps the student focus on finding an adequate approach to a problem.

Our results suggest that going through the trouble of defining sub-methods in flowcharts leads to exactly those cognitive load-problems that flowcharts should help alleviate. Students experience making detailed low-level flowcharts as cumbersome and boring. A flowchart should be used as a method of abstracting from the details, not to further refine decomposed units. Creating a flowchart must remain practical, worth the effort and goal-oriented: high-level.

3. Transform sub-problems into sub-solutions. With the design already thought-out, the student can focus specifically on coding details. Each flowchart component maps to a corresponding sub-method and is tackled individually. Students accurately and easily translate flowchart design into code. According to our observations, any mistakes in the design are adopted into code. This process, in which the student deals with the problem for a second time, does not lead to a reconsideration of the solution. Once students get into a doing mode, the thinking seems to stagnate. This emphasizes the need to think clearly before starting.

Using Greenfoot's visually compelling environment each sub-solution is evaluated individually. The immediate visual feedback reinforces testing. If necessary, design or implementation adjustments are made. Learning problem solving skills and Java language constructs is thus interleaved with its application in an iterative think-act process.

4. Re-compose the sub-solutions into a working program.

The flowchart can be used as a road-map, guiding students through the composition of the sub-solutions into a complete solution. The algorithm that controls the sequence of events is implemented as specified by the flowchart. As the testing of the sub-solutions has been done thoroughly, this step is confined to reconsidering the manner in which sub-methods are interleaved. The result is tested and evaluated to ensure that the solution adheres to the requirements, reducing summarization errors.

The use of high-level flowcharts helps maintain a bird's eye view of the problem at hand, enabling students with solution evaluation: has the problem been solved?

5. Evaluate and iterate.

After having tested each individual sub-solution, the program as a whole is validated for correctness and completeness. Contrary to our experience with pure textual results, students generally don't give up until the result is exactly as proposed. The Greenfoot environment compels students to take appropriate action to fix any errors.

Explicit attention should be given to reflection and evaluation in order to ensure that students learn from their mistakes through an iterative think-act process. This will help them learn from both construct-based and plan-composition errors.

Plans

Plans were taught explicitly, implicitly or not at all (see Section 2). Students had no problems understanding and selecting appropriate plans that were either *explicitly taught* in class or *implicitly taught* through scaffolded assignments. Conversely, students could not apply a plan *without prior instruction* or (scaffolded) practice.

Our results confirm de Raadt's call for teachers to teach plan-strategies explicitely[12].

The role of coding

Coding gives students immediate feedback on their algorithms (in the role of a guide-by-an-expert). Students should be encouraged to actually code and run their algorithms. Lack of coding experience leads to deficient fundamental construct-based knowledge crucial to programming, such as assigning values to variables and understanding booleans. Working together with a picky compiler makes students aware of the need for precision. It helps them find, fix and learn from their own (logical) mistakes.

Students rely on the computer to translate plans into correct and specific solutions. Whether syntax or logical errors, students solve these problems on their own during computer-assisted programming.

Syntax error detection: Most of the CB errors made in the pen-and-paper assessments are syntax-related (such as writing '=>' instead of '>='). The compiler helps detect and remove these. However, these pen-and-paper errors don't disappear over time. Students seem to be strongly reliant on compiler-assistance, and apparently, students don't internalize what the compiler is complaining about. On the other hand, one can argue that knowledge about the exact syntax is of minor importance when designing and implementing an algorithm.

Logical error detection: Immediate (visual) feedback from the programming environment gives students feedback on their algorithm. It supports the detection of logical errors. Our in-class observations indicate that, due to its visual nature, students are highly motivated to keep going until the result is exactly as expected (or of even higher quality).

In contrast to the paper-assessments, students never omit the testing phase during computer-assisted assignments. Whilst testing, students can recognize whether adequate boundaries have been chosen, and adjust these appropriately (possibly by trial-and-error). Even though paper-tests indicate an improvement over time, students continue to make logical errors. It seems as though the correction they make is not internalized. A more explicit reflection and evaluation phase may improve learning from mistakes. This is important, as logical errors are harder to detect in a less visual environment.

Block-based systems

Block-based systems, like Scratch, Blockly, Alice, Greenfoot or AppInventor are particularly successful among young novices. They help abstract from concrete syntax, just like flowcharts. An advantage over flowcharts is that they do not require an explicit coding step to convert to an executable program. A disadvantage is that the instructions used are low-level (just like individual Java statements). For complex problems this reduces to the same types of problems as with concrete code: due to poor design (such as lack of modularization) students have no overview over the complete program. In addition, it appears that the conceptual difficulties in the understanding and use of key concepts of programming such as variables and loops still persist [18]. Moreover, mastering text-based programming languages like Java or Python is an enduring educational goal for students who wish to gain more in-depth experience in coding.

Using block-based systems students are expected to use the computer right from the very start. We observed, not only throughout the course of this study, but also in many other programming courses, that the computer seems to cause students to switch from a think-first-mode to an act-first-mode. The essence of our approach is to keep students away from the computer so that they can focus on problem-solving and design-aspects such as decomposition, instead of diving straight into details.

Suggested improvements to the material

Reflection: Focus on *iterating* between think-act and specifically incorporate *evaluation*. To optimize the learning potential, students should be encouraged to reflect on what went wrong. This will specifically help students to understand syntax errors (such as those related to types and assignments) and logical boundary errors. It should be emphasized that ad-hoc solutions may introduce new errors. If necessary, the flowchart should be adjusted accordingly.
Flowcharts: Establish concrete expectations about high-level flowcharts, indicating which details should be left out. Additionally, design rules should be introduced in order to avoid arrow-spaghetti.

7 CONCLUSION AND FUTURE WORK

The concepts of problem solving, algorithmic thinking and development of programming solutions are closely related. Algorithm design and a structured manner for problem solving become indispensable when dealing with complex problems and their solutions.

The results of this exploratory case study indicate that our approach significantly improves both algorithmic design competences and programming skills of novices programmers.

Thinking-first and planning ahead are important first steps in problem solving. It requires a particular focus, we believe best achieved away from the (distracting) computer. Selection of the

suitable plan seems to benefit from explicit instruction and 'thinking' before 'acting'. Algorithmic thinking skills and a bird's-eye view are needed to link plans together to create an adequate solution.

An appropriate plan to solve a problem (despite of any construct-based errors introduced during implementation) has far more potential of leading to a correct solution, than an incorrect plan which is implemented bug-free. Most construct-based errors are syntax errors which the compiler will complain about. However, logical errors are far more difficult to detect and fix.

Flowcharts, and a stepwise approach to problem solving, can aid novice programmers to keep track of where they are and give guidance to what they need to do next, similar to how a road-map helps navigate. High-level flowcharts assist decomposition, subtask recognition and lighten the cognitive load as each subtask is dealt with independently.

The exploratory case study that we conducted was small-scale. The material has been improved according to the annotations in Section 6. The Dutch version is being published as official secondary school material for the Dutch national curriculum, and includes formative and summative tests. The English version will be made available through the Greenfoot website. We plan to carry out a *follow-up study* in the course of this year. Student and teacher feedback while using the improved version will be collected and analysed. Feedback and new suggestions for improvements are to be incorporated in the following iteration.

REFERENCES

[1] Vicki L Almstrum. 1999. The propositional logic test as a diagnostic tool for misconceptions about logical operations. *Journal of Computers in Mathematics and Science Teaching* 18 (1999), 205–224.

[2] Steve Armstrong, Sally Brown, and Gail Thompson. 2014. *Motivating students.* Routledge.

[3] Tim Bell, Jason Alexander, Isaac Freeman, and Mick Grimley. 2009. Computer science unplugged: School students doing real computing without computers. *The New Zealand Journal of Applied Computing and Information Technology* 13, 1 (2009), 20–29.

[4] M. Butler and M. Morgan. 2007. Learning challenges faced by novice programming students studying high level and low feedback concepts.. In *Proceedings of the ICT: Providing choices for learners and learning*. Ascilite (Singapore2007), 99–107.

[5] Martin C Carlisle, Terry A Wilson, Jeffrey W Humphries, and Steven M Hadfield. 2005. RAPTOR: a visual programming environment for teaching algorithmic problem solving. In *ACM SIGCSE Bulletin*, Vol. 37. ACM, 176–180.

[6] Stephen Chen and Stephen Morris. 2005. Iconic Programming for Flowcharts, Java, Turing, etc. *SIGCSE Bull.* 37, 3 (June 2005), 104–107.

[7] Jacqui Chetty and Duan van der Westhuizen. 2015. Towards a pedagogical design for teaching novice programmers: design-based research as an empirical determinant for success. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research.* ACM, 5–12.

[8] Stephen Cooper. 2010. The Design of Alice. *Trans. Comput. Educ.* 10, 4, Article 15 (Nov. 2010), 16 pages.

[9] Quintin Cutts, Emily Cutts, Stephen Draper, Patrick O'Donnell, and Peter Saffrey. 2010. Manipulating mindset to positively influence introductory programming performance. In *Proceedings of the 41st ACM technical symposium on Computer science education.* ACM, 431–435.

[10] Valentina Dagiene, Linda Mannila, Timo Poranen, Lennart Rolandsson, and Gabriele Stupuriene. 2014. *Reasoning on Children's Cognitive Skills in an Informatics Contest: Findings and Discoveries from Finland, Lithuania, and Sweden.* Springer International Publishing, Cham, 66–77.

[11] Garrett Dancik and Amruth Kumar. 2003. A tutor for counter-controlled loop concepts and its evaluation. In *Frontiers in Education Conference*, Vol. 1. STIPES, T3C–7.

[12] Michael de Raadt. 2008. *Teaching programming strategies explicitly to novice programmers.* Ph.D. Dissertation. University of Southern Queensland.

[13] Michael De Raadt, Mark Toleman, and Richard Watson. 2007. Incorporating programming strategies explicitly into curricula. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88.* Australian Computer Society, Inc., 41–52.

[14] Michael de Raadt, Richard Watson, and Mark Toleman. 2006. Chick sexing and novice programmers: explicit instruction of problem solving strategies. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52.* Australian Computer Society, Inc., 55–62.

[15] Kathi Fisler. 2014. The recurring rainfall problem. In *Proceedings of the tenth annual conference on International computing education research.* ACM, 35–42.

[16] Michal Forišek and Monika Steinová. 2012. Metaphors and Analogies for Teaching Algorithms. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12).* ACM, New York, NY, USA, 15–20.

[17] Merrilyn Goos, Peter Galbraith, and Peter Renshaw. 2002. Socially mediated metacognition: creating collaborative zones of proximal development in small group problem solving. *Educational Studies in Mathematics* 49, 2 (2002), 193–223.

[18] Shuchi Grover and Satabdi Basu. 2017. Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17).* ACM, New York, NY, USA, 267–272.

[19] O. Hazzan, T. Lapidot, and N. Ragonis. 2011. *Guide to Teaching Computer Science: An Activity-Based Approach.* Spriger.

[20] Geoffrey L Herman, Michael C Loui, and Craig Zilles. 2010. Creating the digital logic concept inventory. In *Proceedings of the 41st ACM technical symposium on Computer science education.* ACM, 102–106.

[21] Cindy E Hmelo-Silver and Howard S Barrows. 2008. Facilitating collaborative knowledge building. *Cognition and instruction* 26, 1 (2008), 48–94.

[22] Paul A Kirschner, John Sweller, and Richard E Clark. 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 (2006), 75–86.

[23] Michael Kölling. 2010. The Greenfoot Programming Environment. *Trans. Comput. Educ.* 10, 4 (Nov. 2010), 14:1–14:21.

[24] Marja Kuittinen and Jorma Sajaniemi. 2004. Teaching roles of variables in elementary programming courses. *ACM SIGCSE Bulletin* 36, 3 (2004), 57–61.

[25] Antti-Jussi Lakanen, Vesa Lappalainen, and Ville Isomöttönen. 2015. Revisiting rainfall to explore exam questions and performance on CS1. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research.* ACM, 40–49.

[26] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *Trans. Comput. Educ.* 10, 4 (Nov. 2010), 16:1–16:15.

[27] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin* 33, 4 (2001), 125–180.

[28] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2013. Learning computer science concepts with Scratch. *Computer Science Education* 23, 3 (2013), 239–264.

[29] Cade Metz. 2012. Google blockly lets you hack with no keyboard. (2012).

[30] Briana B. Morrison, Brian Dorn, and Mark Guzdial. 2014. Measuring Cognitive Load in Introductory CS: Adaptation of an Instrument. In *Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14).* ACM, New York, NY, USA, 131–138.

[31] Richard E Pattis. 1981. *Karel the robot: a gentle introduction to the art of programming.* John Wiley & Sons, Inc.

[32] George Polya. 1945. How to solve it: A new aspect of mathematical model. (1945).

[33] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education* 13, 2 (2003), 137–172.

[34] Cynthia C Selby. 2015. Relationships: computational thinking, pedagogy of programming, and Bloom's Taxonomy. In *Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE '15).* ACM, New York, NY, USA, 80–87.

[35] Judy Sheard, S Simon, Margaret Hamilton, and Jan Lönnberg. 2009. Analysis of research into the teaching and learning of programming. In *Proceedings of the fifth international workshop on Computing education research workshop.* ACM, 93–104.

[36] Elliot Soloway, Kate Ehrlich, Jeffrey Bonar, and Judith Greenspan. 1982. *What do novices know about programming?* Yale University, Department of Computer Science.

[37] Juha Sorva. 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education* 13, 2 (2013), 8.

[38] James C Spohrer and Elliot Soloway. 1986. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM* 29, 7 (1986), 624–632.

[39] David Wolber, Hal Abelson, Ellen Spertus, and Liz Looney. 2011. *App Inventor - Create Your Own Android Apps.* O'Reilly. I–XXII, 1–360 pages.