Radboud Universiteit Nijmegen

Institute for Computing and Information Sciences
Software Science (SwS)

Investigating students’ concurrent programming strategies

Master Thesis

Bartjan Zondag

Supervisors:

prof. dr. Erik Barendsen
dr. Sjaak Smetsers

Final version

Nijmegen, November 2017

Abstract

This research focusses on finding the programming strategies students apply while working on a concurrency assignment. There are different strategies one can apply to create a program, three of which are highlighted in this thesis: trail-and-error, adaptation strategy, and reflection-in-action. The strategy used influences the way students learn from working on an assignment. Therefore, it is important for teachers to know which strategy students use and when. However, there is no standardized method to find the strategy that students use. The method described in this study can be used to find the strategy students use based on programming related activities of students while working on a programming assignment. A case study is used to validate the method and to get more insight in the strategy students use while working on a concurrency assignment. Parts of the assignment show trial-and-error behavior while other parts show a mixture of adaption and reflection-in-action strategies. Several directions to increase the distinguishability of these strategies are suggested, which can enhance the results the method yields.

Acknowledgement

This thesis is the final product of the Information Sciences master at the Radboud University Nijmegen. I would like to thank my supervisors Erik Barendsen and Sjaak Smetsers for their support, time and valuable feedback.

I would like to thank my girlfriend, Fanny, for her continued emotional support and linguistic guidance. Finally I would like to thank my family, especially my father and mother, and friends for their support.

Index

1 Introduction

Introductory programming courses are full of challenges for novice programmers (Kolikant, 2001). With the goal to create 'new' programs a novice programmer has to learn which code needs to be added, changed or deleted in order to realize a program. However, only creating code is not enough, a novice has to gather and understand information about the syntax of the programming language and programming concepts, and learn how they intertwine in a program (Whalley J. L., 2006). After this the novice writes and assesses parts of the program to see if it fits the goal the programmer initially had in mind. The way a novice programmer combines these activities is what we call in this study the 'macro strategy' he or she uses to create a program, though a programmer can be unaware that he is using a certain macro strategy.

One of the strategies we use in this study is 'reflection-in-action', which focusses on finding the reason why code is behaving in a certain way. By examining a this behavior and using prior understanding, a programmer tries to come up with reasons why the code gives a certain result. This way of thinking can also be applied beforehand by making 'test cases' that need to be fulfilled in order to check the correctness of the program (Edwards S. H., 2004). The second strategy is the 'adaptation', or sometimes called the copycat, strategy, in which the programmer searches for example solutions that can be integrated in the program (Hou, Jablonski, & Jacob, 2009). This strategy can lead to the introduction of bugs and duplicate code, it also does not guarantee that the copied code in question is understood by the programmer. The last 'strategy' we use in this study is trial-and-error which uses a simple structure of code that is repeatedly written (trial) and found inadequate (error), until the solution is "good enough" (Lönnberg, Berglund, & Malmi, 2009). In this study we treat trial-and-error as a strategy that can be used by novices.

Each of these strategies influences the way novices create a program but also how much they learn from it. Before a novice can use reflection-in-action to predict the outcome of a piece of code, he already needs to master basic comprehension and analysis skills regarding the topic at hand (Buck & Stucki, 2000). Buck & Stucki state that expecting this behavior too early could be harmful for the learning process rather than helpful. However, when reflection-in-action can be used properly it is a 'strong' strategy that can lead to new understanding in a short amounts of time (Edwards S. , 2003). If used correctly the adaptation strategy guides novices in the right direction. Additional procedural guidance and example code assist students in creating a solution. Giving too much guidance can impede the need for novices to increase their programming skills to solve the task at hand (Yadin, 2011). The last 'strategy' trial and error is suitable for beginners in a disciple, but if they keep on using it for extended periods it can become a handicap (Edwards S. H., 2004). This handicap can best be seen in programs with increased complexity were seemingly 'random' changes do not lead to acceptable solutions (Kolikant, 2005).

As we can see each of these three strategies each have their own focus. Trial-and-error and adaptation can be well suited to solve the problem; this does not mean that a novice optimally learns from this experience. However as a teacher how do you know which strategy students employ to work on an assignment? While multiple strategies may be suitable to solve the problem, not all strategies are desired since they do not increase the programming skills of novice programmers.

In this study we showcase a method that can identify strategies students use while working on a programming assignment, this is done by looking at the sequences of programming activities. With this method in mind, a distinction can be made of the strategy students use for different programming concepts within an assignment. This can assist teachers to verify if the students use the strategy that is best suited for the assignment at hand.

2 Background information

The first three sections of this chapter describe how the knowledge of students is tied to strategies they use while working on a programming assignment. Section 2.4, 2.5 and 2.6 focus on the SOLO taxonomy and how this is used in this study. Section 2.7 describes different strategies that are found in other studies. Section 2.8 focuses on difficulties encountered by students while working on programming assignments.

2.1 Constructivism

Constructivists interpret students learning as the development of personalized knowledge frameworks that are continually refined. According to constructivism, a student must actively construct knowledge, rather than absorbing the knowledge from textbooks and lectures (Davis, Maher, & Noddings, 1990). While constructing this knowledge, each student develops their own set of rules, or “alternative frameworks” (Ben-Ari, 2001). These alternative frameworks “naturally occur as part of the transfer and linking process” (Clancy, 2004), represent the prior knowledge of students, essential to construct new knowledge (Smith, diSessa, & Roschelle, 1992). “When learning, the student modifies or expands his or her framework in order to incorporate new knowledge” (Eckerdal, et al., 2006).

2.2 Conceptual Aspects

Programming tasks involve the use of a variety of programming knowledge concepts (Schwill, 1994) (Tew & Guzdial, 2010). A source of programming difficulties can be explained through the lack of understanding of these programming knowledge concepts (“misconceptions”). Another source can be explained by a personal view of the notional machine, an idealized, conceptual computer whose properties are implied by the constructs in the programming language employed (Boulay, O'Shea, & Monk, 1981). Novices programmers do not completely understand what the notional machine can be instructed to do and how it manages to do it. This mismatch between the personal abstraction of novice programmers and the actual working of the notional machine can explain a variety of misconceptions (Sorva, 2013).

The notional machine and execution model that are used to create concurrent programs are considerably more complicated than similar sequential programs. Novice programmers have difficulties to envision and comprehend the parallel and dynamic behavior of concurrent programs. When novice programmers work on a concurrency assignment, the discrepancy between the actual and mental model of novice programmers of the notional machine become visible. The novice programmers' assessment of concurrent execution can give rise to incorrect views on the correctness of his concurrent program (Kolikant, 2005). This is seen back by novice programmers, as they tend to interpret a single successful test run on a characteristic input as sufficient evidence for correctness of their program. Novice programmers often confuse method execution and scheduling. They have difficulties recognizing the execution sequence of threads. For example multiple threads that access the same variable at the same time introducing situational bugs.

2.3 How do novices apply their knowledge

This study focusses on the knowledge of novice programmers, how and when they use programming knowledge and what knowledge they still need to 'gather'. When applying this knowledge, novice programmers can encounter difficulties. Below a set of common difficulties is explained.

Novices encounter difficulties while applying programming knowledge. Different sources of these difficulties are the cognitive load, the amount of 'new' information novice programmers have to process, another source is misconceptions about language constructs, the way a programing language is described compared to a natural language. Spohrer and Soloway (1986) analyze these two difficulties, they state that the cognitive load of novice programmers have a bigger impact on programming errors than misconceptions about language constructs.

This study focusses on concurrency programming, which brings its own additional set of difficulties for novice programmers. These difficulties tend to focus on choosing the appropriate synchronization mechanisms and primitives to meet certain synchronization goals, which are required for concurrent programming. Additional to this, novice programmers have trouble reasoning why the combined use of these synchronization primitives leads to the expected behavior (Xie, Kraemer, & Stirewalt, 2007).

Each student has their own approach in the way they deal with these difficulties. Some spend more time on gathering information about the concepts they are working with, before applying them in their own project. Others tend to start by implementing their code, without knowing how concepts exactly work or fit together. Robins, Rountree and Rountree (2003) argue that most novice programmers spend little time planning. For the case of concurrency, Lönnberg, Berglund and Malmi (2009) found that some students even consider designing unnecessary or even impractical. As a result such students tend to rely on a trial and error strategy (Lönnberg, Berglund, & Malmi, 2009). By not creating designs or plans students are forced to combine plans on the flight, increasing the cognitive load while they are working on assignments.

The above described difficulties novice programmers can have. Experts do not have these difficulties. Bransford, Brown and Cocking (2000) describe serval key principles of experts' knowledge:

1. Experts notice features and meaningful patterns of information that are not noticed by novices.
2. Experts have acquired a great deal of content knowledge that is organized in ways that reflect a deep understanding of their subject matter.
3. Experts' knowledge cannot be reduced to sets of isolated facts or propositions but, instead, reflects contexts of applicability: that is, the knowledge is "conditionalized" on a set of circumstances.
4. Experts are able to flexibly retrieve important aspects of their knowledge with little attentional effort.
5. Though experts know their disciplines thoroughly, this does not guarantee that they are able to teach others.
6. Experts have varying levels of flexibility in their approach to new situations.

(Bransford, Brown, & Cocking, 2000)

The above principles can be explained as follows, when an expert needs to solve a 'new' problem, he can rely on his experience to find the best suited patterns and apply these to create a solution. Combining these different patterns creates a cognitive load, one has to recall different patterns and combine these with the new information, the problem, at hand. However there is a limitation on how much 'new' information a person can process at a time. Since an expert only needs to process limited amounts of new information this forms no obstacle. However this is different when we look at novice programmers. Novice programmers do not have a collection of patterns that they can use to solve a new problem. This in term means that the cognitive load increases, not only must the student create patterns, he or she also has to combine them on the flight, thus increasing the chance of defects to occur.

2.4 SOLO Taxonomy

SOLO, Structure of Observed Learning Outcomes, is a way to classify learning outcomes in terms of their complexity and by doing so it focusses on quality, rather than how many lines of code are correct (Biggs & Collis, 2014). This taxonomy is prominently used for the skill programming (Whalley, Clear, Robbins, & Thompson, 2011). SOLO has five levels of complexity:

- Prestructural/incompetence, lack of knowledge or failing to grasp an understanding of the subject
- unistructural, novices pick up only one or a few aspects of the task
- multistructural, novices see several aspects, however they are unrelated
- relational, multiple aspects of the task are integrated into a whole
- extended abstract, the aspects needed for a task can be applied to a new task

Ginat (2004) created a table that links algorithm designs to the SOLO classification levels, see Table 1. In this study we refer back to this table to link the works of novices to the SOLO level as described by Ginat.

SOLO LEVEL	**Algorithmic design**
Prestructural (P)	Substantial lack of knowledge of selection and implementation of generic design patterns.
Unistructural (U)	Direct translation of the specifications into a straightforward implementation of a generic design pattern
Multistructural (M)	A translation of the specifications into flexible manipulation of a generic design pattern; or a simple, elementary composition of more than one generic pattern.
Relational (R)	A valid well-structured solution that involves the composition of two or more design patterns, integrated in a non-simple, interleaved manner, to form a logical whole.
Extended Abstract (E)	Insightful capitalization on hidden task characteristics; and/or a generalized structure that encapsulates abstraction beyond the required solution.

Table 1 SOLO classification for algorithm design (Ginat, 2004)

In the study of Izu et al. (2016) they created a similar table, this time with the focus on code design, Table 2. In Table 2 we can see that Izu uses the term ‘building blocks’, with this he means code snippets that represent the knowledge that is necessary to complete the programming assignment and exam.

SOLO LEVEL	Code design
Prestructural (P)	Substantial lack of knowledge of basic building blocks and their use to solve the given task.
Unistructural (U)	Use of one building block or template to partially or completely solve the given task.
Multistructural (M)	Modify/extend a building block, or combine sequentially two or more blocks to partially or completely solve the given task.
Relational (R)	Combine and integrate in a non-simple manner two or more building blocks to solve the given task.

Table 2 Generic SOLO classification for code design (Izu, Weerasinghe, & Pope, 2016)

In this study we use both Table 1 and Table 2 to determine what code novices are working on, and in what SOLO level the algorithm of the novices are.

2.5 Programming knowledge

According to the Oxford dictionary the term knowledge is defined as follows:

Facts, information, and skills acquired through experience or education; the theoretical or practical understanding of a subject.

The ideal case would be to study all the aspects of knowledge that novices use during the assignment, however ones intellect, the knowledge a person possesses, is not directly measurable. However we can observe interactions of a person with the world that give a hint to the knowledge this person possesses. For this we use the SOLO taxonomy with the focus on programming knowledge as described in section 2.4. However we are not only interested in the code novices create but also in the way they gather information and assess their own program. We therefore focus on three ‘activity types’ related to programming knowledge: gathering information, implementation, and assessment.

The three activity types are described as follows:

Gathering information: Gathering information describes all information accessed outside the programming environment. The programming environment is the development tool the programmer uses to create the program, this can be an IDE or a text editor. Examples of gathering information are, accessing google.com or stackoverflow.com to inquire information about programming knowledge concepts, but also looking into a problem statement or assignment (the reason why the program needs to be written) or asking help from a third party, a student assistant or teacher.

Implementation: Implementation describes the situation of writing, editing, and removing code by the programmer. Examples of implementation are, adding or editing a class, or function of a program, but also removing lines of code. Different studies have used the SOLO taxonomy to give structure to the way they capture implementations of novices. In this study we use the SOLO taxonomy based of Table 1 and Table 2 of section 2.4.

Assessment: In this study we use the word assessment instead of evaluation, so it is not confused with the evaluation stage of the bloom taxonomy. Assessment describes all the instances that involve code or the resulted documents (automatically generated by the program), and output of program without changing the code. Examples of assessment are looking at the output of a program to see if it gives the desired result or examining a line of code to see if there is a defect.

The three activity types are depicted in Figure 1. As described above, gathering information describes the interaction of external sources, and how they influence internal knowledge, implementation is the exertion of knowledge into the real world, and assessment describes how a person assess their own code and checking or changing their internal knowledge accordingly.

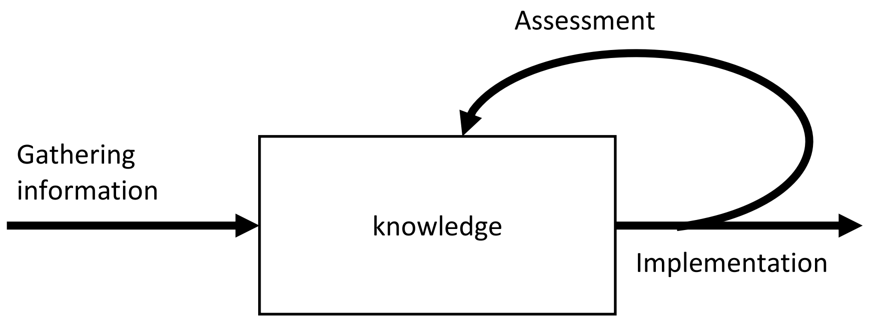


Figure 1 The box represents the internal knowledge of a person. Three interactions are depicted as gathering information, implementation and assessment. A person can exert one's knowledge into the real world and by doing so changing the world, this interaction is called implementation. To assess the correctness of the implementation one can assess their own work, and as such giving hints of constructing or 'changing' his own knowledge.

2.6 Knowledge vs micro strategy

Davies (1993) distinguishes between programming knowledge, for example being able to state how a “for” loop works, with programming strategies, and the way knowledge is applied in a program, for example the way the “for” loop is used in a program. As seen in the example above, a micro strategy is the way knowledge is applied to a program. This in term means that a novice programming ability rests on a foundation of knowledge and how to use this knowledge to come up with a micro strategy.

However these micro strategies are subject to interpretation, when does code belong to one micro strategy and not another. To solve this problem we look to the study of Izu, Weerasinghe & Pope (2016) who focused on a topic close related to micro strategies, called building blocks. These building blocks describe the complexity and correctness of code snippets of the solution in a structured well mannered way. An example of these building blocks are described can be seen in Table 3.

Table 3 List of building blocks (Izu, Weerasinghe, & Pope, 2016)

Building Block	Description
Conditional loop	While(true) with break statement while with simple condition (ie. n > 0) where the variable tested has been initialized, and it will be updated in the body.
Basic input	Accept number from user input
Assignment	Simple assignment, i.e. x = 1

Each building block has a specific, yet abstracted, piece of code tied to it. This helps connecting written code to a concept. For each piece of code the correct approach is described, which can also be used to find approaches that do not work.

2.7 Macro strategies

In this study we are interested in the different macro strategies students employ to come up with solutions while working on an assignment. In this section we describe different macro strategies that students use.

Reflection-in-Action

The Reflection-in-action strategy focusses on thinking before acting. Before you make a change, a “time out” is taken to reason what this change would do to different aspects of the application. This reasoning process relies on experience and knowledge of the programmer to understand which consequences the change has. However this way of thinking does not always come natural. To assist in this process different methods are created, for example TDD (“test-driven development”). TDD enforces the programmer to write tests that check the outcome of parts of an application before the actual code is written. This way the programmer is forced to reason if the code he or she writes adds functionality to fulfill these tests. (Kolikant, 2001) (Tew & Guzdial, 2010).

Adaptation or Copycat Strategy

Adaptation or copycat strategies focus on finding complete or partially complete solutions for the problem at hand. The goal behind the strategy is that someone already made a solution and the easiest way to solve the problem is by applying the found solution to the new problem, with minimal changes necessary. Different studies show that students try to find solutions they can copy from slides and other material provided by the teacher (Edwards S. H., 2004). The works of Spohrer and Soloway (1986) show that examples given in a textbook were copied for the assignment, expecting the behavior of the copied work to be correct. However the students in question did not take the extra requirements of the assignment into account, but they did not notice this.

Trial and error

Trial and error is a well-established technique for beginners in any discipline, and it is no surprise that this is where students start out. But why do students persist in this practice long after it becomes a handicap? (Edwards S. H., 2004)

As seen in many studies, trial and error is a common practice while novices are working on programming assignments. Trial and error strategies focus on changing individual pieces of code and testing if the working of the program is as expected (Lönnberg, Berglund, & Malmi, 2009). However trial and error strategies have the downside of becoming less effective if the problem at hand is large and complex. This can be seen in programming exercises that require multiple changes before the program works correctly (Edwards S. H., 2004). So while trial and error has its place to solve certain problems it is inadequate to solve larger and complex problems.

Combination of macro strategies

Based on different works we see that multiple strategies are combined to come up with a solution. For example it is typical that if the adaptation strategy is used an iteration of the trial-and-error strategy may follow to fit the found example to the problem at hand (Kolikant, 2001). The method we suggest in this paper takes into account that multiple strategies can be applied within the context of one assignment.

2.8 Concurrency

There are many reasons to introduce concurrency, people want to make optimal use of the computers processing power, need to control several objects at the same time, want to create independent objects to create a realistic simulation. For instance, someone might want to play a game and he has to press a key on the keyboard and move the mouse at the same time. Someone else might want a program that simulates multiple cars that drive at the same time. These coordinated and/or simultaneous activities are difficult (if not impossible) to program using a traditional or ‘sequential’ program. The problem is that traditional languages are based on a single process that executes instructions one at a time. This sequential paradigm does not match the way the real world works: People and animals act in parallel, objects interact in parallel. As a result, many real-world activities cannot be modelled in a natural way with sequential programming. The ideal solution is to use a concurrent or parallel programming language, that is, a language that allows programmers to control multiple, interacting processes. However working with concurrency brings its own downsides. One of the biggest downsides is that shared data can be corrupted if multiple threads access it at the same time.

2.9 Difficulties for novices

This section provides a rundown of difficulties that can occur for novices while working on a programming assignment.

Assignment decomposition problems

As described in section 2.3 'How do novices apply their knowledge’, there can be a multitude of reasons why a student cannot decompose the problem into smaller parts. One of these reasons can be explained with statements of Bransford (2000) “Experts are able to flexibly retrieve important aspects of their knowledge with little attentional effort”. When a teacher creates an assignment he or she believes parts of the assignment are trivial, however this is not the case for the students. This in term makes it harder for the student to find the individual tasks in a larger assignment.

Misinterpretation of the assignment

Next to splitting up an assignment it could also be that the assignment is misinterpreted. For example the teacher has a clear goal of what he wants to teach the students. The student however, interprets the goal of the assignment in a different way, leading to the 'wrong' solution or the student gets stuck because he does not understand what he has to do.

Missing content knowledge

Some problems students have are related to knowledge they have of a certain programming knowledge concept. It could be that they have no understanding of a programming knowledge concept, or that they know the theory of a concept, but cannot apply this to various situations. The only way to combat this problem is by accessing external sources that can guide, teach, or assist the student (Bransford, Brown, & Cocking, 2000).

Next to these problems, Resnick (1991) and Kolikant (2001) describe concurrency specific issues that we take into account, below is a highlight of them.

Identification of Synchronization goals

As described in chapter 2.8 'Concurrency', data can be corrupted if multiple threads access the common resource at the same time. The solution to this problem is to apply synchronization, a semaphore is an example of this. A semaphore is a variable or abstract data type used to control access to a common resource. Resnick (1995,1996) and Kolikant (2001) showed that students have problems identifying the right common resource. Instead students focus their efforts on solutions that shift the problem to a new area rather than solving it.

Centralized and Decentralized synchronization solutions

Applying synchronization can oppose challenges to novice programmers, and if done incorrectly this can lead to corrupt data. The corrupt data is caused by multiple threads changing data individually and for example overwriting each other's results.

Centralized solutions and decentralized solutions are both ‘correct’ ways to solve this and other synchronization problem. However both of them come with their own set of rules that students have to follow to implement them. When applying the concept of synchronization, as explained in the previous point, there are two options to do this. Option one is a centralized solution: in the end there is one resource that is edited. The synchronization in this case is handled by the object, computer or database that is responsible for the actual alteration of the resource. As an example, multiple computers want to edit a record in a database. When the database gets the first request it locks the record and only releases it when the computer is done with it, making sure that it can never happen that multiple computers edit the same record.

The second option is decentralized. As an example multiple computers want to edit a record in a database. Before they do this they tell each other that they are going to edit the record, then actually edit the value, and in the end they tell all the computers that they are done editing the record. The main downside of this solution is that if one computer does not comply by the rules multiple computer can still edit the data at the same time (Kolikant, 2001).

Inventing new operations

Some students attempted to solve the synchronization problem by inventing new functions, with the goal to replace a semaphore. These new functions consisted of an unwieldy way to achieve the same as a semaphore, or they ‘solved’ it erroneous. Kolikant (2001) believes the origin of this problem is found in the translation from natural language.

3 This study

Now that we have established the current status of the research field and have given an overview of the relevant literature, we will now focus on the specific aims of this paper. The first section of this chapter describes the goal of this study. To reach the goal of this study, observations of students working on a programming assignment are made (how these observations are made and how they contribute to the research goal will be described in detail in the next chapter). The specific programming assignment used here will be shown and discussed in the second section of this chapter.

Each of these two parts have their own goal. The first part tries to give answer to the main research question

How can we determine what macro strategy students use for different programming knowledge concepts?

The second part explains the case study that is used, where we focus on what the method tells us about this specific case.

3.1 Research goal

The goal of this study is to contribute to a better insight into what macro strategies students use while learning the skill programming. The macro strategy influences how students create an assignment and what they learn from the assignment while working on it. However, finding the macro strategy for each student costs a lot of time which is not available for most teachers. With this method we attempt to strike a balance between the time needed to find these strategies and the results this method gives. The method of this study accomplishes this by recognize macro strategies students use while working on a programming assignment in a systematic way.

One of the key characteristics of the method we suggest in this study is that it can be applied to any type of programming assignment. The expectation is that because of the generic approach different students can be compared in a systematic way rather than in an interpretive way.

By looking at the structure and sequence of activities of students we can find indications of the macro strategies students use. Three macro strategies are highlighted in this study: reflection-in-action, adaptation strategy, and trail-and-error. Multiple strategies can be combined by students to come up with a solution. One of the goals of the method of this study is to find the macro strategy students use for each programming concept. This helps teachers to find the concepts that the students still has problems with.

3.2 The assignment of the students

In this study two groups of students were observed to gather the data. These students follow the course ‘object oriëntatie’, object orientation. The assignment the students have to solve is as follows:

Imagine that a certain number of travellers (between 60 and 90) arrive at a train station. These people must continue to their final destination, a holiday resort, by taxi. Four taxis are available: two with a capacity of four and two with a capacity of seven people. The taxis ride back and forth as long as there are still people waiting at the station. Each taxi transports as many people as it can, or possibly less, depending on the actual number of people still waiting.

The aim of the program the students have to make is to determine how long it will take to transport these passengers. Next to this problem statement students received a UML class diagram showing a sequential solution for the problem, and a code base written in JAVA containing a sequential solution to transport passengers from the train to the station and taxis taking the passengers from the station to the destination. The goal for the students is to change the sequential behavior of the program to a concurrent, parallel or multi-threaded, solution. When looking at the solutions of the students we take the following into account:

“There are two kinds of synchronization goals: (a) prevent instructions of two or more processes from executing at the same time, and (b) prevent a set of instructions from being executed until a condition is satisfied. Synchronization goals are achieved if two conditions are fulfilled: (a) no bad scenarios (those forbidden by the synchronization goals) are possible, and (b) all good scenarios (those not forbidden by synchronization goals) are possible. A synchronization mechanism uses special instructions to achieve synchronization goals.” (Kolikant, 2001)

This means that students should not only apply synchronization in the right areas, but also make sure that it is not applied in areas it does not belong to.

The students in question are in the last semester of their first year. They already finished an assignment about the use of threads in JAVA, and the main focus of the assignment is to teach the students about the check-then-act pattern, shared values, and synchronization. This assignment is a learning exercise were 'new' knowledge is necessary to complete the task at hand. Therefore the students will most likely use information from external sources to assist them, one of these documents is created by the teacher to guide them. This document is called 'the six step plan', this document describes six steps that the student can follow to guide them in the creation of a concurrent program. The six steps are as follows:

1. *Problem analysis: investigate if concurrency is suitable for solving the problem.*
2. *Class design: Identify the objects and their responsibilities, and represent these in a UML class diagram.*
3. *Active classes: determine the classes representing active objects (i.e. objects that have their own thread of execution).*
4. *Modelling thread communication: represent the concurrent task in an activity diagram together with the data structures which are used for communication.*
5. *Synchronization: protect shared data from being corrupted by adding appropriate synchronization instructions. If necessary, combine synchronized methods if no context switch is allowed between successive calls.*
6. *Reflection: analyze your solution (does it work as expected?), and reflect on the chosen approach (could we have done better?)*

4 Method

In this chapter the method used in this study is described. This method is applied in the next chapter to a concurrency programming assignment for first year students as described in section 3.2. The first section deals with to the method used to annotate the knowledge that students use in the observations. The first part of the second section describes how macro strategies are found, based on activities described in section 2.5 "Programming knowledge". The second part of the second section describes the method used to analyze the sequence of the knowledge. These sequences give hints to the macro strategy students use.

While working on assignments novices have to combine and use programming concepts and apply theoretical knowledge to practice. To assist novices in these processes different strategies are used. In this method we look at three widely used strategies, *reflection-in-aciton, adaptation strategy,* and *trial-and-error.* Lye & Koh (2014) reviewed different strategies and how they impact different aspects of the learning process of novices. Lye & Koh state that with regular interval there are discrepancies in the approach of students compared to the strategy that was initially intended. Next to this a means to capture a large group of students is missing to find out if the found results are in line with the naturalistic classroom settings, which are still not well-understood. This method provides a structured way to find the strategy that students employ for different parts of the assignment. Knowing if there is a difference lets teachers change their course material to create a better fit towards the strategy they want students to employ.

4.1 Annotating knowledge in recorded data

In this annotation process three activity types are used, *gathering information*, *implementation*, and *assessment* as described in section 2.5. We first describe the reason why we opted for this approach. For each activity type we describe how we find them. The last part of this section describes the case study specific programming concepts that we pay attention to.

4.1.1 Structured approach

In this study we focus on three different activity types, gathering information, implementation, and assessment. Each of these activities need to be annotate in a structured way that can be applied to a variety of programming assignments in the same manner. To accomplish this we took the approach of Izu, Weerasinghe & Pope (2016), described in section 2.6, which describes a structured way to connected pieces of code to abstract programming concepts, which will be used below in the implementation section. A variant of this annotation process can be used for gathering information and assessment which will be explained in the sections below.

4.1.2 Gathering information

Students access external sources to acquire or verify knowledge. In the activity type gathering information we focus on these external sources by directly observing them in the recorded data. To prevent ambiguity a strict list of words is assigned to each programming concept, for example runnable and threads. This list of words contains variants of the programming knowledge concept, for example runnable, run, and runnables. This list helps to increase the coherency between multiple annotaters. Information can come from the computer and information can come from 'real life' sources. The first focusses on websites, pdf files, etc. ('documents') on the computer. The latter focusses on the interaction between people. This means that every time the students speak or search for a word that is related to a programming knowledge concept an annotated block can start. To structure this approach the following rules are used to decide what data to annotate:

Gathering information from the computer:

- An annotated block starts when the students access a:
    - Website: relate name of webpage/main question to programming knowledge concept or assignment. This relation is based on the list of words for each programming knowledge concept.
    - PDF: relate slide to programming knowledge concept or assignment. If not possible relate document name to programming knowledge concept or assignment. The relation is based on the list of words for each programming knowledge concept.
- Document visible for more than 10 seconds
- A new annotation section starts if the student accesses a different website that contains a word on a different programming knowledge word list.
- The annotation section ends if the students leave an external document

Gathering information from 'real life' sources:

- Annotation section starts when, the student or students ask a third person about programming knowledge, or a piece of code they have written based on Table 4 or students access a book, or other offline documents.

- A new annotation section starts if a word from a different programming knowledge word list is used or a different document is accessed
- An annotation section ends when: the document or third party is no longer involved.

External documents or interaction with people are not coded if they do not explicitly state the programming knowledge concept they belong to, this is based on the list of words as described above. If explicit code, based on Table 4, is seen an additional tag “explicit code” is added to the annotation. The time, duration, source (website, pdf, or third person), and the programming knowledge concept that is used in the recordings is annotated. An annotation is also made for every time the assignment document is accessed in the observation.

4.1.3 Implementation

With the implementation activity type we focus on hints of programming knowledge in the writing and deletion of code by students that are directly observable in the recorded data. The programming knowledge concepts have explicit pieces of code that represent the implementation of the concepts in the programming language JAVA, as described in Table 4.

In the annotation process these definitions for each code piece are used to annotate the specific timeframes in which students interact with that piece of code. Every time a student adds, removes or edits (‘implements’) a line of code that involves the explicit pieces of code, the time and duration are annotated. For example, a student writes a line of code that changes the function *run()* of the class taxi. The annotation block start when the student writes the first character and ends when he or she does not write code anymore or starts implementing code in a different function or class. The maximum interruption of the writing is twenty seconds, if the interruption is longer the block ends.

Each annotated block is given tags that describe the part of the code that is changed (‘scope’), and what programming knowledge this change gives hints to. An example of this is a student who edits the *run()* function of the class taxirunner. The annotated data would contain the following tags: implementation, runnable, class_taxirunner, func_run. In case the students only remove a part of the code, the tag “removed” is added. If a student implements code that is not related to any of the programming knowledge as defined in Table 4, the tag “misc” is added.

4.1.4 Assessment

The third activity type describes the assessment of the code by students. The assessment is divided into two different categories, output of code and assessing code. The output for this assignment is printed in the console. Here the students can see:

- The number of passengers each taxi takes with them each trip
- Each time the train 'arrives' with 'new' passengers
- How long the total simulation took.

Assessing code is done by looking at code without changing it. We only annotate data for assessing code if the following points are true:

- Code area is visible for more than 5 seconds and students did not type code for the last 20 seconds in the visible area
- Code is selected

When students look at a new coding segment a new annotation starts. Assessing code uses the same coding style as described in the implementation section, the class and function are added to the tags. If it is uncertain what function a programmer is looking at, because it is not highlighted or multiple functions span the screen, all functions are added to the annotation.

4.1.5 Case study

For this case study we divide the assignment of the students into two parts each with its own goal, the goal of the first part is for students to recognize and construct active classes, the second goal is for students to add synchronization to said active classes. The assignment is split because each part has its own relevant set of programming knowledge which is annotated independently.

Based on the assignment of the students we focus on the following three key programming knowledge concepts: runnable, thread, and synchronization. Each of these concepts represents mandatory programming knowledge that is necessary to complete the assignment. For each of these concepts we use the official documentation of the programming language JAVA. For example variables and functions can only be placed at certain places within the computer code, and they follow a strict format that cannot be deviated from. This method is similar to the study of Izu et al. (2016) were they used code snippets, called building blocks, to determine the correctness and complexity of the assignments of students. However in this study we focus on the key programming concepts described above rather than the ‘smaller’ code pieces as in the study of Izu.

Table 4 Programming knowledge linked to explicit pieces of code as described in the official JAVA documentation.

Programming knowledge	**Explicit code**
Runnable (Oracle, Runnable, 2017)	• Class ‘name’ implements **runnable** • Public/private void **run(){**….**}**
Thread (Oracle, Thread, 2017)	• The creation of a **New thread/runnable** • **Variable Thread.* (start, stop, etc.)**
Synchronization (Oracle, syncmeth, 2017)	• Public/private **synchronized** ‘return type’ ‘function name’ • Function(){… **synchronized(‘value’){**…**}** ….}

First part of the assignment: threads and runnable

The first part of the assignment is about recognizing and constructing active classes. There are two important programming knowledge concepts for the parts runnable and threads as explained in Table 4. The important classes of this part of the assignment are the 'taxi' class and the 'train' class as described in the chapter 3.2 'The assignment of the students'. In Figure 2 the three activity types are represented with the programming knowledge, classes and functions that are important for each activity type.

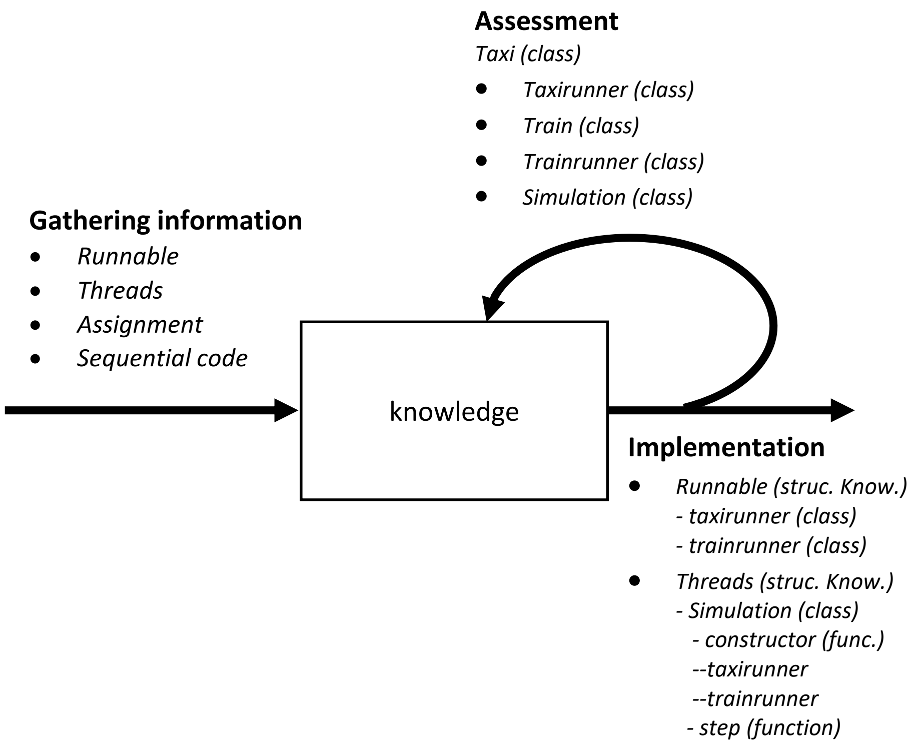


Figure 2 The three activity types combined with the programming knowledge. Shows the expected classes and functions that are interacted with and the programming knowledge that is accessed of external sources.

Second part of the assignment: synchronization

The second part of the assignment is about synchronizing the active classes. There are three important programming knowledge concepts for this part synchronization, check-then-act and shared values as explained in Table 4. There are two possible ways to come up with a solution because of the nature of synchronization called decentralized and centralized as described in section 2.9 'centralized vs decentralized'. In Figure 3 this different approach is shown below the activity type assessment and implementation. The important classes of the decentralized approach are taxi and train, whereas for the centralized solution this is the station class.

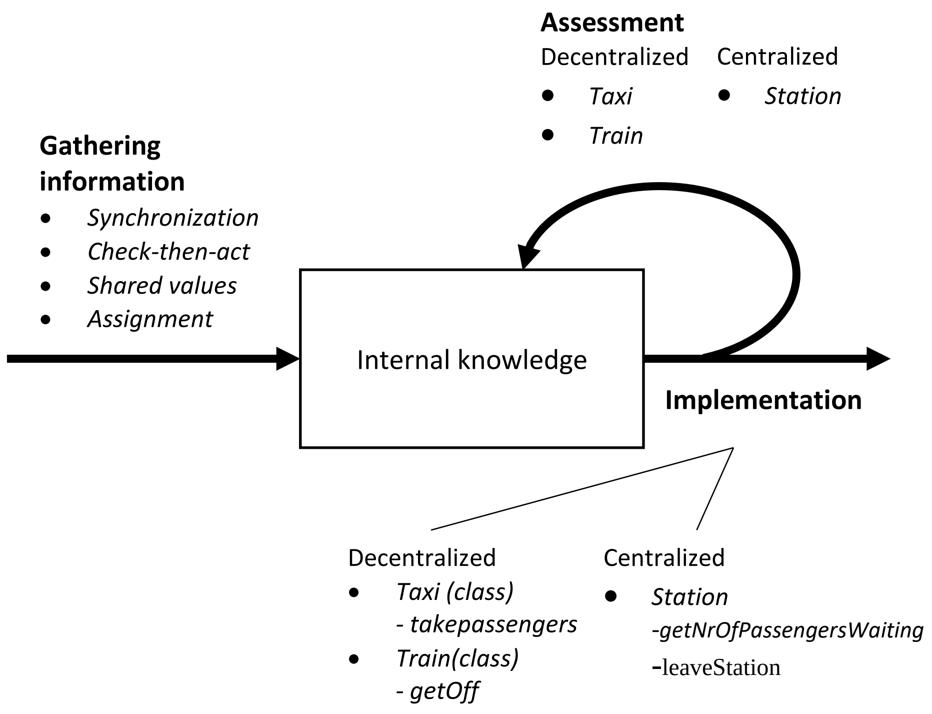


Figure 3 The three activity types combined with the programming knowledge concepts and code snippets.

4.2 Macro strategies and programming concepts

In this section the method to find the macro strategy is described and how this is tied to programming concepts. In section 4.2.1 patterns of the three macro strategies used in this study are explained. Section 4.2.2 shows how we find these patterns.

4.2.1 Macro strategy patterns

In this section an explanation of what patterns each macro strategy has. For each of the three macro strategies: *reflection-in-action*, *adaptation*, and *trial-and-error*, we describe the key characteristics we can observe using the method in the coming sections of this chapter.

As in section 2.5 “Programming knowledge” we pay attention to three activities related to programming: gather information, implementation, and assessment. The goal of this study is to showcase a method that uses these three activities to get hints of the macro strategy students use. Based on literature we describe our interpretation and create example patterns that are related to each of the strategies.

Reflection-in-action

Reflection-in-action, as described by Edwards (2004), differs from trail-and-error since the cycle of writing code and assessing on code is interrupted by longer pauses which may contain gather information sections to acquire new information about a programming concept. Since the students think beforehand how code changes we expect to see almost no to none ‘remove code blocks’ as seen in Figure 4. In the ideal situation students spend more time on assessment and gathering information than implementation, since the students think beforehand what has to change rather than ‘just trying’.

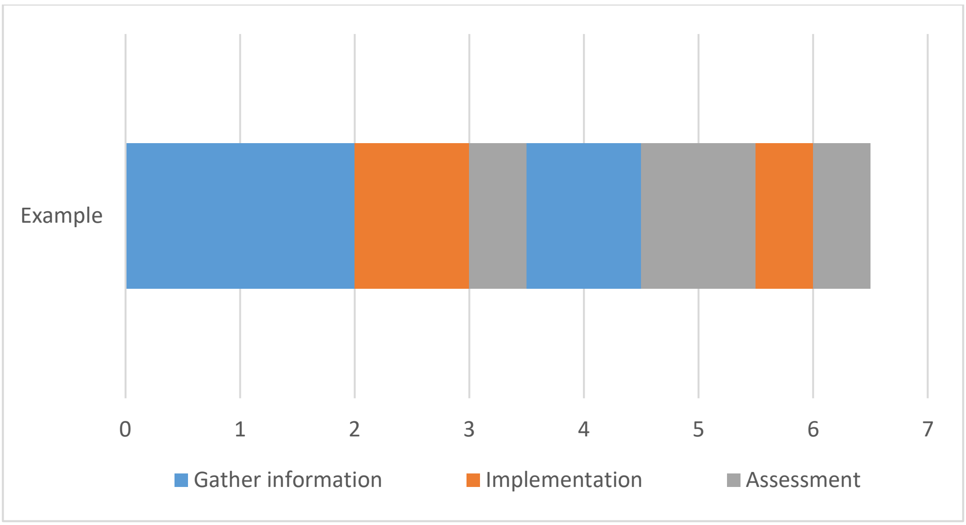


Figure 4 Reflection-in-action. Based on the preference of the group the group starts with gathering information or creates code first. After finding out that the solution is inadequate they gather more information and apply this directly to the code or they assess the code again to see how they have to change it in order to come to a right solution.

Adaptation strategy

The adaptation strategy focusses on finding examples or guidelines from other works and applying them to the situation at hand. Were reflection-in-action focusses on reasoning why code functions as it does, adaptation focusses on trying to find 'explicit' pieces of code that can be used in the solution. In the next section we go into further detail to make these differences clear. The pattern that describes the adaptation pattern is show in Figure 5, first the programmer searches for code examples after which he applies them to his own code. In an ideal situation the found code can be instantly applied to the program at hand, however as seen in Figure 5, it could be that there are remove code parts because students detected that the code does not function properly, after which it is removed or edited to fit the solution better.

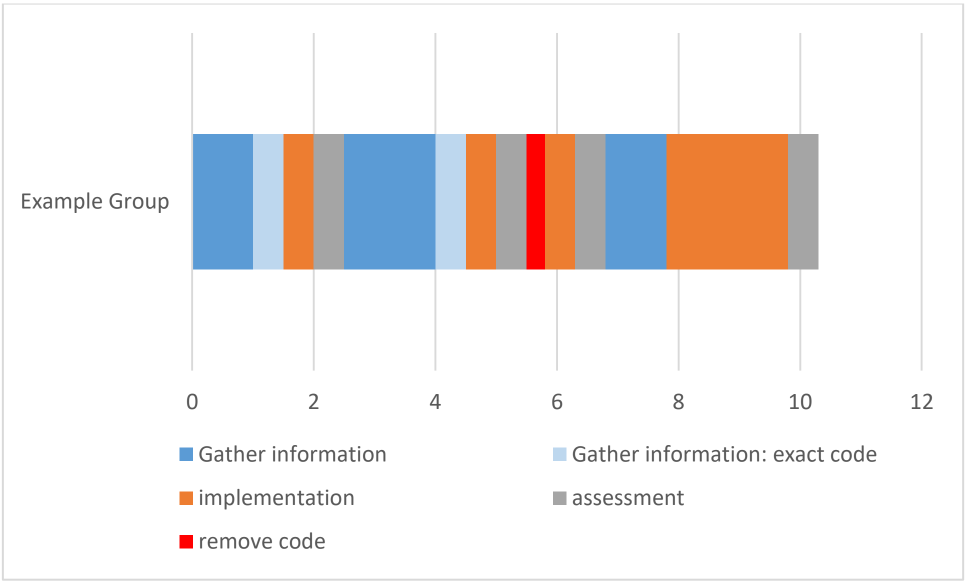


Figure 5 Adaptation strategy pattern. First information is gathered and example code is found, after this the example code is applied to the code base of their own program after which the programmer checks if the implemented solution gives the desired result.

Trial-and-error

Lönnberg, Berglund & Malmi (2009) saw the following pattern in their study that they described as trial-and-error, “code is repeatedly written (trial) and found inadequate (error), until the solution is “good enough”. When translated to a sequence of activities the following would appear as seen in Figure 6. Implementation sections (trial) followed by assessment (error) sections after which the students remove code or add code again. Remove code sections are a sub activity type of the implementation activity were code is removed rather than added which will be explained in detail in the next section.

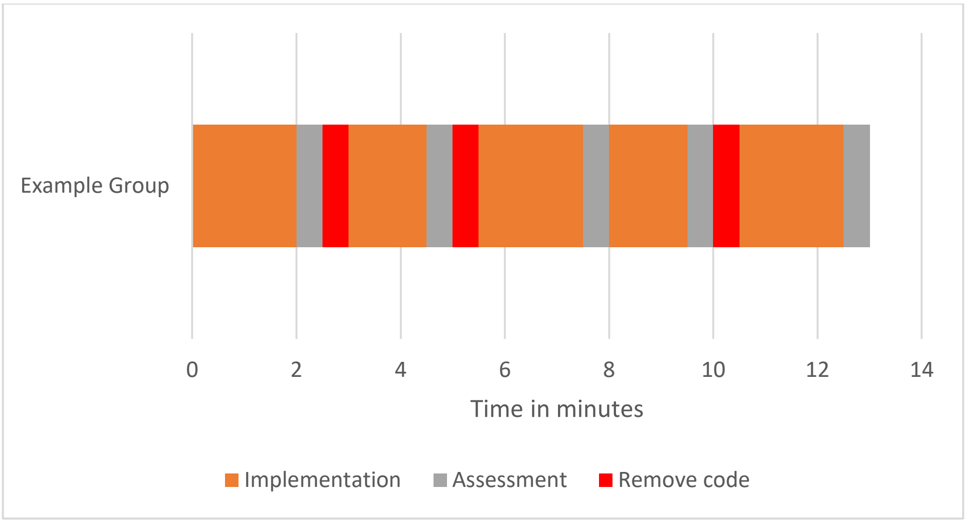


Figure 6 Trial-and-error example. A repeating pattern of Implementation activities followed by assessment, and possibly followed by removing code again without any or almost no gather information sections.

There are two varieties of trial-and-error that we pay attention to, the first is ‘centralized’ trial-and-error were the students rewrite code in one particular function, changing small parts of the code. The second variant is that code is written in a wide variety of functions. This difference can help to determine the cause of this approach, were a centralized approach can suggest that the students know where to change code, but not how, and a decentralized approach could mean that they do not know where they need to change code, and possibly also not what they have to change.

Combination of strategies

Multiple of the strategies above can be used within one assignment. For example a student can start by finding an example on the internet but when he tries to apply it to his own program it fails. The student then starts a trial-and-error strategy to make the code work.

4.2.2 Finding Macro strategy per programming concept

This section focusses on the sequence of building blocks used by students and how this gives hints to the macro strategy used. As mentioned before, the goals are to study what macro strategies students use, and what knowledge is used in these strategies.

In this study we divide the plan the students have to make in three pieces. These pieces are represented by the three programming knowledge concepts as defined in Table 4. By combining the pieces, as described in the start of this section, with the sequence of activity types: gathering information, implementation, and assessment, we can find hints to the way macro strategies are used by the students. While studying the strategy we focus on finding reoccurring patterns, or the lack of this. An example of a reoccurring pattern is: multiple iterations of an implementation block followed by an assessment block followed again by an implementation block. The example gives us hints that a trial and error strategy is used. After this, the sequences of each programming knowledge concept are compared to the patterns of the other programming knowledge concepts to find overlapping patterns or differences between them. The goal of this comparison is to find hints of reoccurring patterns that are overarching and consequently giving hints of possible strategies that are used. The sequences are extracted from the recorded data based on the tags assigned to the annotation blocks from section 4.1.

Start and end of a sequence

Each programming knowledge concept has its own sequences of activity types. In this case study we did not interview the students afterwards, this means that we could not ask the students when they started to work on a programming concept. We therefore resorted to the first time a programming knowledge concept tag is used, since this is the first time the students showed they were working on said programming concept. A sequence ends if the next annotated code block of the observation contains one of the other programming knowledge concept tags. Multiple sequences for one programming knowledge concept are combined into one sequence for the analyses. The end result is one long sequence of multiple annotated code blocks.

Macro strategy per programming knowledge concept

The sequence of each programming knowledge concept separately tells us something about the macro strategy students use per programming knowledge type. For example, some groups might go for a trial and error strategy, characterized by a reoccurring pattern of implementing code, assessing the code, and deleting parts of the code if it did not give the expected results (Lönnberg, Berglund, & Malmi, 2009). For this study this would mean that the activity type implementation is followed by the assessment type after which an implementation type follows again. Other groups create plans by gathering information of examples they find on the internet or in the slides they received from the lectures of the course they are following (Lönnberg, Berglund, & Malmi, 2009). This approach can be seen when the activity type gathering information comes before an implementation type, after which the student does not remove code from the previous implementation. According to literature, trial and error behavior is one of the most common approaches to create plans. In particular this strategy is often applied when a person has little experience in a subject. In this study we are interested to see if this pattern is also visible or if different patterns occur.

Overlap and differences of found macro strategies

The macro strategy of each programming knowledge concepts are compared with each other. The focus of the comparison is on overlapping patterns of the activity types and differences between them. The similarities and differences tell us two things:

1. Similarities between different programming knowledge suggest an overarching strategy the students use for all different parts of the plan
2. Differences give us hints to a different approach per piece of the plan.

Possible explanations for these similarities and differences found in literature are: students have a better understanding of some programming knowledge concepts, students found examples that they could adjust to the new situation, or problems combining different programming knowledge concepts (Lönnberg, Berglund, & Malmi, 2009) (Whalley J. L., 2006).

To further aid the findings of these patterns we pay attention to:

- the total duration students spend on the different programming knowledge concepts

  The time students spend on a programming knowledge concept relative to other programming knowledge concepts can be a possible explanation why certain approaches to plans are used.
- the average duration of annotation blocks
- the amount of annotation blocks per programming knowledge concept

  the average time students spend on annotation blocks and the amount of annotation blocks can give hints to the strategy students used to come up with plans
- the order in which students work on programming knowledge types, the sequence or mixture of programming knowledge types

  The order or mixture of programming knowledge types tells us something about the way the different pieces, programming knowledge concepts, are combined. By doing so this order gives us hints to the approach of different pieces of plans.

5 Results

This chapter describes the results of the case study obtained during this research, using the method described in chapter 4. Section 5.1 describes the programming knowledge students use while working on the concurrency assignment. In section 5.2 the macro strategy of the student is presented and found patterns in the approach are described.

5.1 Programming knowledge

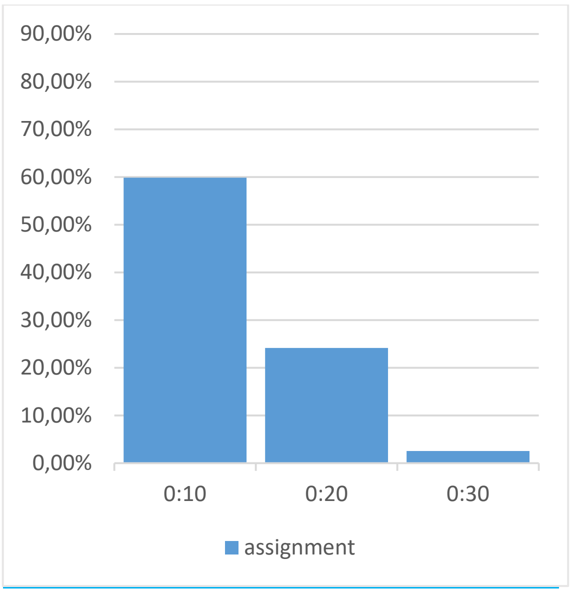
This section contains the results of the analyses of the recorded data. The assignment has two goals each with their own set of programming knowledge concepts. The two goals each have their own section called ‘recognizing and constructing active classes’ which uses the programming knowledge concepts runnable and threads, and ‘synchronizing concurrency tasks’ which describes the programming knowledge concept synchronization and the check-then-act pattern.

5.1.1 Runnable and threads

In this section the results of the annotated data of students while working on recognizing and constructing active classes goal of the assignment is shown. As described in section 0, the important classes for this section are the taxi class and the train class. Both require an active class that allows them to be executed in a thread. Next to this the *step*(), and *constructor()* function of simulation needs to be edited to accommodate this new behavior.

Gathering information

Both group 1 and 2 started with gathering information about the assignment, see Figure 7 and Figure 8 showing the information group 1 and 2 gathered respectively in the first thirty minutes that they spend on the assignment. The annotated data shows that group 1 spend around 9 minutes, of the 30 minutes, gathering information from the assignment. Group 2 spend around 8 minutes gathering information of the assignment. Group 2 also spends around 10 minutes gathering information about programming knowledge runnable. The topics the students gathered information about were in line with the example solution. There are two sources students rely on for the programming concepts runnable: stackoverflow.com and slides of the lectures. Group 1 did not gather information about runnable or threads from external sources, group 2 gathered information about this from external sources.



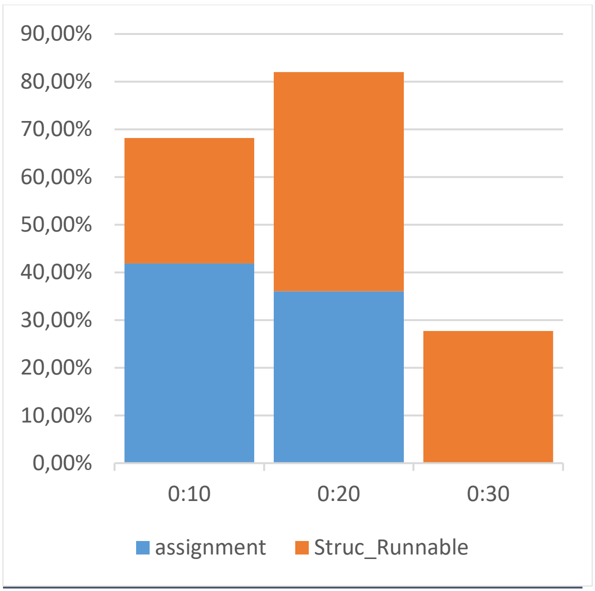


Figure 7 Gathering information group 1 of first 30 min.

Figure 8 Gathering information group 2 first 30 min

Implementation

Group 1 started with implementing the class taxirunner, which is the active class for taxi that implements runnable. Group 1 spend around 1 minute and 20 seconds on this. When implementing the taxirunner they added code in one part.

After this group 1 started to add threads to the *step*() function of the simulation class. They spend around 5 minutes and 30 seconds on this. This time is spend in five different parts. The first part, 4 minutes and 12 seconds, was significantly longer than the other four parts.

Group 2 started with implementing the class taxirunner as well. They spend around 2 minutes and 30 seconds on this. The implementation of runnable is split up in three parts.

Group 2 spend around 1 minute and 50 seconds, in five parts, on adding threads. In the first two parts they applied and removed threads to the *constructor()* function of simulation. They spend 40 seconds and 19 seconds on this respectively. After this they added threads for the taxirunner to the *step()* function of simulation. They first added threads 30 seconds, then they edited code twice, without a part that only removed code. The implementation of both groups are in line with the assignment as given in section 3.2.

Assessment

Group 1 and 2 had different approaches to assess their program. As seen in Figure 9, group 1 mainly looked at the output of the program, a total of 4 minutes and 10 seconds on this. Next to this they 24 seconds looking at the function *calcTotalTime* of station, after which they looked at the output again. They also looked at the function *takepassengers* from Taxi for about 1 minute and 30 seconds.

Group 2 showed less time spend on the output of the program and more on the functions of the program. As seen in Figure 10, group 2 started with looking at different functions of the sequential program. After the first 10 minutes they only showed assessments of the *takeapassengers* function of taxi.

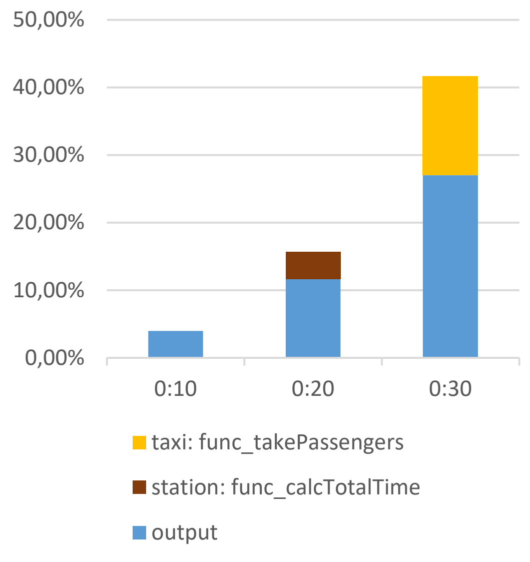


Figure 9 The way group 1 assess their implementation.

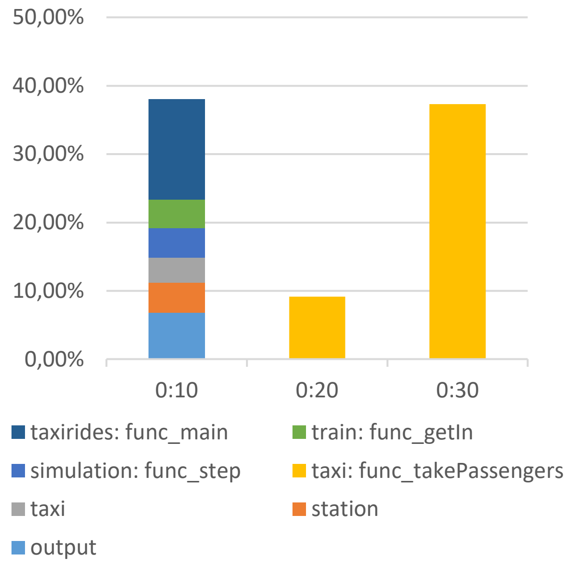


Figure 10 The way group 2 assess their implementation

Both group 1 and 2 spend time assessing the functions of the class station. This is a discrepancy between the found results and the example solution.

5.1.2 Synchronization

The second part of the assignment focusses on the synchronization of the shared resource, the persons waiting at the station. Like the previous section the focus is on the similarities and differences between both groups of students. Both groups give hints that they are working on a decentralized solution, after an intervention of a student assistant one group 'switched' to a centralized solution.

Gathering information

Both groups gathered information about the programming knowledge check-then-act. The source they gather this information from are the slides from the course OO, with the topic check-then-act. However we can see a clear difference in the amount of time both groups spend on this. The first group spend only a few minutes on this topic, as seen in Figure 11. We can see that time spend on the check-then-act pattern ('Struc_checkThenAct') is around 3 minutes. However if we look at the time spend of group 2, Figure 12, this is around 11 minutes. As in the first part, group 1 spend less time on gathering information than group 2. At the 1:50 mark we can see that group 2 gathered information from a third person, a student assistant. While not all audio was captured of this conversation, we could see that they were looking at the station class, in particular the functions *getNumberOfPassengers* and *leaveStation*. These are the two functions that need to be combined, following the check-then-act pattern, for a centralized synchronization solution.

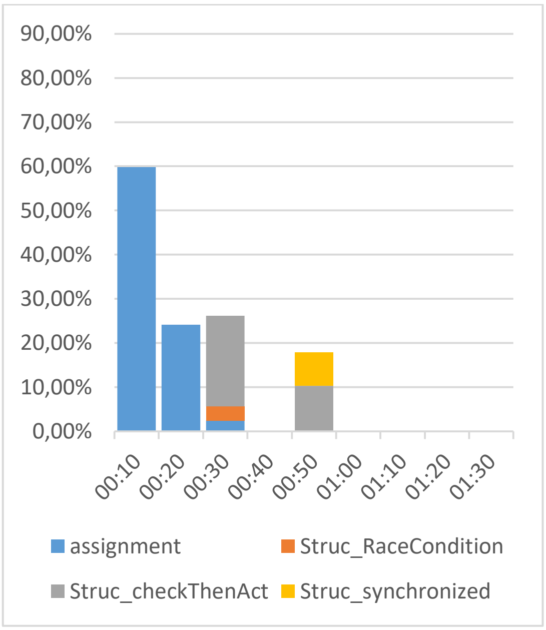


Figure 11 Gathering information by group one. Time spend looking into different programming knowledge concepts

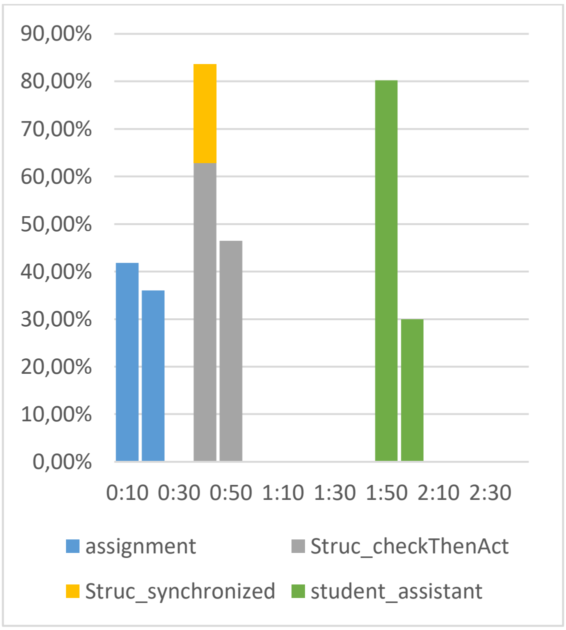


Figure 12 Gathering information by group 2. Time spend looking into different programming knowledge concepts

Implementation

First the observations from group 1 regarding synchronization are shown. As seen in Figure 13, group 1 focused mainly on adding, editing, and removing synchronization. The first minute group 1 changed code that did not belong to any of the programming knowledge concepts as defined, therefore this is under miscellaneous ('misc'). As seen Figure 15, group 1 first focused on applying synchronization to the *step()* function and the taxi. After this they applied synchronization to almost every function of both taxi, and station, at one point both taxi and station almost only had synchronized functions. At the 1:00 mark the students removed all synchronizations they added to the program and focused on the *takePassengers* function of taxi, again reverting all the changes made in the end. Between 1:20 and 1:30 they focused on the station class and added synchronization to five functions, *isClosed, leaveStation, close, enterStation, and getNrOfPassengersWaiting.* However as described in section 0 , for a centralized solution, getNRofPassengersWaiting and leaveStation need to be combined.

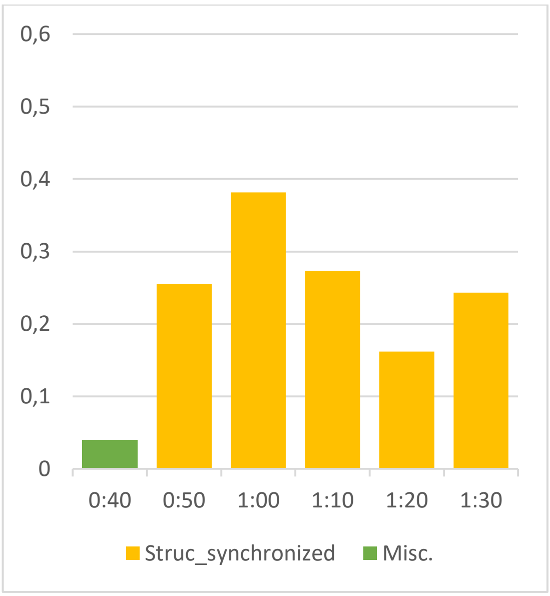


Figure 13 Implementation section of group 1 regarding synchronization. Showing the programming knowledge group 1 used.

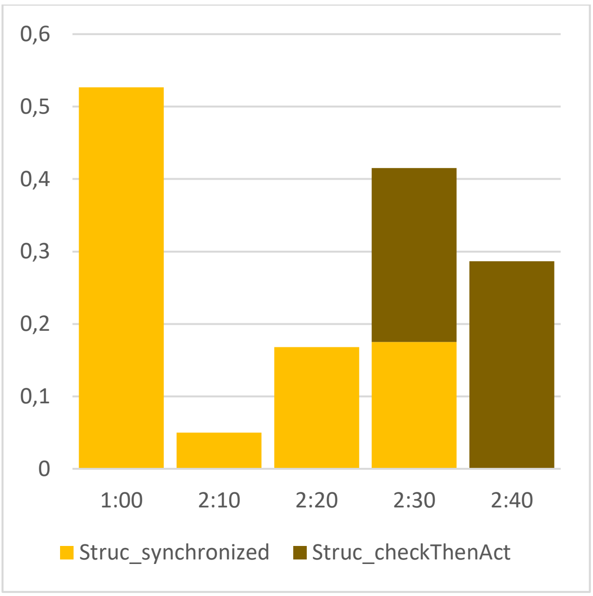
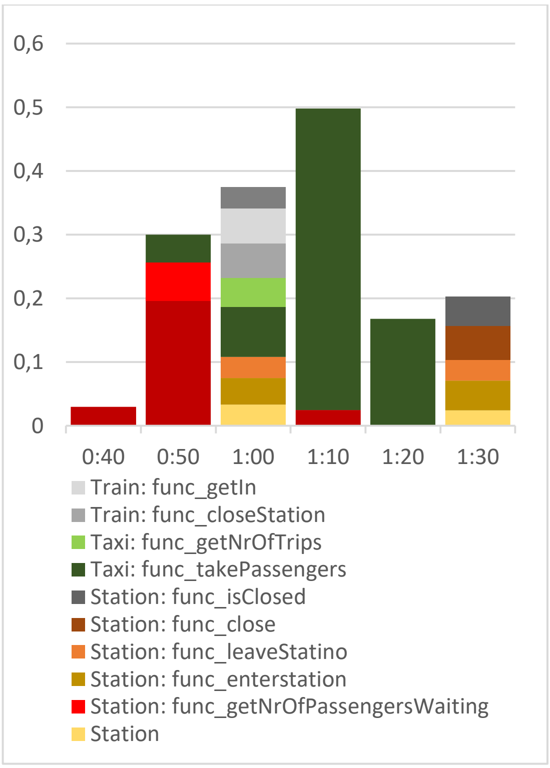


Figure 14 Implementation section of group 2 regarding synchronization. Showing the programming knowledge group 2 used.



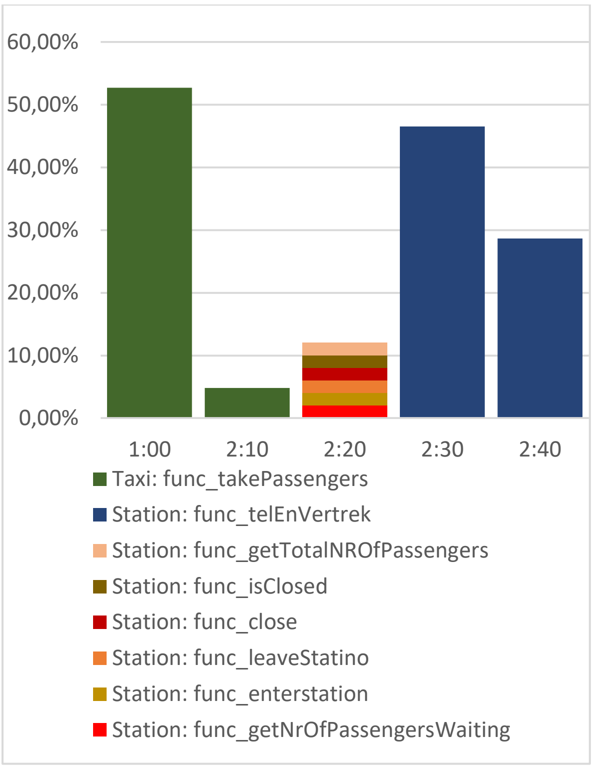


Figure 15 Implementation section of group 1 regarding synchronization. Showing the functions that group 1 changed

Figure 16 Implementation section of group 2 regarding synchronization. Showing the functions that group 2 changed

We now take a look at the implementation section of group 2. Like group 1 all implementations that they do involve synchronization, as seen in Figure 14. The programming pattern check-then-act is shown in the last 20 minutes (2:30-2:40), this is a specific form of synchronization. Group 2 started with implementing synchronized in the *takePassengers* function of the taxi, after adding and removing synchronized a few times they ended up removing all code they implemented here. They then switched over to apply synchronized to all functions of the station, and removing them again after they looked at the output of the application, this will be explained further down in this section. As stated in the gathering information part above, the students ended up applying the check-then-act pattern to a new function they made 'telEnVertrek', translated to countAndLeave. As the name can suggest, group 2 combined the functions of *getNrOfPassengersWaiting*, and *leaveStation* to be called from one synchronized function. This way the 'correct' amount of people will be on the station and moved by the taxi's. This observation also contains some differences compared to the expected 'centralized' solution. The observations did not contain group 2 editing the *takePassengers* function of the taxi class. Here the call to *getNrOfPassengersWaiting*, and *leaveStation* need to be combined to *telEnVertrek*. Also the function *enterStation* needs to be synchronized, since this function also edits the shared value.

Assessment

The assessment section of the synchronization of group 1 is visualized in Figure 17. Group 1 spend around 60 per cent of their assessment time on the output of the application, this totals out to almost 14 minutes. The percentages group 1 spend on assessing functions is mainly weighted towards two functions, *step() in simulation* and *takePassengers* in taxi, spending around 4 for *step* and 3 minutes for *takePassengers*. Group 1 spend between 30 seconds to 1 minutes and 30 seconds on the other functions each.

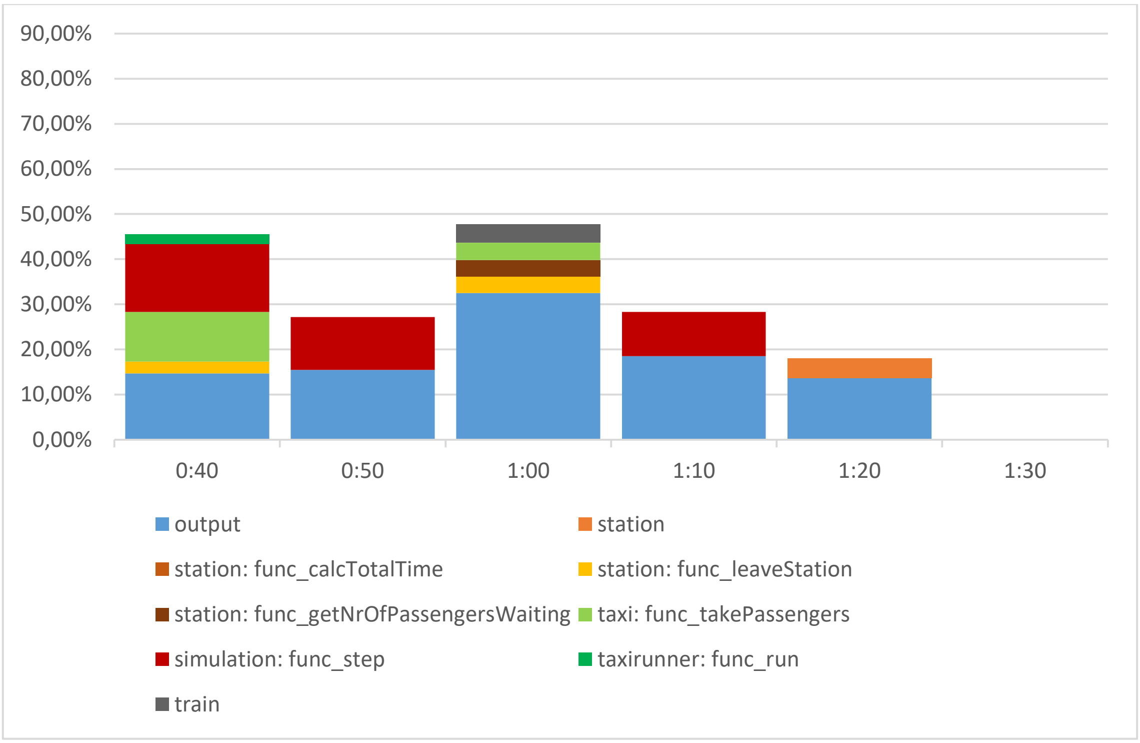


Figure 17 Assessment of synchronization part, group 1.

Group 2 shows a different pattern compared to group 1. Figure 18 visualizes the time group 2 spend on different parts of the assessment. Take note that the time shifts from 1:10 to 1:50, this is because the students took a break. Almost 70 per cent of the observed assessments is group 2 assessing the function *takePassengers* from the class taxi, this is around 25 minutes. At 1:50 they spend one big block of 6 minutes on assessing the *step()* function of simulation. They spend around 1 minute on assessing the functions *leaveStation* and *getNrOfPassengersWaiting* of the station class. Compared to group 1 they spend a short time assessing the output of their program, less than 1 minute.

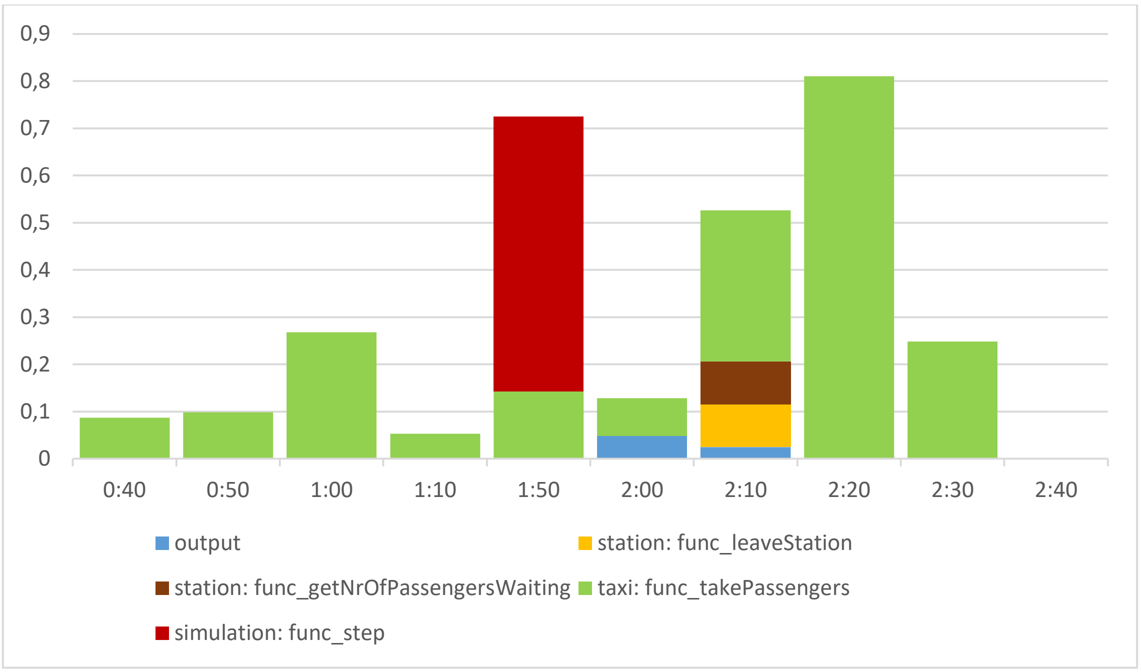


Figure 18 Assessment of synchronization part, group 2.

5.2 Macro strategy per programming concept

In this section the sequence of the three programming knowledge concepts are described. For each programming knowledge concept we look at the similarities and discrepancies between the two observed student groups. After this the sequence of the three programming knowledge concepts per group is compared. The programming knowledge concepts are ordered on chronological order, runnable, threads, and as last synchronization.

5.2.1 Runnable

The first programming knowledge concept of the observation is runnable. In Figure 19 the sequence of group 1 and 2 are shown. The blue blocks represent the information group 2 gathered for runnable. The yellow represents the time group 1 and 2 spend on implementing the taxiRunner, the class that implements runnable. Group 1 had only one implementation block, while group 2 had two implementation blocks, surrounded by gathering information blocks. Before the first implementation block exact code was seen in the gather information block, based on Table 4. The total time both groups spend on implementing runnable is around the same.

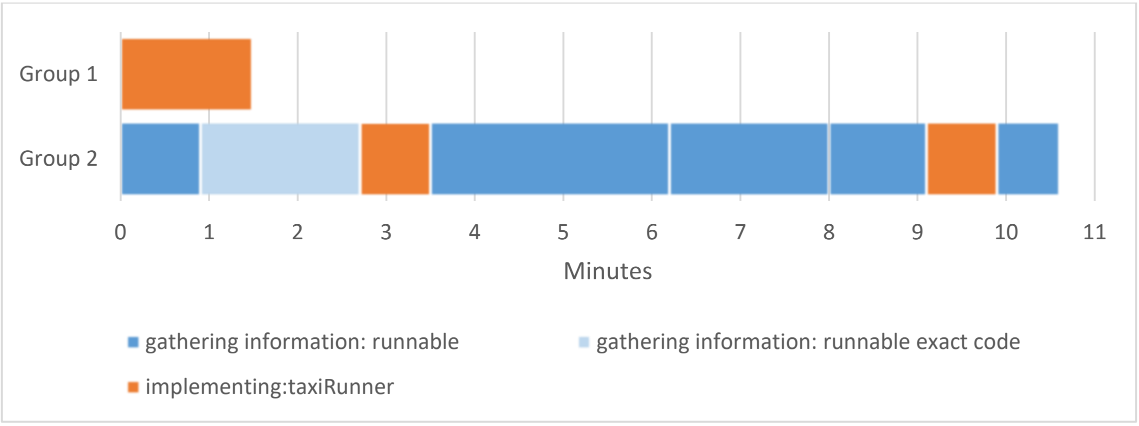


Figure 19 Sequence: Programming knowledge concept Runnable

5.2.2 Threads

The sequence of the second programming knowledge concept, threads, is shown in Figure 20. The yellow blocks represent the time the groups spend on implementing threads, both did this in the step function. The grey tints represent the assessment blocks, light grey represents the time students spend on looking at the output, the two types of dark grey represent the time students spend assessing functions. The blocks with blue tints represent the information students gathered, and lastly the red blocks represent the time students spend on removing code.

Both groups started with implementing threads into the step function. After this both groups assessed the output of the application, and group 2 assessed the function *takePassengers*. After the assessing blocks both groups removed code, had another assessment block, group 1 used the output, group 2 assessed *takePassengers,* another implementation block. Group 2 did not look at the output of the program after the second implementation section. Group 1 did look at the output, and removed code from the step function. After this a sequence of gathering information and assessment of code started, ending with a small implementation block, followed by assessing the output of the program.

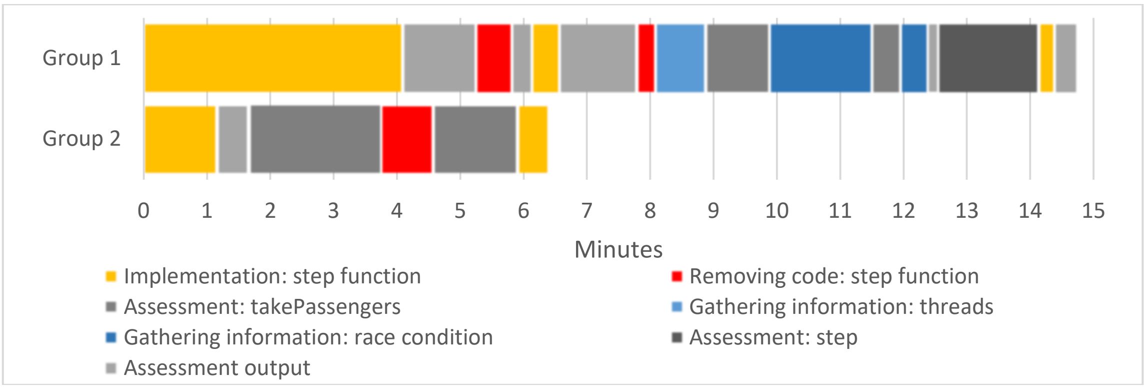


Figure 20 Programming knowledge concept threads. Time spend by groups in minutes.

5.2.3 Synchronization

The sequences of synchronization are described here. First a comparison is made of the global sequence, after which the sequence of the two groups are described. The blocks are described using the overarching tags, implementation synchronization or check-then-act, this is done because of the vast number of different functions both groups changed, the functions that are changed will be described.

As seen in Figure 21, and looking at the sequence as a whole the two groups show a ‘wildly’ different sequence compared to each other. To start with the two main differences between group 1 and 2, first the average ‘size’ of the blocks is very different, and the second difference is that group 1 has a much higher density of blocks compared to group 2. When looking at the patterns of the sequences, similar patterns can be found, implementation blocks are followed by a remove code block or an assessment block.

Most of the blocks of group 1 are around 1 minute or less, the exceptions are the gather information blocks and some assessment blocks. A slight deviation of yellow is used to show the difference of implementations blocks that follow each other.

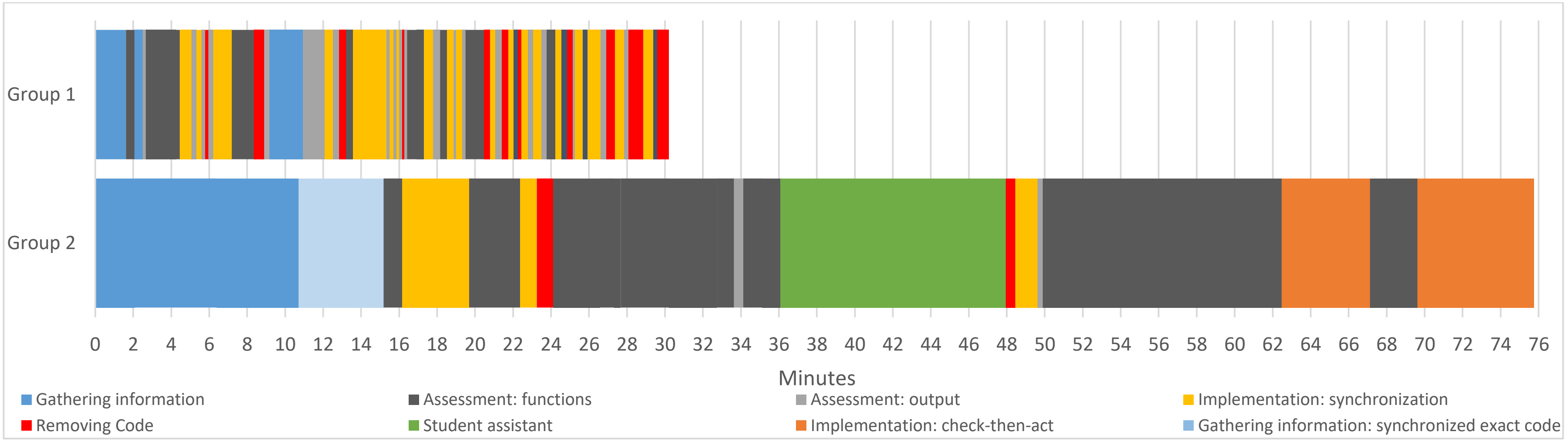


Figure 21 Sequence of the synchronization programming knowledge concepts of both groups. Slightly darker and brighter colors are used to show different blocks of the same type that are next to each other.

Group 1 started with gathering information about the programming knowledge concept synchronization. After which they assessed different functions, *step, takePassengers,* and multiple functions of *station.* After this the group started to apply synchronization to the *takePassengers* function, after which they removed the added code again. They gathered more information about the check-then-act pattern and started to implement synchronization in all the functions of the *station* class. After adding synchronization to all the functions of *station* they assessed the output, and implemented synchronization to the *step* function and to *takePassengers*. After this group 1 shows a pattern where they added synchronization to a part of the program, assessed the output of the program, and removed the synchronization they added. They did this for around 10 minutes, ending up removing all the synchronizations they added, this can be seen from the 20 minute mark to the 30 minute mark, after which the observation stopped.

Group 2 shows a different sequence. As seen in Figure 21, they spend the first 15 minutes gathering information about synchronization, and the check-then-act pattern. Before implementing code group 2 gathered information about code for synchronization. The first two implementation blocks focus on the *takePassengers* function. In the last two implementation blocks group 2 created a new function that combines the functionality of *getNrOfPassengersWaiting* and *leaveStation*, and they applied synchronization to this new function. They started implementing this after they spoke with a student assistant, while talking to the student assistant they looked at the functions *step* and *takePassengers*, however we could not hear what the student assistant or the students said due to the bad audio quality. Therefor it remains unclear what the student assistant said to help the students, this could range from him explaining certain terminology to giving the students the answer to their problem.

5.2.4 Comparing sequences per group

In this section the three programming knowledge concepts, per group, are compared. Key aspects that are compared are: total spend time, patterns, amount of blocks, and average time spend on blocks.

Total time spend

Group 1 started with a 'quick' implementation of runnable (1:30 minutes), the observed time spend on threads is 15 minutes and synchronization 30 minutes.

Group 2 spend more time on runnable (10:30 minutes) than on threads (6:30 minutes), However the biggest part of time spend is on synchronization with 76 minutes.

The total time of the observation of group 1 was 1:38:57, this time includes a 30 minute break in the middle and 20 minutes inactivity at the end. Taking this time into account around 50 minutes are left that could be annotated. For group 1 this means that over 90 per cent of the observation is annotated.

The total time of the observation of group 2 is 2:38:08, this time includes a 45 minute break in the middle and 5 minutes of inactivity at the end. Taking this into account around 100 minutes of the observation could be annotated. This means that for group 2 also around 90 per cent is annotated.

Patterns

Group 1 did not show any consistent iterations of different types of blocks for runnable and threads. However for the programming knowledge concept synchronization they show patterns of implementation, assessment, remove code, which repeats in rapid succession. This pattern can be seen as trail-and-error behavior (Kolikant, 2005) (Lönnberg, Berglund, & Malmi, 2009) (Spohrer & Soloway, 1986). This pattern is also visible at the start of the programming knowledge concept threads. The first implementation blocks at the start of the timelines, of each programming knowledge concept, are longer than those at the end.

Group 2 shows a sequence of gathering information and implementation for runnable. When looking at threads and synchronization this sequence is no longer seen. Group 2 has multiple 'same' block sequences, gathering information from multiple sources, or assessing different functions after each other.

Group 2 give hints to a possible second pattern that can be seen in the concepts runnable and synchronization. This pattern consists of gathering information blocks followed by an implementation block. This can be seen twice for both concepts.

Amount of blocks and average time spend

Group 1 and 2 show differences when looking at the amount of blocks and the duration of the blocks. On average group 2 spends three times the amount of time on gathering information, and assessment blocks. When looking at the implementation blocks the average time is around the same. Group 1 shows an increase in blocks for each programming knowledge concept. Group 2 has the around the same blocks for runnable and threads, and an increase in blocks for synchronization.

Order of programming knowledge types

Group 1 and 2 both show a similar order in their approach of the assignment. The annotated data shows a clear transition between the programming knowledge concepts. The order that both groups worked on the assignment is as follows, first the students worked on the concept runnable, after this the concept threads, and lastly the concept synchronization. There is no annotated data that shows that students used previous concepts once they transitioned to the next concept.

6 Discussion

This chapter contains the discussion points regarding this study. The first part of the discussion focuses on the case study, where we take a look at the data and what this tells about this specific case. The second part of the chapter focuses on the method used, here we look back at the method and see how it holds up to the expectations and how the method can be improved upon.

The first section focusses on the assignment and how the students complied with the guidelines in it. In the second section we discuss what levels of the SOLO taxonomy we can see of the works of students. The third section contains possible reasons why certain parts of the assignment are difficult for the students and how this can be improved upon using the SOLO taxonomy. The fourth section focuses on the method used in this study.

6.1 SOLO taxonomy

In this section we look at the results of the case study from a different angle, with the SOLO taxonomy, for this we use the works of Izu, Weerasinghe & Pope (2016) further described in section 2.4. To complete the case study assignment students have to integrate multiple programming concepts/aspects into a structure, this is the relational level of SOLO. The assignment is not extended abstract since students do not have to create a program from scratch, but rather apply knowledge they know within a certain context already available. While we can classify the whole assignment in one go we can also classify each programming concept separately. For this we use two sources, Table 1 “Algorithmic design” will be described as AD ‘level’ and Table 2 “Code design” will be described as CD ‘level’. In our case study we look how students apply building blocks to the existing code, and in which areas they apply the building blocks. These observations can be used to see several competences associated with the different levels of SOLO: applying, analyzing, comparing, describe, integrate and combining. Students need to be able to integrate each concept separately to the assignment after which we can look at the whole assignment to if the students combined all the concepts in the right way.

Runnable

Both groups adapted the runnable pattern to the situation described at hand. They created a new class to fulfill the goal and integrated this into the current solution. For this both groups had to comprehend the structure of the original code, regarding the taxi class, and how the new code could be applied to this. The patterns were applied to the correct areas of the code, showing the AD multistructural knowledge of the concept runnable and a CD relational level.

Threads

Both groups apply threads to the step function, which makes it so that a new thread is created every time the step function is called. They also show an understanding of the sequential code that the thread replaces. This shows that the students also have a AD multistructural understanding of threads and a CD relational level.

Synchronization

The programming concept 'synchronization' had a different outcome per group, which is why we look at both groups separately. Group 1 applied synchronization in different areas, not always the correct area, however the concept itself and the building block belonging to it was applied correctly. No 'advanced' patterns like check-then-act were detected; the maximum AD level detected was unistructural, since group 1 only applied a direct translation of the specifications rather than adapting it to the situation at hand. When looking at the CD level we see that the students used the 'default' implementation on a seemingly random structure which also indicates a unistructural level.

Group 2 ended up with combining a few building blocks, however not a complete integration was made which indicates a CD rational level. These building blocks were applied on the right areas which shows an AD multistructural level.

Complete assignment

Group 1 managed to combine multiple building blocks, however they did not integrate the concept synchronization in the solution. This means that they did not come up with a valid well-structured solution and end up with an AD multistructural level for the complete assignment.

If we look at group 2 we see the same before the intervention of the student assistant. They did not manage to integrate the concept 'synchronization'. However after the intervention they do manage this. They applied synchronization to the right area and started to integrate this in the solution. Group 2 ended up with AD level ‘relational’, which means that they can come up with a valid well-structured solution.

6.2 Reflecting on the case study

In this section we discuss how we think that the method helps teachers to improve the assignment. We start out with interpreting the data collected after which we come with possible improvements to the assignment from a teachers perspective. In section 6.4 we reflect on the method itself.

The first step we take is to determine which concepts the student do and do not understand. We start by finding the code that is tied to a specific programming concept in the assignment of the student. The results of section 5.1 are used for this. In this case study the concepts ‘runnable’ and ‘threads’ are used in correct areas by both groups, however synchronization is not. By looking at the results described in section 4.2.2 “Macro strategy per programming concept”, we can see that both groups used a different strategy for the programming concept synchronization. Group 1 mainly used a trial-and-error strategy while group 2 shows hints of the adaptation strategy. By looking at the programming activities, the grouped building blocks, we can see that the taxi class was the main focus of the change for both groups. However both groups also applied synchronization to functions that had no influence on the workings of the program. This behavior can possibly be explained by:

1. A lack of understanding of the concept by the students
2. Students understand the concept but it is unclear for the students that the passengers ‘waiting’ in the station class is the crucial variable in the case of synchronization.
3. Students re-evaluate from the end result back to come up with a solution, leading them to the taxi class that takes passengers. Here the students ‘stop’ looking further.

We now look which of these points are more likely to be a possible cause for the students to not finish this assignment correctly, for this we use the SOLO taxonomy (Biggs & Collis, 2014). The students have shown that they already know a way to implement synchronization, however this is done in the wrong spot, which makes ‘a lack of understanding’ unlikely. Described in the terms of the SOLO taxonomy, the students are able to identify that synchronization is needed (Unistructural), and that they can implement the code at different locations that are related to each other (Multistructural) (Lister, Simon, Thompson, Whalley, &

Prasad, 2006). However students do not show that they can apply, relate, or justify the generated code in a correct manner (Relational).

With this in mind we can now look for possibilities to assist the students. There are two ways we can approach the problem, we can try to come up with possible solutions and to adjust the macro strategy of the students to fit this solution, or we can use the strategy of students as a starting point and create a solution per strategy. Both of these solutions need to take into account that the students are potentially not at the 'relational' level.

To make sure that the students understand which value is important and which functions are related to this we suggest that it is mandatory for students to first answer a few questions before they are allowed to work on the assignment. The focus of these questions is to trigger a reflection-in-action strategy were the students have to analyze, relate, justify and reason, rather than only applying knowledge. We think that this would assist the student in case of point 2 and 3 in the list above. The first question should point the students in the direction of the shared value, the passengers waiting at the station, with the expectation that the student already understands that there are multiple taxi's that change a single value. This question should help as a starting point to show a central building block to the students in a unistructural way. After this the functions that interact with the shared values need to be 'exposed' by the students. This has as goal to make students understand were to apply the synchronization. This first question has as focus to extend the building block to the functions that are connected to it and to combine these blocks. The last question focusses on the 'check-then-act' pattern and should force the students to think beforehand which functions need to be combined, to make sure that data cannot be manipulated by multiple taxi's at the same time. The last question should also assist in pointing students to the direction of relating the different blocks in the way that the student can reason why they need to add synchronization; creating a relational level of understanding. One could say that asking these questions reduces the complexity of the problem. However following the logic of the SOLO taxonomy relating, justifying, and analyzing the problem all exist in the relational level of SOLO, which is the same as applying knowledge to the problem, and should therefore not reduce the complexity but rather approach it from a different angle.

Since the students already show that they understand the individual building blocks we think that the questions above provide different angles to help the students to understand which links they are missing in order to finish the assignment.

6.3 Macro strategy fingerprint

While processing the data for this study we found a potentially interesting way to look at the time students spend on an assignment. For this we looked at the time students spend on the three different programming activities: gathering information, implementation, and assessment. In Figure 22 and Figure 23, time spend by group 1 and 2 respectivly on programming activities, we can see that the total time spend is different for both groups. One of the things we noticed is that the time spend on the activity assessment is almost the same, while the time spend on implementation and gathering information is almost inversed.

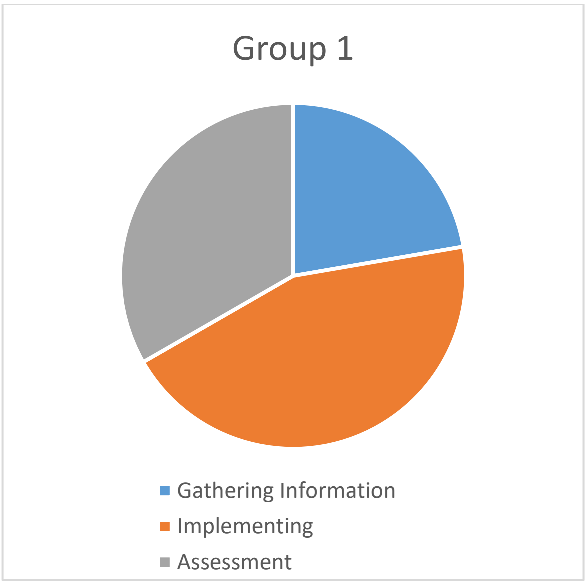


Figure 22 Total time spend on each programming activity, group one

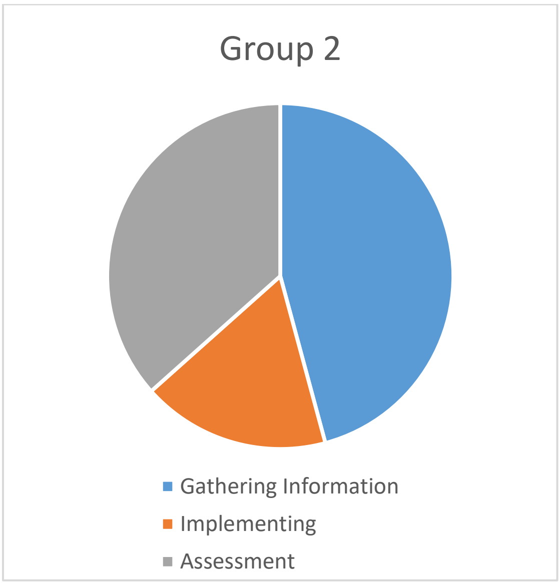


Figure 23 Total time spend on each programming activity, group two

While group 1 showed indications of using an overall strategy of trial-and-error, group 2 mainly showed indications of an adaptation strategy combined with reflection-in-action. It is possible that these differences are seen back in the 'fingerprint', the percentage in time spend in comparison with the total time spend by students. This could help future studies in finding the most likely strategy used by students. Another upside of these fingerprints is that they can be easily compared between assignments of the same students and they can be used to find similarities and differences between different students.

6.4 Reflecting on the method

In this section we discuss the pros and cons of the method suggested in this study. We pay attention to how detailed the method can tell us something about the strategy students use, and with what ease we think this method can be applied to new situations. While we reflect on the method, we keep the goal and sub goals of this study in mind.

The first point we discuss is how detailed the method can describe the macro strategies used by students. In this study we made the choice to group building blocks based on activities. One of the goals is to get an optimum between annotation speed and information we gather mostly succeeded. Some fine tuning of this grouping is still required, because some actions are not tagged (when tagged a code is added to a block) in the annotation process. We think a big improvement would be to tag if students created/typed the code themselves, if they copied the code from an external source or if they copied/moved it from a different part of the code base itself. Also for the 'gathering information' part additional tags should be added. The goal of this is to further distinguish if students are looking at or talking about code, and if they want to copy the code or if they want to learn about the structure of the code. So while we already can see different patterns with the current method, we think that the additional tags above would give us even more insight in the macro strategies used by the students.

For this method to work, a minimum number of annotation blocks are required, based on the patterns as described in section 4.2.1. Each pattern has a different amount of sequenced activities associated to it. Based on the sequences of activities we expect that around 5 to 10 annotated activities are necessary to capture any one of the patterns described in the method of this study. The trial-and-error pattern can be clearly seen, however when a mixture of strategies is used a better distinction between activities is necessary to create a clear picture.

7 Conclusion

In this study it was investigated if a new method could be used to identify the macro strategies students use while working on a programming assignment. To achieve this goal, a case study is performed on two groups of students working on a parallel programming assignment. The first part of this chapter summarizes the conclusions specific to the case study, the second part focusses on the viability of method itself.

In the case study the strategy of two groups were captured. The first group indicates a *trial-and-error* strategy and did not successfully finish the assignment, the behavior of the second group point to a combination of the *reflection-in-action* and *adaptation* strategy and ended up with an almost complete solution. The different results can mainly be seen at the programming concept synchronization, with the focus on the shared value. While the successful group showed the creation process of a correct centralized check-then-act pattern the unsuccessful group did not. The main focus of the unsuccessful group was the taxi class where they applied a trial-and-error strategy to come up with a solution. However the problem was multi-staged which means that a single line edit could not give a correct solution, making finding this solution by trial-and-error highly unlikely.

We now answer the research question regarding the in this study described method: how can we determine what macro strategy students use for different programming knowledge concepts?

The method that is described in this study is suited of finding different strategies student use while working on a programming assignment. The method gives indications of differences between *reflection-in-action*, *adaptation* strategies, and *trial-and-error* strategies. While with the current form of the method trial-and-error strategy is clearly distinguishable, the nuance of indicators between reflection on action and the adaptation strategy, needs to be made more clear. For this, a recommendation is the use of additional indicators, regarding the way programmers use information they find. For example: if they copy code or find explanatory texts about programming concepts. The method becomes increasingly reliable for sequences with a higher number of different activities.

With this work, we expect to contribute to giving better insight for teachers and researchers in the way students work on resolving programming assignments. This creates an opportunity for teachers to change course material to better fit the strategy they want students to employ.

Cited Works

Anderson, J. (2000). Cognitive psychology and its implications. *New York: Worth Publishing*.

Anderson, J., & Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from working memory. *Human-computer interaction 1,2*, 107-131.

Ben-Ari, M. (2001). Constructivism in computer science education. *Computers in Mathematics and Science Teaching*, 45-73.

Biggs, J. B., & Collis, K. F. (2014). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome).* Academic Press.

Boulay, B. d., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 237-249.

Bransford, J. D., Brown, A. L., & Cocking, R. R. (2000). *How people learn.*

Buck, D., & Stucki, D. J. (2000). Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. *ACM SIGCSE Bulletin, 32(1)*, 75-79.

Clancy, M. (2004). Misconceptions and attitudes that interfere with learning to program. *Computer Science Education Research*, 85-100.

Davies, S. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies, 39*, 237-267.

Davis, R., Maher, C., & Noddings, N. (1990). Constructivist views of the teaching and learning of mathematics. *Monograph*, 336-341.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., & Zander, C. (2006). Putting threshold concepts into context in computer science education. *ITICSE*, 103-107.

Edwards, S. (2003). Rethinking computer science education from a test-first perspective. *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 148-155.

Edwards, S. H. (2004). Using software Testing to move students from trial-and-error to reflection-in-action. *ACM SIGCSE Bulletin, 36(1)*, 26-30.

Ginat, D. (2004). Noivce Loop boundaries and Range Conceptions. *Computer Science Education*, 14(3):165-181.

Hou, D., Jablonski, P., & Jacob, F. (2009). CnP: Towards an Environment for the Proactive Management of Copy-and-Paste programming. *Program Comprehension*, 238-242.

Izu, C., Weerasinghe, A., & Pope, C. (2016). A study of code design skills in novice programmers using the SOLO taxonomy. *In Proceedings of the 2016 ACM Conference on International Computing Education Research*, 251-259.

Kolikant, Y. B.-D. (2001). Gardeners and cinema tickets: High school students' preconceptions of concurrency. *Computer Science Education, 11(3)*, 221-245.

Kolikant, Y. B.-D. (2005). Students’ alternative standards for correctness. *ACM*, 37-43.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin, 38(3)*, 118-122.

Lönnberg, J., Berglund, A., & Malmi, L. (2009). How students develop concurrent programs. *Australian Computer Society*, 129-138.

Lye, S. Y., & Koh, J. H. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior*, 51-61.

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2007). Investigating the viability of mental models held by novice programmers. *ACM SIGCSE Bulletin*(39(1)), 499-503.

Maalej, W., Tiarks, R., Roehm, T., & Koschke, R. (2014). On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology*, 23-31.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., . . . Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *ACM SIGCSE Bulletin*(33(4)), 125-180.

Oracle. (2017). *Runnable*. Retrieved from Oracle: https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html

Oracle. (2017). *syncmeth*. Retrieved from Oracle: https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Oracle. (2017). *Thread*. Retrieved from Oracle: http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

Or-Bach, R., & Lavy, I. (2004). Cognitive activities of abstraction in object orientation: an empirical study. *ACM SIGCSE Bulletin*(36(2)), 82-86.

Rist, R. (1995). Program structure and design. *Cognitive Science, 19*, 507-562.

Schwill, A. (1994). Fundamental ideas of computer science. *Bulletin of the European Association for Theoretical Computer Science*, 53:274-274.

Sleeman, D., Putnam, R., Baxter, J., Kuspa, & Pascal, L. (1981). high-school students: A study of misconceptions. *Education computer research*, 221-226.

Smith, J., diSessa, A., & Roschelle, J. (1992). Misconeptions reconceived: A constructivist analysis of knowledge in transition. *Learning Sciences*, 115-163.

Soloway, E., Spohrer, J., & Littman, D. (1988). Generating alternative designs. *Teaching and learning computer programming*, 137-152.

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13(2):8.

Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct?. *ACM* (29.7), 624-632.

Tew, A. E., & Guzdial, M. (2010). Developing a validated assessment of fundamental CS1 concepts. *In Proceedings of the 41st ACM technical symposium on Computer science education*, 97-101.

Whalley, J. L. (2006). An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies. *Proceedings of the 8th Australasian Conference on Computing Education-Volume*(52), 243-252.

Whalley, J., Clear, T., Robbins, P., & Thompson, E. (2011). Salient elements in novice solutions to code writing problems. *Conferences in Research and Practice in Information Technology Series*, 114:37-45.

Xie, S., Kraemer, E., & Stirewalt, R. E. (2007). Design and evaluation of a diagrammatic notation to aid in the understanding of concurrency concepts. *ICSE*, 727-731.

Yadin, A. (2011). Reducing the dropout rate in an introductory programming course. *ACM inroads, 2(4)*, 71-76.