



Towards understanding students' sensemaking of test case design

Niels Doorn^{a,*}, Tanja E.J. Vos^{a,b}, Beatriz Marín^b

^a Open Universiteit, Valkenburgerweg 177, Heerlen, The Netherlands

^b Universitat Politècnica de València, Camino de Vera, s/n. 46022, Valencia, Spain



ARTICLE INFO

Keywords:

Software Testing
Sensemaking
Software Engineering
Computer science educational research
Higher education

ABSTRACT

Context: Software testing is the most used technique for quality assurance in industry. However, in computer science education software testing is still treated as a second-class citizen and students are unable to test their software well enough. One reason for this is that teaching the subject of software testing is difficult as it is a complex intellectual activity for which students need to allocate multiple cognitive resources at the same time. A myriad of primary and secondary studies have tried to solve this problem in education, however still with very limited results.

Objective: Before we can design interventions to improve our pedagogical approaches, we need to gain more in-depth understanding and recognition of sensemaking as it is happening when students design test cases.

Method: An initial exploratory study identified four different sensemaking approaches used by students while creating test models. In this paper we present a follow-up study with 50 students from a large university in Spain. The used methodology was based on the previous study with the improvements that originated from its evaluation. We asked the participants to create a test model based on a description of a test problem using a specialized web-based tool for modeling test cases. We measured how well these models fit the test problem, the sensemaking process that students went through when creating the models, and the students' perception of the modeling task. The participants received no compensation for their efforts, and we scheduled the experiment during a regular class. Apart from the created models and their metadata, we also collected recordings of the students' computer screens made during the experiment and used a questionnaire to study their perspectives on the assignment. All the collected textual, graphical, and video data was analyzed using an iterative inductive analysis process to allow new information about the different sensemaking approaches to emerge.

Results: We gained better insights into the sensemaking processes of students while modeling test cases for a problem. The results enabled us to refine our previous findings, and we identified new sensemaking approaches.

Conclusions: Based on these results, we can further investigate ways to influence the sense-making process in education, the possible misconceptions that have a negative influence on it, and the desired mental model we want our students to have to design test cases.

1. Introduction

As the role of software in our society increases, its quality becomes more important. However, this quality is not always evident. The actual use of software is often different from the expected use, leading to failures that can highly impact the system in which

* Corresponding author.

E-mail address: niels.doorn@ou.nl (N. Doorn).

<https://doi.org/10.1016/j.datak.2023.102199>

Received 11 November 2022; Received in revised form 18 May 2023; Accepted 27 May 2023

Available online 1 June 2023

0169-023X/© 2023 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license

(<http://creativecommons.org/licenses/by/4.0/>).

the software functions, and the value it provides to the stakeholders. Many consider *software testing* to be one of the most effective ways to assure software quality and avoid these failures, however, our computer science curricula do not train computer science students to test well and hence, after graduation, they often arrive at software companies unprepared [1–5]. This was recently stressed by Titus Winters, principal engineer at Google, who presented a keynote [6] on the gap between industry and computer science education, and stated that one of the most important subjects missing in computer science is software testing.¹

In order to design interventions and improve our didactic approaches, we want to understand the complete process students go through when testing. We need to take a holistic view [7] to gain a greater perspective on the different approaches students may take, their potential strengths and pitfalls, and their uses in different parts of a testing course. A better characterization of the sensemaking approaches would give us greater insight into that process.

This work uses an iterative inductive analysis process to identify sensemaking approaches taken by students while creating tests for a given problem using exploration and modeling. In earlier work, we performed a study with the intention to gain initial insights into this sensemaking process with a smaller group of students [8]. In that study, we identified four different sensemaking approaches used by students to model test cases. The results of that study enabled us to perform a more sophisticated experiment with a larger group of students. With this study, we want to validate our previous findings, gain more insights in to these approaches, and also identify other possible approaches.

In this paper we describe an experiment to further explore the sensemaking of students' while they design test cases using a modeling tool. We found new approaches of test case design that we identified by analyzing the produced models, the screen recordings taken while modeling, the filled in questionnaires, and the written summaries of the participants.

The contributions of this paper are:

- The identification of new sensemaking approaches taken by students while modeling test cases.
- The identification of practices taken by students where no sensemaking is involved.
- The refinement of our previous identified sensemaking approaches.
- We made connections between the sensemaking approaches and other subjects in the curriculum.

This paper is structured as follows: we first present an overview of the software testing education landscape in Section 2, followed by a conciliation of prominent schools in software testing education in Section 3. After that, we give an explanation of making sense when testing in Section 4. Next, we present the design of the experiment in Section 5. This is followed by a description of the execution of the experiment in Section 6. The analysis of the results are presented and discussed in Section 7. The answers to the research questions can be found in Section 8. We discuss this research and its threats to validity in Section 9. This is followed by related work in Section 10. Finally, our main conclusions and further work are presented in Section 11.

2. Software testing education

The area of software testing is comprised of a myriad of topics, subareas and technologies [9]. Different test processes are recognized like planning, control, design, execution, evaluation and reporting. These processes have different activities at different levels of the software development process (programming, design, requirements, etc.) and give rise to different types of testing (unit, integration, system, acceptance testing). Moreover, different properties can be tested (functionalities, performance, security, etc.) at different levels, giving rise to many different, informal as well as formal, techniques. All these activities need to be managed, monitored and continuously improved. Furthermore, testing can be done manually or automated, the latter gives rise to a myriad of solutions and technologies to choose from. Finally, test results need to be communicated effectively to ensure that they serve their purpose: enabling developers to improve the quality of the tested software. Fig. 1 shows some of the different questions that software testing education can answer and the many options that educators have to choose from.

Software testing has been identified as the most used quality assurance technique in industry [10,11]. It encompasses so many different topics, that one would expect it to be well integrated and covered in computer science curricula. Unfortunately, this is currently not the case. Two recent systematic literature reviews show that testing is often only a tiny part of the curriculum, which is often taught late in the program and often as an isolated topic [1,2]. A systematically developed body of knowledge with didactic approaches, educational settings, and learning outcomes on software testing is lacking.

Software testing is often taught based on teachers' preferences, ideas, convictions and intuitive ideas on what is important, what works and what does not. However, many choices can be made at the curriculum and course level leading to all kinds of different educational approaches. It is a multidimensional space of overlapping choices that can be combined in different ways.

At the curriculum level, we can teach testing in designated courses solely dedicated to testing, we can give more emphasis to testing in related courses (like programming or software engineering) or we can do both. When testing is explicitly treated in programming courses, we can decide on voluntary versus obligatory testing in assignments [1]. Observations tell us that, when the testing assignment is left voluntary, most of the time students do not develop any test cases for their programs. Even if they do test, they mainly check common program behavior, leaving parts of the code uncovered and failures undetected [12].

At the course level, we can decide to go for *broad* or *narrow* treatment of testing. Broad would mean that we can only briefly touch upon the many topics and sub areas of the field mentioned above such that students have an overview (ISTQB and other standardization curricula are structured this way, e.g. IEEE and ACM [13]). Narrow would mean that we focus on a few specific parts (e.g., a unit testing framework or test case design or performance testing) and go more into depth for the chosen contents.

¹ Although not in the abstract of the keynote, Winters confirmed this statement in personal communications with one of the authors.

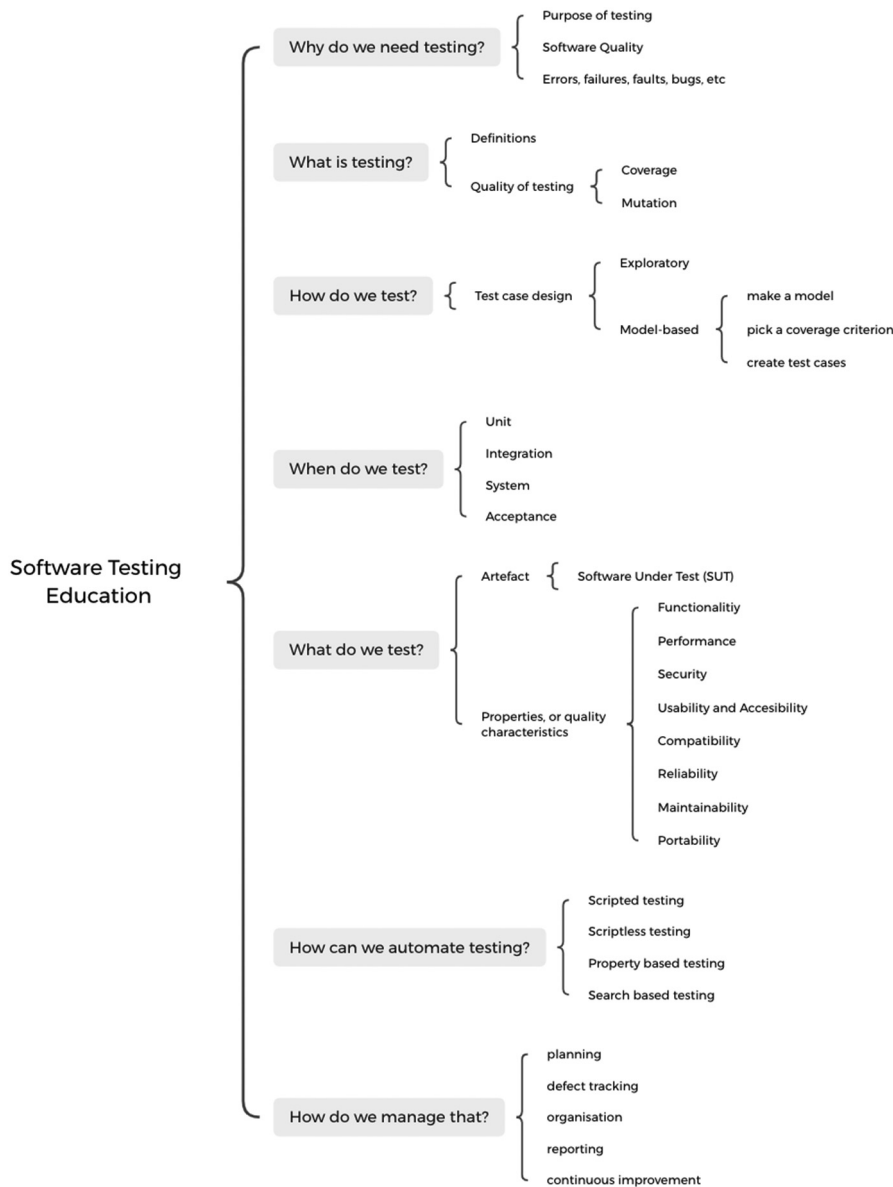


Fig. 1. A mindmap showing the software testing landscape.

We can moreover choose to put emphasis on *general testing skills* (e.g., test case design, reporting or planning) or *specific technologies* (e.g. Junit [14] for unit testing with Java, Sikuli [15] for automating tests at the system or acceptance level, or JMeter [16] for performance testing). Also, we can choose to apply these skills or technologies to well-defined *toy problems* or more *realistic software* Systems Under Test (SUTs) for which no well-defined test solution exists (students do not always like these too much). Other choices are related to whether we teach our students that testing is *informal and exploratory*, or *formal and model-based*. These approaches come from two completely different schools [17] with contradictory views on how software testing should be taught. These two schools have discussed for years and have never come to common ground (we will discuss this further in Section 3).

In this paper, we concentrate on the teaching of *how to test*, as in the topic of *test case design* as presented in Fig. 1. We will consider functional tests at the system specification level. We consider test case design to be one of the most challenging and important parts of testing since the test cases you create determine the quality of your testing. To be able to create high-quality tests, students needs a combination of conceptual knowledge, attitude and scientific thinking to understand the software product and its context [18]. In current curricula test case design is not being taught well enough. Research [12,19] shows that students are often under the impression that they design test cases in a structured way, and that their sets of test cases are complete; however, that this is not the case.

The aim of this paper is to concentrate on the sensemaking process while designing test cases with the intention of improving testing education.

3. Reconciling schools of software testing

There are different schools with contradictory views and terminology on how software testing should be defined, done and taught. Although the community cannot even get agreement upon how the schools are named, we can roughly distinguish the following two.

There is the **analytical school**, where the emphasis is on better testing by improved precision of specifications and many types of models, i.e., *model-based testing*. This school has many proponents in academia (some prominent examples are [20–23]). The idea is that the test cases are derived from a model that describes the functional aspects of the system under test. The model guides the tests that need to be executed. By using models, test cases can in principle be automatically generated, which improves the maintainability of them.

There is the **context-driven school**, where emphasis is on better testing by adapting to the circumstances under which the product is developed and used. Test cases are made up while exploring, learning and questioning, i.e., *exploratory testing*. This school has most of its proponents in consultancy, the most prominent examples being [24–26].

Studies have been published to find out which one is better with no clear conclusions [27–30]. We believe this is because both schools are right. Testing should be done and taught as being both model-based and exploratory.

Test case design requires “figuring out” important characteristics and information about the system under test by exploring, questioning, studying, observing and inferring. Based on this information, we can construct a test model. The next step is selecting an appropriate coverage criterion, and finally, we can create test cases. These steps are also shown in the Model-based testing branch in Section 2. However, we can only make useful test models for test case design once we have made enough sense about the problem at hand. In this paper we want to study the sensemaking approaches that students go through while designing test cases using models. If we are able to get a better understanding of these approaches, the tools in our didactic toolbox can be improved [7].

4. Making sense when testing software

Odden and Russ [31] have proposed the following definition of sensemaking in science education that is coherent with existing definitions and theories.

Definition 1. Sensemaking is a dynamic process of building or revising an explanation in order to “figure something out” – to ascertain the mechanism underlying a phenomenon in order to resolve a gap or inconsistency in one’s understanding.

While sensemaking, students use informal — and formal knowledge to explain or understand possible situations based on the information they can find and observe. This is an iterative process of coming up with ideas, and connecting them with other ideas. These connections can at some point in time stabilize into a, for the student, coherent set of ideas and understandings.

As we indicated before, good test case design needs a combination of conceptual knowledge, attitude, and scientific thinking to understand the software product and its context [18]. Students need to learn how to build test cases using an exploratory model based testing approach, i.e., they need to explore to find what is important to test and based on that they should create a model for testing. The nature of exploratory testing itself can be described as “figuring out” what the problem at hand is about. The problem at hand needs to be studied, the domain of the problem needs to be understood, and all relevant aspects need to be taken into account to create a model.

In the experiment presented in this article, we studied the sensemaking of test case design according to the definition of Odden and Russ. We wanted to see how prior knowledge, ideas, and intuitions from students are being used while they are “figuring out what is important to test given the problem at hand”. Moreover, we wanted to see how, and what model (i.e., new knowledge) students are able to construct by making connections and integrating prior knowledge with new knowledge.

5. Design of the experiment

We performed a quasi-experiment [32] to study students’ sensemaking while solving an assignment that asked them to design test cases by creating a flow-chart like model. We used the well-known guidelines presented in [33] for this experiment.

This experiment builds on our preliminary exploratory experiment [8]. In a sense, we replicated the previous experiment with additions to enhance the reliability of the study, and to gain more insights in the sensemaking of students. We added the use of screen recordings of the modeling sessions to be able to analyze the steps students take while modeling. The second addition is that we asked the participants to write summaries of their approach of the assignment. We used the same research questions and variables, but improved the experiment with these additions.

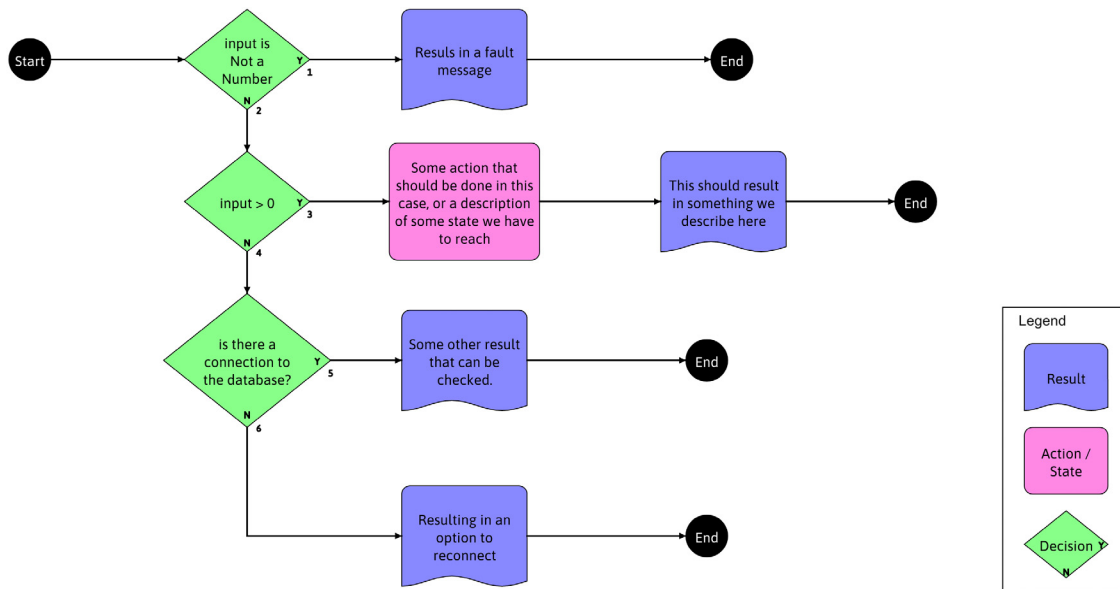


Fig. 2. An example testcompass model (TCMs).

5.1. Context

The subjects are bachelor students enrolled in an obligatory programming course which is part of the first year multimedia bachelor of a large university in Spain. The bachelor counts for 240 European Credit Transfer and Accumulation System (ECTS) credits in total. In this course, testing is only indirectly taught by using Test Informed Learning with Examples (TILE) [34,35]. This is an approach where assignments are seamlessly modified in subtle ways to make students more aware of software testing early on, without completely overhauling an existing programming course. Examples of such modifications are providing test data to the student, asking them questions, and rephrasing assignments in such a way that we ask students to *test* instead of *run* their program. Considering that subjects correspond to first-year students that pass the national exam to entry the university with similar scores, we advocate that they have similar base knowledge. Moreover, taking into account that students are cursing the first course related to computer science, where they learn how to program and how to test by the application of the TILE approach, we can assume that all their knowledge on testing is what they learned in the course.

5.2. TestCompass tool

During the experiment, the students utilized TestCompass, a web-based tool² designed for constructing flowchart-like models. TestCompass Models (TCMs) are easy to construct since they must adhere to only a few guidelines. Each TCM should start with a start-node and end with at least one end-node. There are only three types of nodes besides that: state/action, decision, and result. These nodes have no restrictions regarding the labels they can contain, their semantics is loosely defined by their type and the user-defined label. The connections between the nodes must adhere to specific but simple guidelines that are checked automatically by the tool:

- start, state/action, and result nodes should have one outgoing link
- decision, state/action, result, and end nodes can have multiple incoming links
- decision nodes must have two outgoing links: one for the YES (or true) path and another for the NO (or false) path of the decision
- one or more paths can be followed from the start node to the exit nodes

An example test model can be found in Fig. 2. The example illustrates how a TCM can be used to describe the test-relevant aspects of a SUT, for example, aspects related to the input domain, the morphology of the inputs, or the environment in which it is executed (e.g. the database connection).

TCMs are abstract flow-charts that provide a visual representation making it easier for students to comprehend complex ideas and relationships. Since flow charts are commonly used in everyday life, students are already familiar with their basic structure,

² <https://www.compass-testservices.com/>.

Testing the average age calculator

You work for a travel company. The sales department wants to know what the average age is of the people who booked their holidays with your company. One of the developers in your team has developed a program to calculate the average age for a hundred people at the time. The program can handle up to a hundred dates of births and calculates the average age in years. It gets its data from a remote server as a .txt file, where each line contains the name and the age.

Assignment: Design a test model in TestCompass to adequately test the application.

Fig. 3. The test assignment as it was provided to the students.

allowing them to quickly grasp the information being presented. Consequently, using these diagrams for the test modeling activity helps to minimize distractions for students during the assignments.

Another advantage is that the tool can generate test cases from a TCM for different coverage criteria:

- node coverage — in which all nodes are visited at least once
- link coverage — in which all links are visited at least once,
- condition coverage — in which all paths from condition nodes are visited,
- path coverage — in which all possible paths through the model are visited

This enables us to compare the test suites generated by the students' models.

5.3. Research questions

We wanted to get further insights into the students' sensemaking while analyzing a test problem and creating the corresponding test models. To achieve that, we wanted to measure different aspects of the models and observe the way they were designed by the students. Therefore, we defined the following research questions:

- RQ1: What are the different approaches that students use for creating test models and “figuring out” what to test?**
- RQ2: How is the use of each approach distributed among the student population?**
- RQ3: What is the students' perception of understanding and usefulness of the created test model?**
- RQ4: What connections have students made with their prior gained knowledge?**

5.4. Description of the SUT

The exercise consists of testing a SUT that calculates an average based on data from a text file, accessible on a remote server. Both the file and the remote server are purely fictional to increase the complexity of the problem, and it is not required for the student to actually access the file or the server during the assignment. The complexity and high level description of the assignment assure enough options for students to explore the problem, and to model it in individual different ways. The assignment description as it was provided to the students is in Fig. 3.

5.5. Variables and data analysis process

In order to answer the research questions, we have collected four different data items. We collected the students' models, their written summaries and questionnaires asking about the students' approach of the given task (described in Section 5.7), and we collected screen recordings taken during their work on the assignment. To allow new approaches to emerge from the data set, we used an iterative inductive analysis process to qualitatively analyze the data. We have chosen this approach because it has the ability to adapt to new information emerging from the diverse types of data we collected, such as new approaches, practices, or other findings that are of interest. Using this analysis method we were able to measure the following variables to look for indications and aspects of the sensemaking approaches and other practices used by the students.

v1: Quantitative aspects and properties of the resulting models:

- *Number of nodes* in the model (all types of nodes)
- *Number of decision-type-nodes* that are used in the model
- *Complexity* of the model. This is measured by looking at the number of test cases it generates for different coverage criteria (node, edge, multiple, and path).
- *Time* used to create the models as reported by the students.

- v2:** How the resulting models fit the given test problem — aspects related to the **new knowledge created**.
- v3:** The **observations about the process of modeling the tests** by the students — aspects related to the **figuring out**.
- v4:** The **perception of the students** about the whole test modeling task:
- Perception of difficulty to use the tooling
 - Perception of clearness of the functionality of the tool
 - Perception of understanding of the problem
 - Perception of the usefulness of their model for testing

5.6. Ethical aspects

Participation for the students in this experiment was voluntary. Before the start of the session, we asked the students for their consent in partaking. All the students consented to participate and for their anonymized data to be used by the researchers.

The participation in the experiment did not affect the grading of the course or any other course or subject. Nevertheless, we explicitly asked the students to take the effort to create the test models to the best of their abilities.

All collected data from the participating students was anonymous and not traceable to the individual students. We assigned numbers to the students based on a mapping between the students' administrative id's and a sequence of numbers. We only used these assigned numbers throughout the rest of the process, the mapping was not used in the rest of the study. Moreover, the collected artefacts do not contain any personal data.

5.7. Tasks and materials

For the experiment we planned the following tasks. First, the TestCompass tool was presented to the students by an experienced lecturer familiar with the tooling. Students were also provided with written instructions on activating and using the tool.

Each student was given a description of the assignment as shown in Fig. 3, and the way in which to record the screen and how to hand in the created models after finishing the assignment. Subsequently, we asked them to analyze the case and create a testing model using the TestCompass tool for the described system under test.

When the students finished creating their test models, we asked them to write a summary as if it would be passed on to help someone else, a peer, who needs to do the same assignment. This scenario encouraged participants to focus on the information that they considered to be the most important for another person to read, increasing ecological validity in the tasks [36].

After the summary, we asked them to answer a questionnaire in order to gain a better understanding of their perception of modeling the test cases and for their feedback. We asked the following questions:

- Q1:** How difficult was it for you to use the TestCompass tool? (*Likert scale: 1-very hard, 5-very easy*)
- Q2:** How clear was the functionality of the tool for you? (*Likert scale: 1-very unclear, 5-very clear*)
- Q3:** How difficult was it to understand the specifications? (*Likert scale: 1-very difficult, 5-very easy*)
- Q4:** Did your understanding of the specifications improve by using modelling? (*Likert scale: 1-not at all, 5-very much*)
- Q5:** Do you think your model covers all parts of the specifications? (*multiple choice: Yes, my model is complete, No, my model is not complete, I am not sure if my model is complete, I don't understand this question*)
- Q6:** How much time did you spent on modelling (in minutes)? (*Open end question*)
- Q7:** Feedback or remarks (optional but much appreciated) (*Open end question*)

After the questionnaire, we asked the students to upload their created model and their summary in a Google form. A summary of the experimental tasks is shown in Fig. 4.

6. Execution

In the session, which took place on the 20th of December 2021, a total of 50 students were present. During the experiment, 40 models were created by these students. Not all students recorded their screen, in total, 32 recordings were made. The questionnaire was filled in by 45 students. Not all students who were enrolled in the course were present at the start of the scheduled time slot, to accommodate late arriving students, the start of the experiment was postponed for a few minutes. The scheduled time for this experiment was sufficient to allow for this delay. The instructions and the explanation of the tool and the assignment were done plenary.

To enable the recording of the screens, we used Microsoft Teams. The students are familiar with this software, and have it installed on their laptops by default. Microsoft Teams allows for screen sharing and recordings of meetings. We created a so called 'team' in Teams with 80 numbered channels, making sure we would have a channel for each student. In each channel, a student

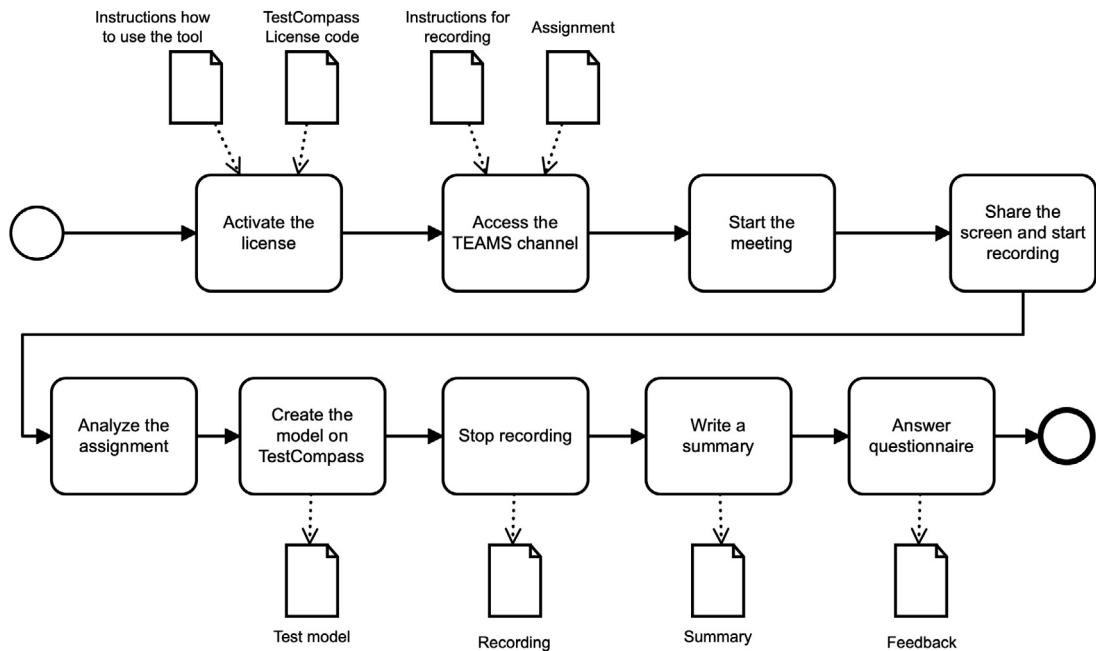


Fig. 4. Summary of experimental tasks.

could start a meeting, then share their screen, and start a recording. When the student finished, the recording was automatically saved in the channel, allowing the researchers to download the recordings centrally from the team. This might be considered as non-intentional use of this software, but it provides exactly what we needed without the need to install new software.

To make sure we could match the recordings of the students' screens with the models and filled in questionnaires, we created a list with a mapping between students and the Teams channel numbers. This list was shared in the Microsoft Teams environment as well. This way, students knew in which assigned Teams channel they needed to work.

The students were each given a licence key for the TestCompass tool. To activate the licence, the students needed to visit the TestCompass website, register and activate the TestCompass tool. Since the tool is web based, no further installation of software on the local machine was required. None of the students had any problems gaining access to registered and working with TestCompass tool.

After the preparations, the students started with the assignment. The assignment was provided digitally in PDF format. To avoid influencing their modeling efforts, no further instructions were provided on how to model the assignment. Although the intention was to let students individually work on the assignment, some students cooperated or discussed the assignment with others. The researchers present during the experiment observed this, but decided not to interfere.

7. Analysis of the results

After the students completed the assignment, we collected all the data items. Firstly we exported the models from TestCompass and converted them to bitmap images for easier analysis, secondly we generated the test cases for all possible scenarios for our quantitative data. After that, we collected all surveys, summaries, and screen recordings.

The bitmaps of the models and the surveys are made available through an online repository which can be accessed through a webpage on <https://edu.nl/jedr9>. The webpage is protected with Sensemaking123# as a password.

Analysis was done using different tools, we used TestCompass to export the created models and for the automatic generation of test cases, Atlas.ti for analysis using open coding and axial coding of all the collected artefacts. Both quantitative and some qualitative results were kept in a spreadsheet and in a Jupyter Lab notebook. We used Jupyter Lab to analyze the data and create graphical representations of the data.

7.1. Quantitative aspects of the models (v1)

We counted the nodes in which a distinction was made on the semantic meaning of the node types. The start and end nodes were disregarded in the count since they are both compulsory to use, a model is invalid without at least one start and one end node, and the use of more than one end node does not give any insight into the complexity of the model. If more than one end node is used, this is done for clarity of the model. The decision nodes of each model were counted separately since they give a good indication of

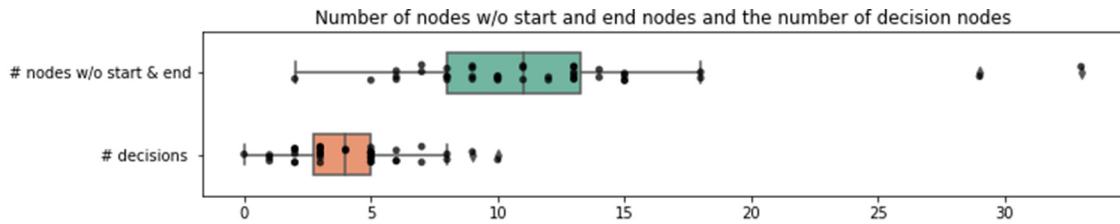


Fig. 5. Number of nodes w/o start and end nodes and the number of decision nodes ($n = 40$).

Table 1

Number of nodes w/o start and end nodes and the number of decision nodes table ($n = 40$).

	# nodes w/o start & end	# decisions
mean	11.7	4.175
std	5.716	2.308
min	2	0
max	33	10

Table 2

Number of test cases generated by TestCompass for different coverage criteria ($n = 40$).

	Nodes	Edge	Multiple condition	Path
mean	3.750	4.525	4.750	25.175
std	2.192	2.764	2.415	66.704
min	0	0	0	0
max	10	13	11	256

Table 3

Test case coverages filtered for infinite loops ($n = 37$).

	Nodes	Edge	Multiple condition	Path
mean	3.81	4.648	4.756	6.459
std	2.234	2.81	2.498	4.5
min	0	0	0	0
max	10	13	11	24

the complexity of the model. The complexity of models increases because each decision node creates a fork in the execution paths, resulting in an extra test case for path coverage. Table 1 summarizes these counts.

Looking at the distribution in Fig. 5, we see that most of the models contain no more than fifteen nodes, of which a third are decision nodes.

TestCompass allows for automatic test case generation using different coverage criteria. The amount of test cases generated for these tests give an indication of the complexity of the created model. If a model contains an infinite loop, TestCompass will generate the first 256 test cases for path coverage, and it will issue a warning that there are more test cases possible. The tool is unable to detect infinite loops nor provides options to handle them differently. This is reflected in the data as shown in Table 2. Three of the created models contain such infinite loops. This high number of path coverage test cases causes a distorted view. To get a more accurate insight in the number of test cases for path coverage, we filtered these infinite loop models of the results, as can be seen in Table 3.

We still see a high deviation value for path coverage, which is also reflected in the distribution as can be seen in Fig. 6. This indicates that there is a large variety in the complexity of the models created by the students.

Related to the time measurements, we asked the students to give an approximate indication of the amount of time they spent on the exercise using an open-ended question. The answers range from 5 min to 66 min, with a majority of students who indicate to have spent 40 min or more. Since we asked for an approximate indication, we are unable to calculate any reliable statistical information about the time used.

7.2. Analyzing the newly created knowledge and the process and identification the sensemaking approaches (v2 and v3)

The goal is to identify the newly created knowledge processes from analysis of the collected artefacts and to examine these processes and their use to develop theories of how students approach the creation of test models.

The authors independently assessed all the models to get a good overview of the produced models regarding the defined variables. After that initial survey, two of the authors discussed the models together to identify initial indications of new knowledge creation and sensemaking approaches. Simultaneously, we informally graded our assessments of the fit of the models for the given problem on a one to ten scale, where one would be an unusable model or a model lacking most of the requirements, and a ten being a perfect

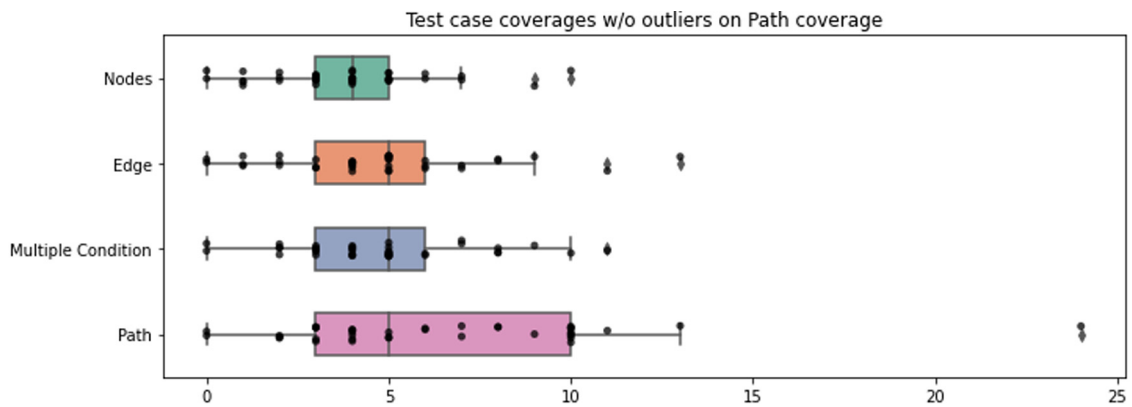


Fig. 6. Test case coverages w/o outliers on path coverage ($n = 37$).

score. After that, the two authors shared their insights with the third author to get a consensus on the initial indications and fit of the models.

Following this initial evaluation of the models, we looked for other information by open coding all artefacts, including the models, screen recordings, questionnaires and summaries. The screen recordings were studied to find the usage of the tooling, and since we recorded the whole screen, the usage of other tools such as notebooks or, in one case, the use of a mindmap tool to support the process of creating a test model. To allow new approaches to emerge from the data set, we used semi-structured iterative inductive analysis, i.e. an incremental process of coding, refining and updating. We started with a small set of codes that emerged from the initial assessment of the models and our previous work. Subsequently, we gradually added new ones or refined existing ones using inductive reasoning while studying the collected data. This process was semi-structured by incorporating observable keywords and key concepts associated with two pertinent variables, namely *v2* (aspects related to the new knowledge created) and *v3* (aspects related to the figuring out). This gradually led us to build Table 4. This way of working allowed for gradual improvement in accuracy and understanding over time.

For example, artefacts coded with the code *variable declarations* show that the student tried to use a typical programming construct while figuring out how to model tests. We relate this code to *v2* (aspects related to the figuring out) — the student's looks to be under the impression that modeling test cases needs to be approached in the same way a programming task needs to be approached. We recognized this behavior from our previous experiment and classify this as the developer approach.

Using this method, we classified all the open codes with practices based on the new information that emerged while analyzing the artefacts.

We were able to identify five different practices taken by students: developer, convergent tester, divergent tester, lazy student, clueless student. The last two are not considered to be sensemaking approaches, being practices where no real figuring out is involved.

Students use a variety of approaches to construct their models, which are interconnected. In this section, we discuss each approach and provide examples of the corresponding models. It is common for students to use more than one approach, and we will highlight how different approaches work together to create a comprehensive model.

The *lazy student practice* characteristics are the lack of effort. The student understands the tooling, but does not create a model with the level of detail for it to be usable. We identified two models with this practice. Fig. 7 gives an example of a model of this practice.

The second practice we identified is the *clueless student*. The main characteristic of this practice is that the student clearly struggles to understand the assignment. This was apparent in one of the questionnaire summaries where participant 11 mentioned the following:

Come to class on time because if you don't, you won't find out anything.

We identified three models with this practice, for example the model as shown in Fig. 8.

The most common practice we saw is the so called *developer approach*. This approach shows typical programming constructs in the model. For example: a student that explicitly models a loop to iterate over the dates in the JSON file. Another common indicator from the results are the attempts to declare variables in the model and the modeling of algorithms. Sometimes students write their programming intentions in remarks in the model, presumably because the modeling tool does not provide adequate support for the way these students want to model since the tool is designed to model test cases, not to program. Fig. 9 gives an example of a model of this approach. This approach is also identifiable in the summaries. For example, participant 56 clearly thought in 'code' and mentioned:

I just thought about what errors would come up in python and then I "marked" them

Table 4

The used codes to analyze the models, summaries, and screen recordings of the students and the aspects by which we grouped them, and the amount of times the open code was used in the artefacts. The practices are abbreviated between brackets in the following way: d = developer, ct = convergent tester, dt = divergent tester, l = lazy student, c = clueless student.

New knowledge created aspects (v2)			
Open code	#	Open code	#
number of dates [d]	25	leap year [dt]	7
date format check [ct]	23	algorithm in model [d]	7
valid day [ct]	18	network check [ct]	7
valid month [ct]	16	thinking about code [d]	6
loop [d]	15	tool understanding [c]	6
valid year [ct]	11	boundary value testing [ct]	3
file check [ct]	8	comment in the model	3
not born yet check [ct]	8	boundary value testing [ct]	3
thinking about code [d]	6	too old check [dt]	3
algorithm in comment [d]	3	variable declarations [d]	3
comment in the model	3	defect in model	1
assignment understanding	3	happy path server [ct]	1
code based testing [d]	2	too young check [ct]	1
happy path data [ct]	1	unhappy path data [ct]	1
mistake in model [c]	1		
unclear [c]	1		
unhappy path server [ct]	1		
The figuring out aspects (v3)		Experiment related	
Open code	#	Open code	#
misunderstood specification [c]	6	laptop problems	2
mental model related	4	no model submitted	1
missed the introduction [c]	3	problems with Teams	1
testing skills improvement	3		
use of a mindmap	2		
high level approach [l]	1		
negative attitude	1		
difficulty to design tests	1		

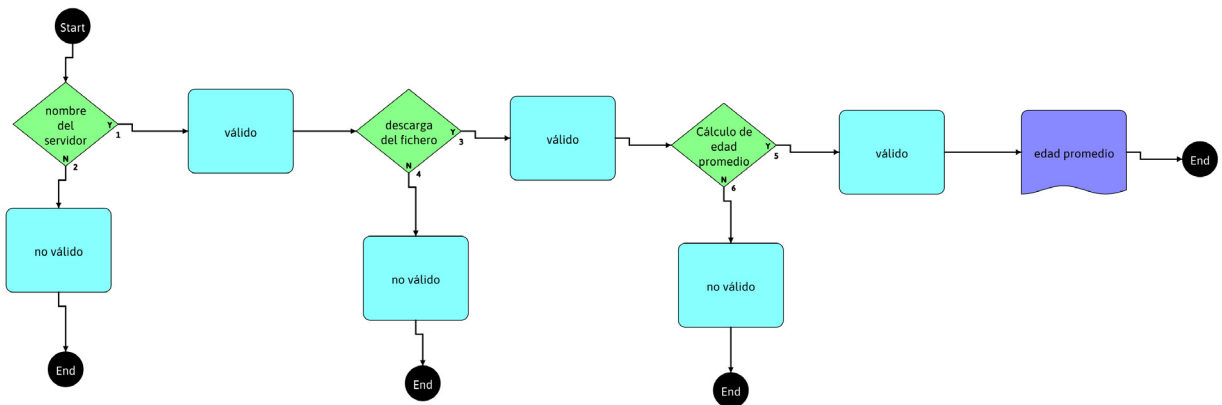


Fig. 7. An example of a model using a ‘lazy student’ practice of participant 27. The labels in the nodes are in Spanish. Following the numbered paths they translate in English to ‘name of the server’ for the first node, then paths 1,3,5: ‘valid’, 2,4,6: ‘invalid’, and the final node reads ‘average age’.

Another example of participant 33 shows the same approach:

I have followed the logic that I usually use when I make a program. As for my strategies, I have tried to find the conditions in the text of the exercise, and from there, I have transferred them to the diagram.

Some students describe the algorithms to calculate the average age in the summaries, participant 10 wrote:

We must see that the year they give us is positive, that the month is within the range between 1 and 12 and, the days [are] between 1 and 31 (except for February that will have to be put up to 29). Then we must also see that the number of years that they give us is less or equal to 100, since it is the maximum that the program allows. Once all these conditions have passed, we will have to obtain the age through the year of birth. Once we have all the ages that we have entered into the program we add them, and divide them by the number of entries we have.

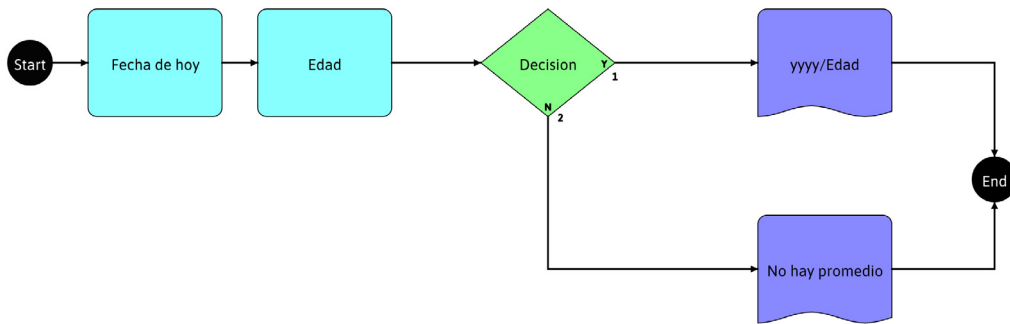


Fig. 8. An example of a model using a ‘clueless student’ practice of participant 4. The labels in the nodes are in Spanish. From left to right, and following the numbered paths, they translate in English to ‘retrieve the file’ for the first node, ‘average’ for the second node, after ‘decision’, path one leads to a division of the year with the average, and path two leads to ‘no average age’.

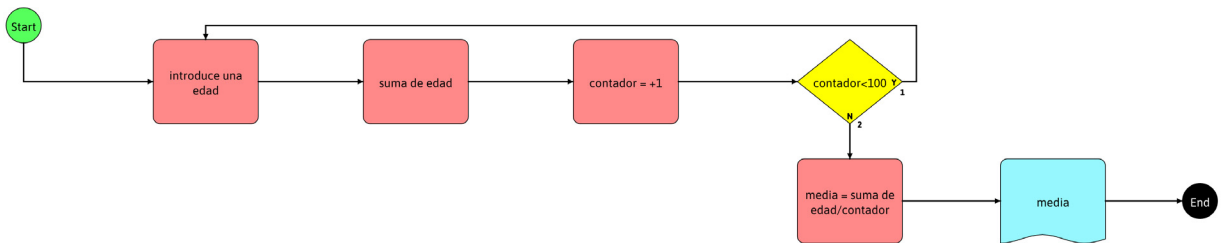


Fig. 9. An example of a model using a ‘developer’ approach of participant 9. The labels in the nodes are in Spanish. From left to right, and following the numbered paths they translate in English to ‘enter an age’ for the first node, ‘sum of the age’ for the second node. This is followed by ‘increase the counter by one’, ‘counter < 100’, ‘average = sum of the age/counter’ and ‘average’. The decision node checks if the value of the counter is below 100, if so, it continues with the first node, effectively creating a loop. If the condition is not met, the other path is followed, in which the average of the ages is calculated and presented.

And participant 20 followed a similar approach:

Once we know that the dates are correct, the age will be calculated by subtracting the current year (2021) minus the year of birth. Finally, when we have calculated the ages, we will add them and divide them by the total number of people to calculate the mean.

The second most common practice we saw is the *convergent tester approach*. Its characteristics are the modeling of scenarios for testing that are inferred solely from the text of the assignment but not beyond. Fig. 10 gives an example of a model of this approach. We identified the convergent tester approach in both the models and the questionnaires. For example, participant 18 mentioned the following in the summary:

(...) you have to think about precisely the cases that can cause problems such as poor data entry, exceptions (such as 0 or 1 data/s) or exceeding the number of elements (more than 100) where a message would be issued announcing these errors.

We identified three different types of scenarios used in the convergent tester approach: the *connection with the server* scenario, where the connection with the server is checked; several *happy path data* scenarios, in which the data of the file is checked for validity, correct formatting and so on, and *unhappy path data*, where the opposite is checked, for example for invalid dates. These last two types of scenarios do not always coincide in the models, often, only one of the two is checked, as can be seen in Fig. 12.

Finally, we identified the *divergent tester approach*. This approach is characterized by the modeling of scenarios that are not directly inferred by the text of the assignment. The scenarios were designed based on exploration of the problem. We saw three indicators for the divergent tester approach in the created models: the handling of leap years, testing unrealistically high ages of persons, and scenarios to test if the server was switched on. Fig. 11 gives an example of a model of this approach. In this model we see that the student checks for leap years, an indication of exploring the problem.

Participant 84 mentions the following in the summary:

(...) be careful because they may give negative or too large numbers

Participant 61 clearly thinks about possible problems with the text file:

(...) the program has to identify the .txt file. Here we have two options, that it recognizes the file and therefore follows the program or that it does not recognize it by terminating the program.

This also shows that students can take more than one approach, in this case the participant also took a developer approach.

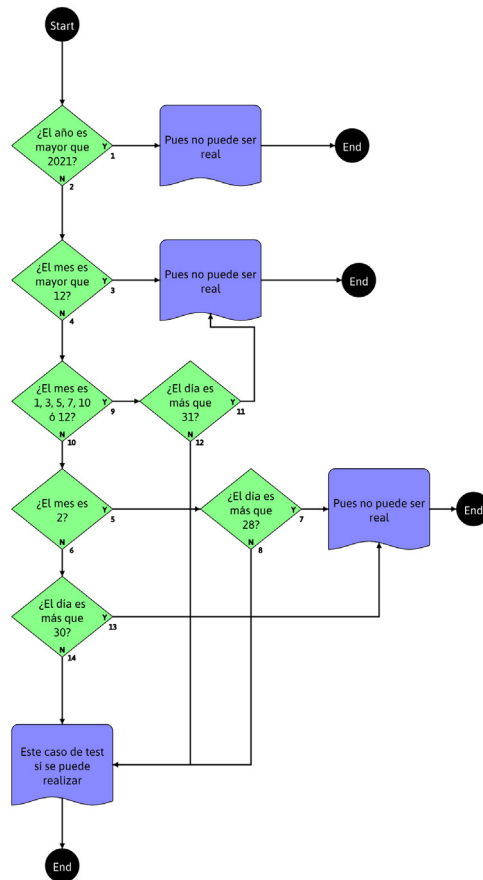


Fig. 10. An example of a model using a ‘convergent testers’ approach of participant 20. The labels in the nodes are in Spanish. From top to bottom, and following the numbered paths they translate in English to ‘the year is greater than 2021?’, following path 1: ‘This can’t be valid’, following path 2: ‘is the month greater than 12?’, via path 3: ‘This can’t be valid’. The other decision nodes check for valid day and month combinations. If we follow path 4, the node reads: ‘is the month 1,3,5,7,10 or 12?’. Following path 9 we reach a node that reads ‘is the day greater than 31?’, following path 7: ‘This can’t be valid’ and following path 10: ‘is the month 2?’. Through path 5 and 7 we reach nodes reading ‘is the day greater than 28?’ and ‘This can’t be valid’. Through path 6 we reach a decision node reading ‘is the day greater than 31?’. The final node reads ‘this test case checks if the value is valid’.

Another example from the summaries comes from participant 62, who gives a clear insight in the sensemaking approach taken:

To have test cases, we must think about the most common errors that we usually make ourselves, such as confusing the name of the files, the file is not being in the same folder as the program, dates entered in the file not being in the format that they ask us. The latter implies that the position of the month and the day are in the wrong order, etc.

7.3. The students’ perception of understanding and usefulness of the created test model (v4)

We measured the following aspects of the students the perception:

- Perception of difficulty to use the tooling.
- Perception of clearness of the functionality of the tool.
- Perception of understanding of the problem.
- Perception of the usefulness of their model for testing.

We used four questions with Likert scale answer options. Fig. 13 shows the results of these measurements for each aspect.

Many of the students indicated that they perceive the tool to be difficult to use, 53% indicated that they find it hard or very hard to use. Only 22% of the students find the tool easy or very easy to use. Only 13% of the students indicate that the functionality of the tool was clear to them, whereas 65% indicate to find it unclear or very unclear. The tool selected to create the test models was new for the students, which can influence the perceptions of the difficulty of using the tool and the clarity of the functionality of the tool. Nevertheless, after explaining the tool there were very few questions about it. Additionally, while observing the students

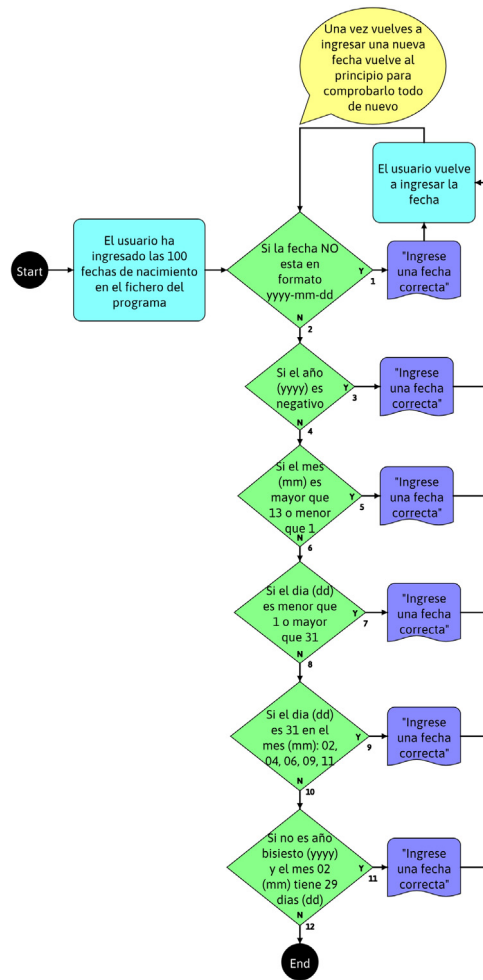


Fig. 11. An example of a model using a ‘divergent testers’ approach of participant 40. The labels in the nodes are in Spanish. The remark node explains that this flow is repeated for all dates, basically creating a loop. Following the numbered paths the nodes translate in English to ‘The user has entered the 100 dates of births in the file’ for the first node after the start, the top decision node checks the correct format of a date: ‘is not in the format yyyy-mm-dd?’, the second checks for negative years: ‘is the year (yyyy) negative?’, the third for valid months: ‘is the month (mm) greater than 13 or less than 1?’, the fourth for valid days: ‘is the day (dd) less than 1 or more than 31?’, the fifth for valid day and month combinations: ‘is the day (dd) 31 and the month (mm): 02, 04, 06, 09, 11?’, and the final decision nodes checks for leap years and the 26th of February: ‘Is it a leap year (yyyy) and the month 02 (mm) and the 29th day (dd)?’. All the ‘N’ branches of the decision nodes are followed by nodes reading ‘Input a valid value’ followed by a node reading ‘the user re-enters the date’.

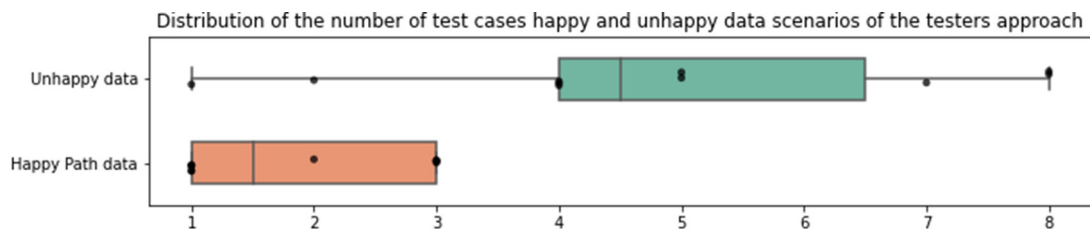


Fig. 12. Distribution of the number of test cases for each aspect of the tester approach.

we did not see them struggle during the use of the tool (no questions, no problems detected in the screen recordings). Moreover, after the assignment, all the students were able to produce models.

A majority of the students indicated to perceive the assignment to be difficult (62%). The assignment was designed to be simple enough to understand the SUT, yet complex enough to allow students to explore and think outside of the box (i.e. test problems

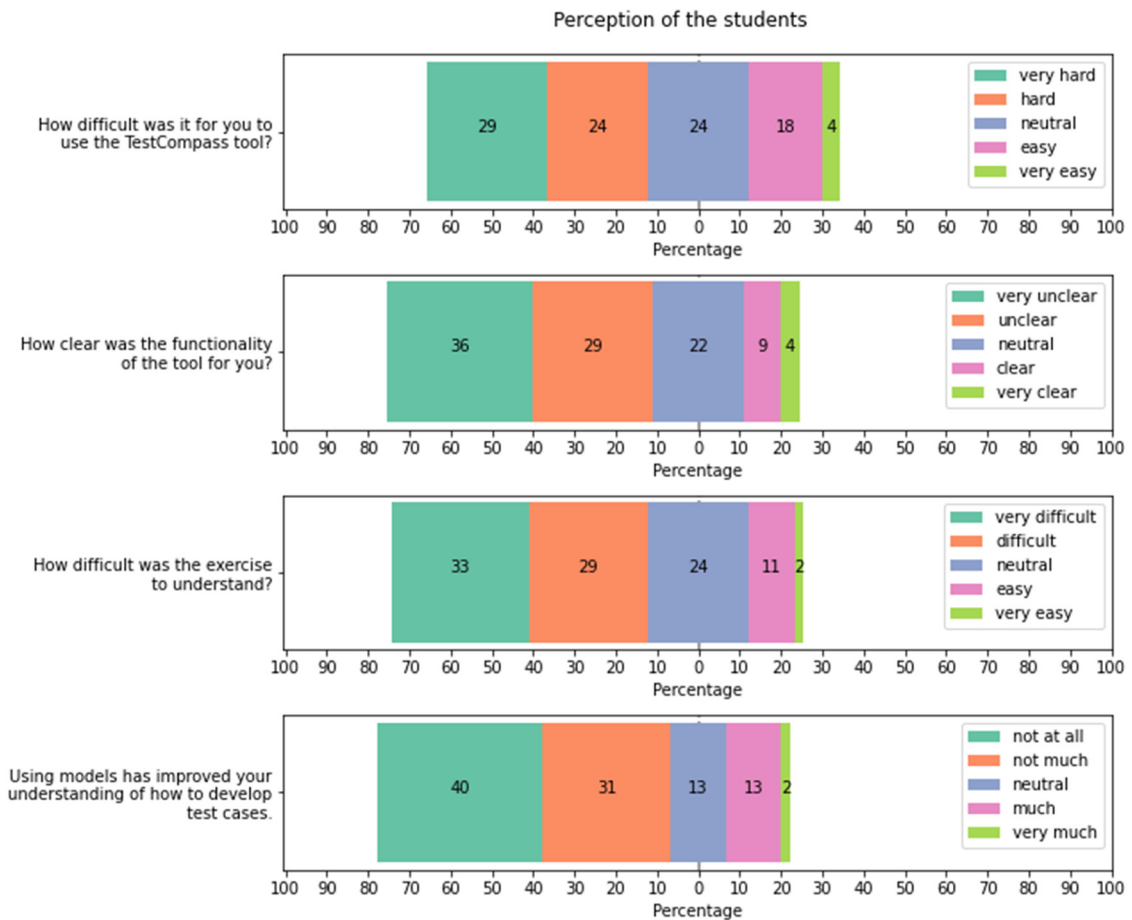


Fig. 13. Perception of the students (n = 45).

related to the connection to the server, the file processing and the formatting of the date). Even though students perceive it to be difficult to understand, we see in the resulting test models that they indeed understood the problem. Moreover, the level of the assignment was similar to others from the course.

The use of models to increase the understanding of how to develop test cases is perceived by 15% of the students as much or very much, and by 71% as not much or not at all. Since we did not measure the testing knowledge of students, we consider that the lack of testing knowledge could influence the perceptions of students. Therefore, we plan to characterize this knowledge by adding some questions related to testing in our coming-up experiments.

We also measured the perceived completeness of the models. Thirty-three of the students indicated that they are not sure that their model covers all the specifications. Five of them indicated that they believe their model is complete, and eight indicate that they are sure that their model is incomplete.

8. Answering the RQs

For RQ1, “What are the different approaches that students use for creating test models and ‘figuring out’ what to test?”, we distinguished 5 different practices, three of which we consider to be sensemaking approaches. We clearly see students who use the developer approach and apply programming constructs, originating from their programming knowledge, in the models. No new knowledge is constructed nor connections with other concepts are made. The modeling tool is used to solely model a programming solution, and not to create a model to test the system. The developer approach is therefore characterized by students building a model based on their existing knowledge without connecting their programming knowledge with other prior knowledge.

The convergent testers approach consists of students determining test scenarios and model those. To determine test cases, they use their prior knowledge and their programming knowledge, to figure out what test cases are meaningful.

It could be that when using the divergent testers approach, the students are creating new knowledge by building connections between their prior programming knowledge and other knowledge, for example, knowledge about the concepts used in the assignment. Using modeling may help in facilitating the building of new connections between the concepts in the description of the assignment and the model.

Table 5
The identified practices ($n = 40$). Students can use more than one practice, but the developer and convergent tester approach are mutually exclusive.

Identified practice	Number of students
Lazy student practice	2
Clueless student practice	3
Convergent tester approach	10
Developer approach	26
Divergent tester approach	5

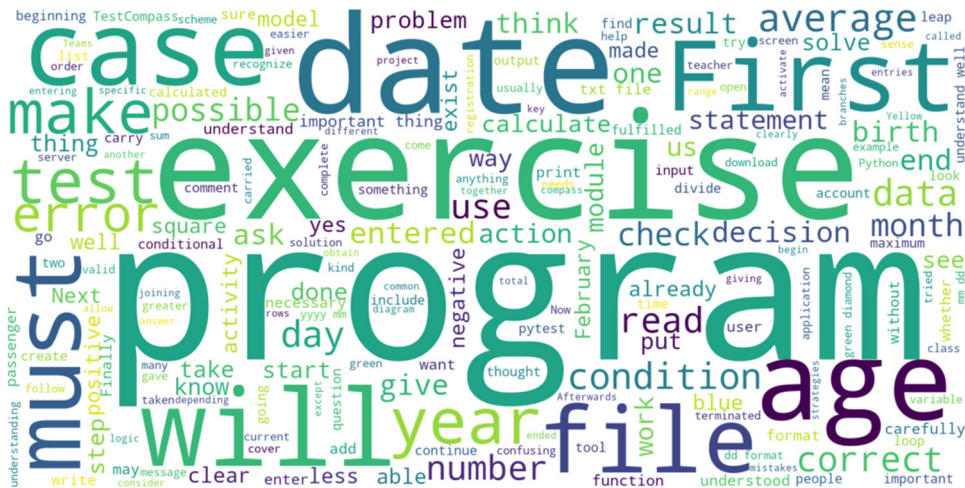


Fig. 14. A word cloud of the summaries showing which words were used most.

To answer RQ2, “How is the use of each approach distributed among the student population?”, we present Table 5 which gives an overview of the different practices and their distribution over the student population. As we can see in this table, the developer approach is the most common used, followed by the convergent testers approach. The divergent testers approach is used least often of the sensemaking approaches.

Since students were participating in a programming course taught with TILE, an innovative educational method to teach testing by using examples, we expect students mainly use the testing knowledge with the programming knowledge to create meaningful test models, i.e. the convergent approach. Taking into account the results, we plan to perform future experiments to investigate the effectiveness of the testing teaching approach in order to understand why the majority of students follow a developers approach despite having testing knowledge.

To answer RQ3, “What is the students’ perception of understanding and usefulness of the created test model?”, we analyzed the answers given by the students on the questionnaire, their produced models, the screen recordings and the observations by the researchers during the experiment. We see that the participants indicate that they find the use of the tooling difficult, and that the functionality of the tool was unclear to them. There is a discrepancy between these answers and our observations during the experiment in the classroom, the recordings of the screens and the produced models. If we look at the produced models, we see that most models are technically correctly constructed. The same applies to the understanding of the exercise. During the exercise there were not many questions regarding the exercise. In the answers we see that a small part of the student see a benefit in the construction of models to develop test cases.

In order to answer RQ4, “What connections have students made with their prior gained knowledge?”, we looked at all the collected data for indications of the activation of prior knowledge. Based on the summaries, we see a strong connection with existing programming knowledge. If we look at the word cloud in Fig. 14, we can see that the word ‘Program’ is used many times. The most common practice we identified is the developer approach, this also suggests that there is a strong connection with programming subjects. Other significant indications of the activation of prior knowledge were not found.

8.1. Comparing the results with our previous study

We did our previous study with nine master students, but it demonstrated the existing of different sensemaking approaches followed by students. In this study, we adapted the design of our experiment to be able to get a better insight into the way students create models. We conducted this experiment with 50 bachelor students. In the previous study, we identified four different approaches: *happy path approach*, a *developer approach*, an *exploratory approach*, and a *student approach*. In this study, we gained more insight in the details of the happy path approach and were able to refine its naming in convergent testers approach with

the identification of three different testing scenarios. Moreover, we refine the previous recognized *exploratory approach* naming in *divergent testing approach* which better represents thinking out of the box.

We also refined the previously identified *student approach* as two distinct other practices, the lazy student practice, and the clueless student practice.

If we compare the models produced in this study compared to our previous study, we see that in this study the models contain on average less nodes, 11.7 in this study vs. 14.6 in the previous. The number of generated test cases is similar between the studies for node and edge coverage, and much lower for multiple condition testing, 4.7 in this study compared to 7.2 in the previous. The number of test cases for path coverage leaving out the models with loops is also much lower, 6.45 compared to 16.2. Taking into account that the assignment was different for the previous study compared to the assignment for this study, it is expected to see differences in these numbers.

Looking at the perception of the students, we see a large difference between our previous study and this study. In the previous study, the participants were much more positive about the perception of the use and functionality of the tool and the assignment.

We do not see any substantial difference in the time spent compared to the previous study.

9. Discussion and threats to validity

We have designed and executed the experiment carefully following well-known guidelines for software engineering experimentation [33]. Nevertheless, some threats could affect our results. In this section we discuss some threats and the mitigation strategies followed.

We observed that some students worked together during the experiment. It was not possible to make a registration of which participant worked with whom. Based on the number of models, we can estimate that a maximum of five students teamed up with other student. This has an impact on the results since we are not able to clearly distinguish the individual attributions to the produced models. This not necessarily means that we would have identified different approaches had this not been the case, it might be that some models are more usable because of these collaborative works.

A few students lacked intrinsic motivation to put in their best effort. We did not provide a concrete incentive to participate in the study, other than requesting the students to put in their best effort. Since the number of students who lacked motivation is small, we do not believe this had a great negative influence on the outcomes.

The students who participated follow a bachelor in multimedia, in which programming is a mandatory subject. This is a somewhat different group than the master student's telecommunication who participated in the previous study. It might be interesting to compare the results of this study with students of a software engineering oriented program or specific software testing courses to get a better understanding of the representativeness of subjects, so we could generalize the results.

The selection process of the used system under test involved was thorough and involved researchers with backgrounds in software testing, software engineering, and education. Based on the results of the previous study, it was decided to use an assignment that would be complex enough to be able to model in many ways, including exploring the problem, yet not to complicated that it would require advanced knowledge. For example, the use of a remote server responsible for providing the file with the dates of births makes it possible to explore all kinds of problems concerning the availability of the server, problems with the transfer of data et cetera. The content of the file also makes it possible to address different aspects such as formatting, leap years and invalid dates. Analyzing the data, we did not see models that completely explored the whole problem, but we see exploration in some cases. This indicates that the complexity of the problem is at the right level. The perception of the students does indicate that the problem was hard to understand and that the students are not convinced about their understanding of the problem. This contradicts the quality of the models as we assessed it.

The use of tools may threaten the effectiveness and efficiency of the design of test cases. Our focus is not related to the effectiveness and efficiency, instead we are focused on the exploration of students' sensemaking approaches. Nevertheless, we selected a simple flow-chart web based tool to create the test model to mitigate this threat, and also any threats related to the students not understanding the formalism of the model, to avoid misunderstandings which could generate noise when studying their sensemaking approaches of test case design. We also used Microsoft Teams to record the experimental session, which students are familiar with since it is often used by students in their regular classes.

10. Related work

In a recent systematic literature review on testing education, Garousi et al. [2] provided a comprehensive classification of related work, highlighting the most significant contributions. These include pedagogical approaches for teaching testing, proposals for specific teaching tools, courseware, gamification, as well as observations and trends. Garousi et al. identify little work related to research providing insights into students' learning and testing behavior, particularly during the design of test cases. Up until 2020, Garousi identified only four relevant papers, three of which were published over 20 years ago [37–39]. The fourth paper is from Aniche et al. (2019) [40] who analyzed the feedback reports from 230 students. The paper concentrates on the topics and the activities from their course and identified challenges related to software testing education, such as students' difficulty in choosing appropriate testing techniques and creating maintainable test code. Their work proposes adding a pragmatic perspective to software testing courses, and they report on successful teaching strategies, common mistakes, and preferred learning activities for students.

To the best of our knowledge, there have been only two more recent contributions since 2020 that are relevant to our work's objectives and could fit within the same context. In the following paragraphs, we will provide a detailed description of these contributions.

Enoiu, et al. [41] present a software testing cycle that is centered around their observation that test design and execution can be viewed as a classical problem solving process like the Plan Do Check Act cycle. Their study recognizes the need to explore how testers create test cases in future works. Our study takes a step into this direction by looking at the sensemaking of students.

Niche et al. [42] explore how software developers engineer test cases in practice, through an observational study. Their findings suggest that there is still much to be learned about how test cases are designed in software engineering practice. Given the importance of this objective, in this paper, we specifically aimed at exploring the intricacies of the sensemaking process when designing test cases, with the ultimate aim of enhancing computer science education.

11. Conclusions and further work

The findings in this study validate the existence of different sensemaking approaches. The most common sensemaking approaches for test-case design is the developer approach where the students use constructs similar to those of programming to model the problem. They are applying knowledge which they have previously gained during programming related courses.

Only in the least used divergent tester approach we see students who create new knowledge about the domain of the problem under test, and also on the design of test cases.

Looking at these identified approaches, we hypothesize that the problem with test education worldwide is that the wrong design paradigm [43] is being used. Testing is mostly being taught from a *rational design paradigm* (i.e. emphasizing algorithmic problem-solving, planning, and methods) – similar to how we teach programming. This leads to insufficient and deficient testing since students model the test problem in a similar way they model the programming problem (i.e. developers approach). While doing this, they will fail to explore and experiment with the software to create the new knowledge needed to give an informed conclusion about the quality of the software. We need to teach testing using the *empirical design paradigm* (i.e. emphasizing reflection-in-action). Reflection-in-action, as described by Schön [44], is a characterization of how people complete tasks in the face of uncertainty and novelty, that consists of carrying out an experiment which serves to understand what is happening and (re-)define a strategy (i.e. divergent approach). When testing, students do not have to solve a problem, instead they should start an experiment in which they state the hypothesis and define research questions related to the quality of the SUT. Once the questions have been formulated, students can conduct their investigations in the form of an experiment, in which they look for answers and generate new knowledge.

Using the wrong paradigm to teach testing leads to the identified problems with motivation and engagement detected in [1]. In software testing, there is no *the problem*, nor *the solution*. Consequently, teaching it with the rational paradigm, such that students do have the feeling they have to come up with *the solution* leaves them frustrated with a sense of inconsistency. And, when students are left with these kinds of inconsistencies, they may feel that what they learned just does not make sense [45]. This ultimately, without doubt, contributes to a lack of motivation for testing and all consequences described above.

Future work should concentrate on radically changing the paradigm that is currently being used at universities to teach software testing. We need to go from rational to empirical.

As the next immediate step, it is crucial to gain a more comprehensive understanding of the empirical design paradigm for test education. This can be achieved by conducting in-depth research and analysis to identify the key components and principles of this approach, as well as exploring how it can be effectively applied in the field of education. Additionally, it will be important to examine the potential challenges and limitations of this paradigm and to develop strategies to overcome them. By gaining a better picture of the empirical design paradigm for test education, we can take meaningful steps towards improving the quality and effectiveness of testing in education, and ultimately enhancing the learning outcomes of students. In order to do this we will start to analyze test professionals to find out the thought processes they go through when testing and gain a deep understanding of their activities. For this, it is necessary to take a holistic view to gain a greater perspective and characterization of the sensemaking approaches used by professionals when they test. This can be done by performing task analysis with semi-structured clinical interviews to study the sensemaking of these professionals while performing testing tasks [45].

Having a clear understanding of how students should approach test case design is critical for identifying an effective cognitive model that can guide instructional designs to improve teaching and learning strategies. In essence, this cognitive model should serve as the Rosetta stone that unlocks effective test case design. By leveraging this model, we aim to create instructional designs that can be applied in a variety of educational contexts.

Finally, we plan to use epistemic games as a framework to describe different sensemaking approaches in terms of patterns of activities that students and professionals successfully use on various testing tasks. Epistemic games [46] are well suited for this since they characterize how members of a community of practice work to construct knowledge. These games will be able to direct the focus from just “learning about” (software testing) to “learning to be” (a software tester) [46].

CRedit authorship contribution statement

Niels Doorn: Conceptualization, Methodology, Validation, Writing – original draft, Writing – review & editing. **Tanja E.J. Vos:** Conceptualization, Methodology, Investigation, Writing — original draft, Writing – review & editing, Supervision. **Beatriz Marín:** Conceptualization, Methodology, Validation, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We like to thank Silvio Cacace for his input on the assignment, and for the use of the TestCompass tool, Erik Barendsen for his advice and feedback on this research and of course the participating students from the bachelor GTDM at the Universitat Politècnica de València. The work leading to this paper has received funding from the European Union by the ENACTEST project (101055874) and the Erasmus+ project QPeD under contract number 2020-1-NL01-KA203-064626.

References

- [1] L. Scatolon, J. Carver, R. Garcia, E. Barbosa, Software testing in introductory programming courses, in: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, in: SIGCSE '19, ACM, New York, NY, USA, 2019, pp. 421–427, <http://dx.doi.org/10.1145/3287324.3287384>.
- [2] V. Garousi, A. Rainer, P. Lauvås Jr., A. Arcuri, Software-testing education: A systematic literature mapping, *J. Syst. Softw.* 165 (2020) 110570.
- [3] T. Astigarraga, E.M. Dow, C. Lara, R. Prewitt, M.R. Ward, The emerging role of software testing in curricula, in: 2010 IEEE Transforming Engineering Education: Creating Interdisciplinary Skills for Complex Global Environments, IEEE, 2010, pp. 1–26.
- [4] Y. Lizama, D. Varona, P. Waychal, L.F. Capretz, The unpopularity of the software tester role among software practitioners: a case study, in: *Advances in RAMS Engineering*, Springer, 2020, pp. 185–197.
- [5] N. Silvis-Cividjian, Awesome bug manifesto: Teaching an engaging and inspiring course on software testing (position paper), in: 2021 Third International Workshop on Software Engineering Education for the Next Generation, SEENG, IEEE, 2021, pp. 16–20.
- [6] T. Winters, The gap between industry and CS education, in: Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol. 1, ITICSE '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 2–3, <http://dx.doi.org/10.1145/3502718.3534205>.
- [7] T.O.B. Odden, R.S. Russ, Sensemaking epistemic game: A model of student sensemaking processes in introductory physics, *Phys. Rev. Phys. Educ. Res.* 14 (2018) 020122, <http://dx.doi.org/10.1103/PhysRevPhysEducRes.14.020122>, URL <https://link.aps.org/doi/10.1103/PhysRevPhysEducRes.14.020122>.
- [8] N. Doorn, T.E.J. Vos, B. Marín, Exploring students' sensemaking of test case design. An initial study, in: 2021 IEEE 21th International Conference on Software Quality, Reliability and Security Companion (QRS-C), 2021, pp. 1–6, <http://dx.doi.org/10.1109/QRS-C55045.2021.00161>.
- [9] T.E.J. Vos, N. van Vugt-Hage, Software Testing, Open Universiteit The Netherlands, 2019.
- [10] T.E.J. Vos, F.F. Lindlar, B. Wilmes, A. Windisch, A.I. Baars, P.M. Kruse, H. Gross, J. Wegener, Evolutionary functional black-box testing in an industrial setting, *Softw. Qual. J.* 21 (2) (2013) 259–288.
- [11] M. Mayeda, A. Andrews, Evaluating software testing techniques: A systematic mapping study, *Adv. Comput.* 123 (2021) 41–114.
- [12] L. Bijlsma, N. Doorn, H. Passier, H. Pootjes, S. Stuurman, How do students test software units? in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), IEEE Press, Virtual original Madrid, 2021, pp. 189–198, <http://dx.doi.org/10.1109/ICSE-SEET52601.2021.00029>.
- [13] Joint Task Force on Computing Curricula IEEE Computer Society Association for Computing Machinery, Software Engineering 2014 Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering A Volume of the Computing Curricula Series, ACM and IEEE, 2015, URL <https://ieeecs-media.computer.org/assets/pdf/se2014.pdf>.
- [14] T.J. Team, JUnit framework 5, 2022, URL <https://junit.org/junit5/>.
- [15] T.S. Team, Sikuli, 2022, URL <http://sikulix.com>.
- [16] Apache, JMeter, 2022, URL <https://jmeter.apache.org>.
- [17] C. Kaner, J. Bach, B. Pettichord, Lessons Learned in Software Testing: A Context-Driven Approach, Wiley, 2011, URL <https://books.google.nl/books?id=byZmT73R1a8C>.
- [18] J. Bach, A tester's syllabus, 2022, URL <https://www.satisfice.com/resources>.
- [19] Y.B.-D. Kolikant, Students' alternative standards for correctness, in: Proceedings of the First International Workshop on Computing Education Research, ICER '05, Association for Computing Machinery, New York, NY, USA, 2005, pp. 37–43, <http://dx.doi.org/10.1145/1089786.1089790>.
- [20] J. Tretmans, On the existence of practical testers, in: ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday, Springer International Publishing, Cham, 2017, pp. 87–106, http://dx.doi.org/10.1007/978-3-319-68270-9_5, Ch. 1.
- [21] N. Li, J. Offutt, Test oracle strategies for model-based testing, *IEEE Trans. Softw. Eng.* 43 (4) (2017) 372–395, <http://dx.doi.org/10.1109/TSE.2016.2597136>.
- [22] I. Schieferdecker, T. Ritter, Advanced software engineering, developing and testing model-based software securely and efficiently, in: Digital Transformation, Springer Berlin Heidelberg, Berlin, Heidelberg, 2019, pp. 353–369, http://dx.doi.org/10.1007/978-3-662-58134-6_21, Ch. 1.
- [23] B. Marín, C. Gallardo, D. Quiroga, G. Giachetti, E. Serral, Testing of model-driven development applications, *Softw. Qual. J.* 25 (2) (2017) 407–435.
- [24] J.A. Whittaker, Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design, first ed., Addison-Wesley Professional, 2009.
- [25] C. Kaner, Exploratory testing, in: Quality Assurance Institute Worldwide Annual Software Testing Conference, 2006, pp. 1–14.
- [26] J. Bach, Exploratory testing explained, 2003.
- [27] W. Afzal, A.N. Ghazi, J. Itkonen, R. Torkar, A. Andrews, K. Bhatti, An experiment on the effectiveness and efficiency of exploratory testing, *Empir. Softw. Eng.* 20 (3) (2015) 844–878, <http://dx.doi.org/10.1007/s10664-014-9301-4>.
- [28] J. Itkonen, M.V. Mäntylä, Are test cases needed? Replicated comparison between exploratory and test-case-based software testing, *Empir. Softw. Eng.* 19 (2) (2014) 303–342, <http://dx.doi.org/10.1007/s10664-013-9266-8>.
- [29] D. Pfahl, H. Yin, M.V. Mäntylä, J. Münch, How is exploratory testing used? A state-of-the-practice survey, in: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Association for Computing Machinery, New York, NY, USA, 2014, <http://dx.doi.org/10.1145/2652524.2652531>.
- [30] T. Mårtensson, A. Martini, D. Ståhl, J. Bosch, Excellence in exploratory testing: Success factors in large-scale industry projects, in: X. Franch, T. Männistö, S. Martínez-Fernández (Eds.), Product-Focused Software Process Improvement, Springer International Publishing, Cham, 2019, pp. 299–314.
- [31] T.O.B. Odden, R.S. Russ, Defining sensemaking: Bringing clarity to a fragmented theoretical construct, *Sci. Educ.* 103 (1) (2019) 187–205.
- [32] T.D. Cook, D.T. Campbell, W. Shadish, Experimental and Quasi-Experimental Designs for Generalized Causal Inference, Houghton Mifflin, Boston, MA, Boston, 2002.
- [33] A. Jedlitschka, M. Ciolkowski, D. Pfahl, Reporting experiments in software engineering, in: Guide to Advanced Empirical Software Engineering, Springer London, London, 2008, pp. 201–228.
- [34] N. Doorn, T.E.J. Vos, B. Marín, Test informed learning with examples assignments repository, 2021, URL <https://tile-repository.github.io/>.
- [35] T.E.J. Vos, N. Doorn, B. Marín, Test informed learning with examples, in: Proceedings of the 10th Computer Science Education Research Conference, CSERC '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1–2, <http://dx.doi.org/10.1145/3507923.3507924>.
- [36] M.J. Wilson, M.L. Wilson, A comparison of techniques for measuring sensemaking and learning within participant-generated summaries, *J. Am. Soc. Inf. Sci. Technol.* 64 (2) (2013) 291–306, <http://dx.doi.org/10.1002/asi.22758>, arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1002/asi.22758>, URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.22758>.
- [37] K. Buffardi, S.H. Edwards, A formative study of influences on student testing behaviors, in: Proceedings of the 45th ACM Technical Symposium on Computer Science Education, SIGCSE '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 597–602, <http://dx.doi.org/10.1145/2538862.2538982>.

- [38] J.L. Whalley, A. Philpott, A unit testing approach to building novice programmers' skills and confidence, in: *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114, ACE '11*, Australian Computer Society, Inc., AUS, 2011, pp. 113–118.
- [39] C. Kaner, S. Padmanabhan, Practice and transfer of learning in the teaching of software testing, in: *20th Conference on Software Engineering Education & Training (CSEET'07)*, 2007, pp. 157–166, <http://dx.doi.org/10.1109/CSEET.2007.38>.
- [40] M. Aniche, F. Hermans, A. van Deursen, Pragmatic software testing education, in: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE '19*, Association for Computing Machinery, New York, NY, USA, 2019, pp. 414–420, <http://dx.doi.org/10.1145/3287324.3287461>.
- [41] E. Enoiu, G. Tukseferi, R. Feldt, Towards a model of testers' cognitive processes: Software testing as a problem solving approach, in: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2020, pp. 272–279, <http://dx.doi.org/10.1109/QRS-C51114.2020.00053>.
- [42] M. Aniche, C. Treude, A. Zaidman, How developers engineer test cases: An observational study, *IEEE Trans. Softw. Eng.* 48 (12) (2022) 4925–4946, <http://dx.doi.org/10.1109/TSE.2021.3129889>.
- [43] P. Ralph, The two paradigms of software development research, *Sci. Comput. Program.* 156 (2018) 68–89.
- [44] D.A. Schon, *The Reflective Practitioner: How Professionals Think in Action*, Vol. 5126, Basic Books, 1984.
- [45] T.O.B. Odden, How conceptual blends support sensemaking: A case study from introductory physics, *Sci. Educ.* 105 (5) (2021) 989–1012, <http://dx.doi.org/10.1002/sce.21674>, arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1002/sce.21674>, URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/sce.21674>.
- [46] J. Tuminaro, E.F. Redish, Elements of a cognitive model of physics problem solving: Epistemic games, *Phys. Rev. ST Phys. Educ. Res.* 3 (2007) 020101, <http://dx.doi.org/10.1103/PhysRevSTPER.3.020101>, URL <https://link.aps.org/doi/10.1103/PhysRevSTPER.3.020101>.

Niels Doorn is a Ph.D. Student at the Open Universiteit and works as a Team Leader / Lecturer Researcher at the NHL Stenden University of Applied Sciences. He has a software engineering background and researches ways to improve software testing in different computer science educational contexts.

Tanja E.J. Vos studied computer science at the University of Utrecht (The Netherlands) and obtained her Ph.D. on formal verification in 2000 at the same university. Currently, she is an associate professor of the Technical University of Valencia (UPV) and a full professor at the Open Universiteit in The Netherlands. She has more than 20 years of experience with formal methods and software testing. Her research focuses on automated testing. She has been teaching for more than 20 years and has been involved in many research projects on software testing in an industrial setting. She has successfully coordinated various EU-funded projects like EvoTest 2006–2009, and FITTEST 2010–2013, and is involved in various Erasmus and Leonardo initiatives that try to help business understand academia and vice versa. Currently, she is coordinating the research, development and in exploiting of the TESTAR (www.testar.org) tool that came out of the FITTEST project.

Beatriz Marín, Ph.D. in Computer Science from Universitat Politècnica de Valencia - Spain (2011), with more than 10 years of experience in teaching, academic management, and leading research projects. She has been project manager in companies, senior researcher, professor of undergraduate and graduate students, director of research projects, and during the last 3 years, coordinator of research and master grade at Universidad Diego Portales - Chile. Nowadays, she is working as senior researcher of the Valencian Institute of Artificial Intelligence (VRAIN), Spain. She has more than 50 scientific articles in the software engineering area, particularly in the areas of conceptual modeling, software testing, model-driven software development, empirical software engineering, and gamification. She participates actively in scientific committees of national and international conferences, and also in journals of recognized prestige.