Lukkarinen, Aleksi; Lehtinen, Teemu; Haaranen, Lassi; Malmi, Lauri

# An Event Listener or an Event Handler? Students Explain Event-drivenness in JavaScript

*Published in:*
Proceedings of 21st Koli Calling International Conference on Computing Education Research, Koli Calling 2021

*DOI:*
10.1145/3488042.3488051

Published: 17/11/2021

*Document Version*
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

*Please cite the original version:*
Lukkarinen, A., Lehtinen, T., Haaranen, L., & Malmi, L. (2021). An Event Listener or an Event Handler? Students Explain Event-drivenness in JavaScript. In O. Seppälä, & A. Petersen (Eds.), *Proceedings of 21st Koli Calling International Conference on Computing Education Research, Koli Calling 2021* (pp. 1-10). Article 23 ACM. https://doi.org/10.1145/3488042.3488051

**ACM Reference format:**
Aleksi Lukkarinen, Teemu Lehtinen, Lassi Haaranen, and Lauri Malmi. 2021. An Event Listener or an Event Handler?: Students Explain Event-drivenness in JavaScript. In *21st Koli Calling International Conference on Computing Education Research (Koli Calling '21), November 18–21, 2021, Joensuu, Finland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3488042.3488051

# An Event Listener or an Event Handler?

## Students Explain Event-drivenness in JavaScript

Aleksi Lukkarinen
Aalto University
Espoo, Finland

Teemu Lehtinen
Aalto University
Espoo, Finland

Lassi Haaranen
Aalto University
Espoo, Finland

Lauri Malmi
Aalto University
Espoo, Finland

## ABSTRACT

When students in programming courses are taught event-driven programming (EDP) for the first time, they face new terminology and concepts that they should internalize. Moreover, they learn a fully new approach for reasoning about program logic and execution order. However, there is a lack of research in students' understanding of these concepts. In this paper, we describe a study, in which we asked web development students to explain their conception of EDP: what are the main concepts involved and how they interact. Moreover, we asked them to explain the execution of a short piece of JavaScript code that focuses on basic usage of events and event listeners. The answers, which we requested as concept maps and text, were analyzed using inductive content analysis. Our results clearly demonstrate shortcomings in the students' learning and illustrate various misunderstandings that they may have regarding EDP. Based on the findings, we give suggestions for improving the teaching of EDP.

## CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; • **Software and its engineering** → *Publish-subscribe / event-based architectures.*

## KEYWORDS

event-oriented, event-based, JavaScript, concept map, computer science education, programming education

**ACM Reference Format:**
Aleksi Lukkarinen, Teemu Lehtinen, Lassi Haaranen, and Lauri Malmi. 2021. An Event Listener or an Event Handler?: Students Explain Event-drivenness in JavaScript. In *21st Koli Calling International Conference on Computing Education Research (Koli Calling '21), November 18–21, 2021, Joensuu, Finland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3488042.3488051


*Koli Calling '21, November 18–21, 2021, Joensuu, Finland*

ACM ISBN 978-1-4503-8488-9/21/11…$15.00
https://doi.org/10.1145/3488042.3488051

## 1 INTRODUCTION

When using current web or mobile applications, students are frequently using software which has been implemented using *event-driven programming*[1] (EDP) techniques. Interactive graphical user interfaces (GUI) usually apply this technique, in addition to numerous other software that react on events (e.g., various embedded systems, services/servers, device drivers, and operating systems). Compared to procedural or object-oriented programming, which students often learn in introductory programming courses, EDP has a fundamental difference: The execution order in the program code is not anymore well-defined in the sense that if we know the program state (variables, stack, heap), we could, in general, conclude how the program execution proceeds in a deterministic way. Events triggered elsewhere are being dispatched to the listening programs, which then proceed to execute code in event handlers. Thus, the incoming event stream defines, at its own abstraction level, the program's behavior on each execution.

ACM and IEEE included EDP as a core topic within *Programming Fundamentals* knowledge focus group in Computing Curricula 2001 [24]. More recently, both Computer Science Curricula 2013 [23] and Computer Engineering Curricula 2016 [11] address EDP under several knowledge areas. Furthermore, Computing Curricula 2020 [25], which moves from knowledge-based learning to competencies, includes events as a draft competency of computer science under *Programming Languages* area: *"Write event handlers for a web developer for use in reactive systems such as GUIs."*

Computer Science Curricula 2013 includes knowledge unit *Fundamental Programming Concepts* under knowledge area *Software Development Fundamentals*, giving it the following description:

> This knowledge unit builds the foundation for core concepts in the Programming Languages Knowledge Area, most notably in the paradigm-specific units: Object-Oriented Programming, Functional Programming, and Event-Driven & Reactive Programming.

Thus, the curricular guidelines treat EDP even as a programming paradigm. Interestingly, Krishnamurthi and Fisler [14] challenge this position. They remind that the concept of a programming paradigm is not clearly defined and define procedural, object-oriented, and functional programming as *organizational characteristics* that define the overarching method of arranging program code into manageable units. On the contrary, event-drivenness is *a behavioral characteristic*, which makes it orthogonal to the organizational

[1] Sometimes known as *"event-based programming"* or *"event-oriented programming."*

ones. Therefore, event-driven programming (EDP) is, by nature, a different type of concept to teach and learn.

While students are likely to encounter examples and even simple exercises of EDP in the context of graphical user interfaces (GUI), their conception of what happens under the hood during program execution may often remain obscure. For instance, adding and activating a new button with a GUI programming framework, such as Java Swing or JavaFX, is carried out by using some library calls, following or modifying some example code. Yet, the complete picture of what actually happens remains too abstract. One may realize this easily when debugging an event-driven program, which requires understanding the code execution order carefully.

Despite the emphasis given to EDP in defining curricula and the challenges students face in practice, there is surprisingly little research carried out to investigate how students learn EDP. A recent mapping review by Lukkarinen et al. [17] identified numerous papers that support teaching and learning EDP, for instance, by simplifying existing graphics libraries. However, they found almost no empirical research that would investigate how students learn EDP and understand EDP concepts, or work that would evaluate the impact of various educational tools and pedagogical practices targeted to support learning EDP.

In this paper, we seek to address this research gap in a study that focuses on students' various understandings and misunderstandings concerning EDP concepts and how EDP works. Our context is an introduction to web development and programming course that is offered as an open course for people who would like to familiarize themselves with building basic web applications using JavaScript. The course has been given several times and the experience indicates that students face difficulties especially on the exercise round concerning EDP. Our goal is to better understand the background of these difficulties to enable us to design remedying actions or changes in course learning resources. Our research questions are the following:

- RQ1 What kinds of relations do students describe between EDP-related concepts?
- RQ2 What runtime behavior do students associate with EDP?
- RQ3 What EDP-related misunderstandings do students have?

We start by presenting the central EDP concepts that concern our study (Section 2), followed by a review of relevant literature in Section 3. Next, in Section 4, we present the course context, learning resources, and the exercises, from which we collected data. The results for those exercises are presented in Section 5. Finally, we discuss the implications of our findings in Section 6 and conclude in Section 7 with some suggestions for programming teachers.

## 2 CONCEPTS OF EVENT-DRIVEN PROGRAMMING

Whereas event-driven programming (EDP) has some fundamental ideas, the exact terminology and implementation details vary between systems and development environments. Although the course that is the context of this study (§ 4.1) also introduces server-side programming, the development environment for the students in our study is pure JavaScript inside the Internet browser. The course material covers the basics of what is visible to the web developer but only scratches the surface on what happens below that level. Consequently, in this study we are naturally focused on events and event handlers, whereas concepts related to dispatching events are of secondary importance. Below we present a brief list of fundamental EDP concepts as they are related to this study and intended to be understood in its compass.

- *Event.* (1) Any occurrence that is meaningful for an observer, or (2) an object that implements the `Event` interface [26] to hold information about such an occurrence and to represent it while being passed around and processed in the computer. In other contexts, the second meaning is also known as *a message*.
- *Event Listener.* A callback function that (1) the programmer writes and (2) is being called to respond to an event. In other contexts, it is also known as *an event handler*.
- *Adding an Event Listener.* Having an event listener[2] added to the event listener list of a node in a Document Object Model (DOM) tree [26], so that it will be called when a specific type of event is to be processed in relation to that node.
- *Removing an Event Listener.* Having an event listener removed from the event listener list of a node in the DOM tree.
- *Event Queue.* A data structure that holds events. Often they essentially resemble first-in-first-out lists, but some environments might impose priorities and other special conditions on their handling of events. Here, it refers to the JavaScript engine's part that holds events that are waiting to be processed.
- *Event Loop.* In general, a loop that waits for events to be available and then dispatches those events to be processed. Here, it refers to the JavaScript engine's part that reads events waiting in the event queue and starts to process a new one after the previous one has been fully processed (i.e., when the JavaScript call stack is empty). In other contexts, it is also known as *a message loop*, *a message pump*, and *a main loop*.

## 3 RELATED WORK

Event-drivenness itself is widely present in contemporary computerized applications, and, for instance, tools that are intended for event-driven programming (EDP) and application development are being researched. Also, in computing education, various tools and pedagogical approaches have been presented to help teaching and learning EDP. However, as found out by a recent mapping review of Lukkarinen et al. [17], not many publications exist about empirical research that would target teaching and learning EDP, and even fewer of them are concerned with event-driven programming directly instead of describing miscellaneous related results while primarily focusing on something else. The few works that contribute fragments to EDP-related knowledge based on empirical data analysis mainly focus on visual block-based languages such as Scratch, LaPlaya, and App Inventor [e.g., 5]. The other works address topics such as computational thinking [7], oral metaphors and allegories in problem solving [8], and learning object oriented programming [20].

A potential pitfall for novice programmers is fragile learning. That is, the student is capable of writing programs that produce

[2] The Document Object Model application programming interface also gives the possibility to pass in an object that implements the `EventListener` interface [26], that is, implements a `handleEvent()` method. However, this and the various options related to adding event listeners were outside the scope of the course.

correct results and behave as expected while still having fundamental misconceptions or uncertainty regarding the programming concepts used [12, 18]. One study demonstrating this in EDP context is reported in [16].

Even though misconceptions are a broadly researched topic in computing education [e.g., 22], there seems to exist a research gap when it comes to misconceptions regarding EDP. In a recent inventory of programming language misconceptions [2]—which includes misconceptions in JavaScript—no misconceptions related to events are listed. At the time of writing, the accompanying site[3] does not include any of the core concepts related to EDP, either. We hope that our present work is a starting point for mapping out misconceptions in EDP. However, until we have conducted research focusing on the misconceptions themselves, we refer to issues in students' understanding of EDP as *misunderstandings*.

This study exploits concept mapping[4] [e.g., 3, 4]—a method for both presenting information [e.g., 10] and studying one's understanding of a subject area. In addition to its numerous applications in education and research in general, also computer science educators have tapped into its possibilities [e.g., 27, 28]. Particularly, it has been used to boost and research on students' understanding of topics such as object-oriented programming [e.g., 1, 21] and input&output regarding computer architecture [15]. In addition, Keppens and Hay [13] discuss methods for assessing concept maps, and Hubwieser and Mühling [9] describe efforts towards identifying sub-graphs from them.

# 4 STUDY ARRANGEMENTS

## 4.1 Course Context

The course where the study was conducted is called Introduction to Web Development and Programming. The intended audience is lifelong learners who wish to learn the basics of programming and get familiar with web development. The course is taken by a varied audience, some of whom have earlier programming experience in another language, and some who are complete beginners. To complete the course, students use a bespoke online learning management system. The course material includes approximately 70 000 words of text, video explanations, interactive code visualizations, and automatically graded exercises.

The course material is divided into eight rounds. The first four cover the basics of browser programming including Hyper-Text Markup Language, Cascading Style Sheets, and JavaScript. The latter four rounds introduce the students to server-side programming with Node.js. Round 4 presents Document Object Model [26] and event-driven programming in JavaScript, focusing on *events* and *event listeners* (see § 2).

While completing the course, the students receive points from automatically graded exercises. Some of the exercises are smaller and solved within the browser, for example, multiple choice questions or coding exercises with few lines of code. In addition to the browser exercises, there are larger exercises with file submissions. The students complete them on their own computers, submit them to the course platform for grading, and if the automated tests pass, they are awarded points. The course has no final exam. Instead, the student needs to gather at least one third of the available points to pass the course, and at least 80 % to receive the best possible grade.

[3]See: https://progmiscon.org/concepts/, accessed July 27, 2021.
[4]Not to be confused with *group concept mapping* [e.g., 19].

## 4.2 Data Collection and Analysis

During the course implementation of Spring 2021 (see § 4.1 above), we tapped into the following four data sources to facilitate this study: An enrollment survey (§ 4.2.1), a concept comprehension exercise (E1, § 4.2.2), a code comprehension exercise (E2, § 4.2.3), and a course feedback questionnaire (§ 4.2.4). The two exercises were located at the end of Round 4, were the only exercises we graded manually, and awarded together about 2 % of the total course points.

We analyzed the collected data using inductive content analysis, where we built data-driven categories of students' responses to each task in the preliminary survey and exercises E1 and E2. The categories were refined during the analysis, until their match with the raw data was considered acceptable. Our primary goal for the analysis is qualitative, i.e. identifying various types of understandings. While we also report some numerical results of frequencies, their role is descriptive only and we do not seek to present them as generalizable results.

*4.2.1 Enrollment Survey.* To provide a context for the study, we utilized an enrollment survey that students took when beginning the course. It covered background information, such as students' age, gender, previous education, and programming experience. For this study, we added four questions (P1–P4) that addressed the students' preliminary knowledge in event-driven programming, providing data for RQ1 and RQ2. The first questions asked the student to explain, in their own words, how they understand the concepts *event, in general* (P1), *event, in programming* (P2), and *event-driven programming* (P3). The last question (P4) asked them to explain, if they could, the effects of the last line of the following code block (syntax was not highlighted), after it has been executed:

```
var c = 0
function f(e) {c = c + 1}
const n = document.getElementById("box")
n.addEventListener("mouseenter", f)
```

We categorized the responses into small sets of qualitative categories using inductive content analysis. Two researchers carried out the analysis until they reached a consensus on the categories of each response.

*4.2.2 Concept Comprehension Exercise (E1).* The first exercise, E1, requested students to explain (1) what concepts are related to event-driven programming, (2) what do those concepts mean, (3) what kinds of relations do they have with each other, and (4) what kinds of activities do they perform alone and together. The exercise provided data for all of our three research questions. We accepted answers including a textual description, a concept map, or both. We allowed the students to draw the concept map by hand and submit a photo of it. We also explained the basic idea of a concept map, emphasized that it is not a mind map, and gave an example map of 17 nodes and 27 described edges regarding university-related concepts, such as various people and institutional units.

In our analysis, we produced various descriptive statistics, such as answering method regarding diagram type and text, counts of nodes and various types of edges (process flow, data flow, structure, and ambiguous), lengths of textual answers, and numbers of

concepts present. Furthermore, to eliminate noise and differences in answering methods and thus make the answers easily comparable, we normalized them into class and activity diagrams in Unified Modeling Language (UML). We then analyzed those reductions for various characteristics regarding the relationships between the concepts as well as the process conveyed in the answer. For analyzing the process, we developed an analysis model regarding the steps present in the answers.

We chose UML diagrams as the target medium for the normalizations, as it is a standardized and well-known as well as easily capable to represent the necessary data. The normalization of students' answers happened on the basis of reducing them to the essence that they seemed to be trying to convey about event-driven programming (EDP). Because of the answers included both textual and diagrammatic expression as well as varying terminology, this process and its results were imprecise and subjective to some extent. To alleviate this uncertainty, another researcher reviewed a part of the normalizations for correctness.

*4.2.3 Code Comprehension Exercise (E2).* To address RQ2 and RQ3 more comprehensively, we added the second exercise: E2. In it, we presented the students with the program code in Listing 1 and urged them to try it in a web browser themselves. Afterwards, they were requested to describe, in their own words, (1) what the JavaScript code does after a button on the web page is clicked, and (2) what are the effects of a click of a button regarding the content of that web page.

```
<div id="a"><button id="b">1</button></div>
<script>
  function a(b) {
    let c = b.target;
    c.removeEventListener("click", a);
    let d = document.createElement("button");
    d.innerText = parseInt(c.innerText) + 1;
    d.addEventListener("click", a);
    document.getElementById("a").appendChild(d);
  }
  document.getElementById("b")
                .addEventListener("click", a);
</script>
```

**Listing 1: The JavaScript code to be explained in E2 (see § 4.2.3). The surrounding boilerplate HTML code is omitted.**

In our analysis, we modeled the program execution steps following a click of a button and how those steps are related to other program elements. We examined which steps were present in answers to either of the posed questions. We focus on what the included steps communicate about students' understanding of the program code, its execution, and its design. Two researchers identified the responses seeking for consensus how the steps were presented.

*4.2.4 Course Feedback.* In the hope of understanding the students' perspectives about the usefulness of E1 (§ 4.2.2) and E2 (§ 4.2.3) "for reviewing and deepening [their] understanding of event-driven programming," we augmented the course feedback form with a free-text answer field for both questions.

# 5 RESULTS

130 students enrolled in the course implementation, on which this study was carried out (§ 4.1). Of them, 85 students (65 %) completed the enrollment survey, of which 74 students (57 %) gave their research consent and identified themselves as being over 18 years old. This group contains all the students who answered to exercises E1 (§ 4.2.2) and E2 (§ 4.2.3), and the results in § 5.1 concern only it. Furthermore, 28 students answered at least one of E1 and E2. One answer to E1 was completely off-topic, so we excluded it from the study. After this we had 22 students, who answered to both E1 and E2. Two students answered only to E1, and 4 only to E2. Thus, the total number of answers was 24 for E1 and 26 for E2.

## 5.1 Enrollment Survey

As the course was targeted to life-long learners, the student background was quite different from a traditional CS1 or an introduction-to-web-technologies course. 38 (52 %) of participants were females and 34 (46 %) males; one student identified as "other" and one did not respond. The age range was as wide as 19–69 years (three did not respond), with a great majority (45) within the age range of 26–35 years and most others (13) in 36–45 years. The highest completed level of education varied as much: 2 had an elementary school degree, 11 a high school degree, 3 vocational, 23 bachelor's, 31 master's, and finally, 1 had a doctoral degree.

Despite the generally high level of background education, the programming experience was very limited. 57 students (77 %) identified themselves as novice programmers and 15 (20 %) as non-novices; 2 did not answer. On the other hand, 46 (62 %) had taken some programming course, and 24 (33 %) had taken none. This obviously reflects that most of them considered their programming skills obsolete. 26 students estimated that they had programmed less than 10 hours, and 20 estimated to have 11–100 hours of experience.

Based on the above information, we concluded that a great majority of participants could be considered novices. A substantial minority did have some relevant programming skills, but apparently not in JavaScript or in web development context.

Next, we investigated students' responses to the four questions on event-driven programming (see § 4.2.1). The first question (P1) concerned explaining *events, in general.* 41 students responded to this. Of those, 22 focused on *something* or *some actions* happening without specifying any context, 14 on human activities, such as social events, and 5 associated *event* with software.

The second question (P2) asked students to explain *events, in programming.* Only 28 students responded to this, which reflects that most students were novices. Of these, 10 responses focused on something happening in code level (e.g., *"a trigger for executing a function"*), 6 focused on user actions, such as mouse clicks, and 7 covered both code level and user action (e.g., *"this is an event that triggers certain actions, for example 'click' can calls a function which closes a popup"*). 5 responses did not identify any context for the event (e.g., *"something that happens after something triggers it?"*).

The third question (P3) asked students to explain event-driven programming. There were 17 responses with some understandable content: Only 2 responses were fairly comprehensive, thus indicating a good level of understanding of event-driven programming. 4 responses focused on designing a program to respond to events,

7 on triggering some action in program, and 4 on triggering some action without specifying relation to programming.

The final task (P4) concerned explaining the working of the code snippet (§ 4.2.1). The correct answer would explain that the final line adds an event listener to the box element and that each time the mouse cursor hits the box, the counter variable is incremented. There were 28 responses, of which only one was considered complete. Three more were correct otherwise but did not explain the role of the event listener. 10 responses explained that a function call is carried out when the user does something with the mouse (the cursor enters the box or a mouse button is clicked, the latter of which is a wrong event). 8 responses were clearly incorrect, while out of these, 4 were related to event listening. The rest, 6 responses, were lacking a proper explanation or were clearly incorrect.

In summary, the responses in the enrollment survey indicate that most students did not have any understanding of event-driven programming, a small group had some vague understanding, and only a few individuals demonstrated basic understanding of EDP.

## 5.2 Concept Comprehension Exercise (E1)

We present the results from the concept comprehension exercise E1 in four parts: descriptive statistics (§ 5.2.1), structure (§ 5.2.2), runtime behavior (§ 5.2.3), and special cases (§ 5.2.4).

*5.2.1 Descriptive Statistics.* From the 24 accepted answers that we received to the concept comprehension exercise E1, five were unsuitable for the same analysis as the others; we describe them separately in § 5.2.4 and have excluded them from other results. Of the other 19 answers, 10 (53%) contained only a diagram, 6 (32%) were textual, and 3 exploited both formats. Although we asked the students for concept diagrams, 6 diagrams (46%) were practically flow charts and only 7 (54%) resembled a concept diagram.

The lengths of the textual (parts of) answers varied approximately between 490 and 2600 characters, with the average of 980. The diagrams had between 5 and 26 nodes with an average of 12, and 4–35 edges (avg. 15). Most commonly, edges represented some aspect of procedural flow: Ten diagrams contained such edges, and they were the dominant edge type in five diagrams (38%), four of which were practically flow charts. The next-common aspect for the edges to describe was structure. These edges were present in seven diagrams and the dominant type in three (23%). In addition, three diagrams (23%) contained procedural and structural edges almost equally and thus did not have a single dominant edge type. Three diagrams contained edges, whose meaning was ambiguous or not given; they were dominant in two diagrams. Finally, only one diagram contained edges that represent data flow.

Of the 19 answers, all but one (95%) mentioned both the concepts of *event* and *event handler*; 12 (63%) mentioned both *event handler* and *event listener*. The occurrence that the event data represents was mentioned in 13 answers (68%). Both mouse (12; 63%) and keyboard (10; 53%) as input channels were practically equally frequent and were mentioned in a few answers that did not mention the occurrence (above). Eight answers (42%) referenced to nodes or elements of Document Object Model. Seven answers (37%) either mentioned or implied that *an event loop* exists or does something, and also seven answers listed specific mouse and keyboard events. Finally, only one to three answers acknowledged such concepts as *an event queue*, the `document.readyState` property, and the methods `setTimeout` and `setInterval`.

*5.2.2 Structure (RQ1).* This section answers to our first research question—the relationships that students identified between concepts related to EDP. Our exploration will focus on the following five areas: (1) Events, (2) Document Object Model (DOM), (3) the relationships between DOM and event, listeners, and handlers, (4) the relationships between events, event listeners, and event handlers, and (5) event listeners/handlers.

**DOM.** A few answers touched on DOM. Most of these answers only mentioned the document (Figure 1, A) or its elements (B) alone without discussing them further. However, one answer conveyed that the document contains elements (C), two answers explained that DOM contains nodes that can contain objects or elements (D), and finally, one answer depicted the document as containing elements that, in turn, contain properties (E).

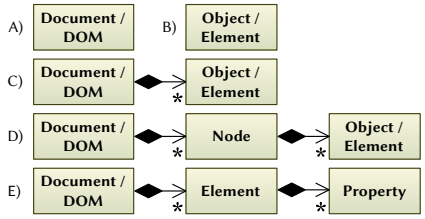


**Figure 1: Representations of the Document Object Model in students' answers (see § 5.2.2).**

On our course, DOM was discussed in the chapter preceding event handling. It was introduced using a quote from MDN Web Docs, which described it essentially as as a tree that consists of nodes that contain objects. The rest of the chapter presented ways to find, add, and remove elements, but did not discuss DOM's technical details further. Considering this, it is not surprising that the single-level descriptions (A and B) were the most common ones.

**Event.** One of the well-understood concepts was *an event* itself: 16 answers contained information representing a structure similar to Figure 2, part A, with at least one example. Most common examples were mouse and keyboard actions, and some answers also mentioned things such as changes on the web page, change of the state of an object, and passed time. Two answers characterized an event as a change of state (Figure 2, B).

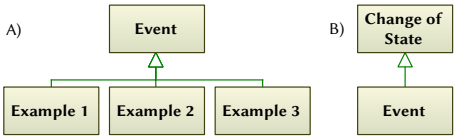


**Figure 2: The term *event* in students' answers (see § 5.2.2).**

As a contradicting example to the above, one respondent associated term *event* with the observable result of handling an event

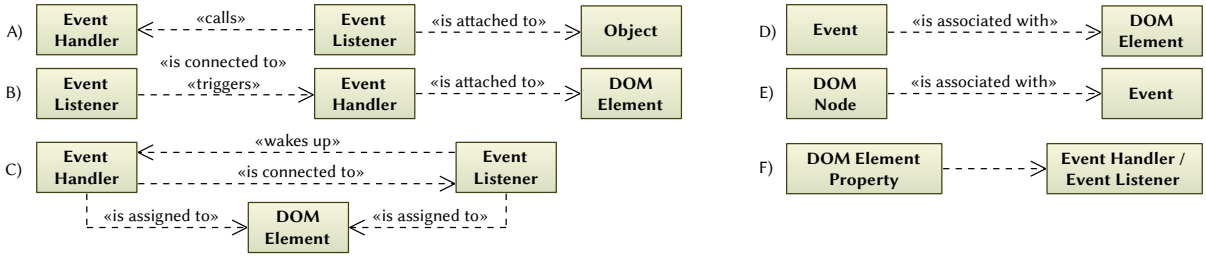


**Figure 3: Relationships between Document Object Model, *event listener*, and *event handler* in students' answers (see § 5.2.2).**

(i.e., the occurrence), that is, the side effects of executing the related event handler. To represent the occurrence that triggers the process of handling itself, this respondent used term *event trigger*.

Whereas the term *event* was used mostly correctly, only few answers implied the difference between the triggering occurrence and the data chunk representing that occurrence in the computer; most of the answers were ambiguous in this respect.

**Event Listener and Event Handler.** There seemed to be confusion between the terms *event listener* and *event handler*, and most of the answers used them without specifying what they actually are. Six answers described handler as a subprogram (Figure 4, A), one answer portrayed it as a *callback* that, in turn, is a subprogram, and one answer described both handler and listener to be subprograms (Figure 4, B). In general, the answers used some variant of the term *subprogram*, such as *function* or *method*.

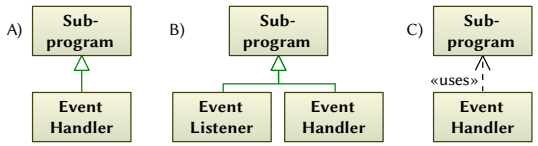


**Figure 4: The essence of the terms *event handler* and *event listener* in students' answers (see § 5.2.2).**

Interestingly, seven answers could be interpreted to mean that the subprogram to be executed in response to an event is neither an event listener nor an event handler, but yet another level of abstraction instead (Figure 4, C). Terms such as *uses*, *executes* (Figure 6, D), and *activates* (Figure 8) were used to describe the relationship between this subprogram and its parent.

**DOM vs. Events, Listeners, and Handlers.** The answers connected DOM to event-related concepts in many ways; we give some examples from individual answers. Our first three examples show interpretations of a situation that contains both event listener and event handler. The first interpretation was that an event listener is attached to some object and calls an event handler (Figure 3, A); another answer described the same attachment without the *call* relationship from listener to handler. The second interpretation (B) was that an event handler is attached to a DOM element, and an event listener is connected to the handler as well as triggers it. Naturally, the third interpretation (C) was that both listener and handler are assigned to a DOM element.

~~A pair of answers described events to be associated with DOM elements (D), whereas the other one had drawn this relationship the other way around (E).~~ Finally, one diagram contained a relationship from a DOM element property to an "event handler/listener" (F) but did not elaborate on the kind of that relationship.

**Events vs. Listeners vs. Handlers.** As with relationships concerning Document Object Model above, there were many versions of relationships between events, listeners, and handlers. The most common characteristic, in seven responses, was that an event listener listens to events and *calls*, or *wakes up*, an event handler (Figure 5, A) when it detects an event.

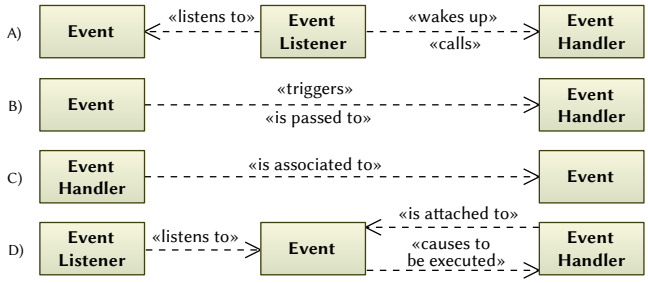


**Figure 5: Relationships between *event*, *event listener*, and *event handler* as depicted in students' answers (see § 5.2.2).**

The second-most-common version was more abstract: an event somehow triggers an event handler (Figure 5, B). One other answer explained that an event causes a trigger to be fired, and that the trigger causes an event handler to be executed. Other examples include an event handler being associated to an event (Figure 5, C) as well as the same as a more elaborated version with an event listener listening to an event and a recognized event causing an event handler being executed (Figure 5, D).

*5.2.3 Runtime Behavior (RQ2).* The following discussion answers our second research question—the runtime behavior that students associate with EDP. Most of the answers described only runtime behavior, but a few answers included development-time aspects as well. Thus, we will hereafter use "process" as a more general term. From the 19 answers included (§ 5.2.1) to the conceptual analysis, 1 was too ambiguous for even inferring a sensible process, and 2 did not present a clear order for all the steps of the process. 12 (63 %) conveyed a sequential process, of which 9 had two steps, 1 had three steps, 1 had four, and 1 six steps; we give examples of these in Figure 6. Finally, the last four (21 %) processes had complexities, such as loops and concurrency (e.g., Figures 7 and 8).

**CORRECTION: The stricken-through sentence above should be as follows:**
A pair of answers described the relationship between events and DOM elements. One answer stated that events are associated with DOM elements (D), whereas the other one had drawn this association the other way around (E).

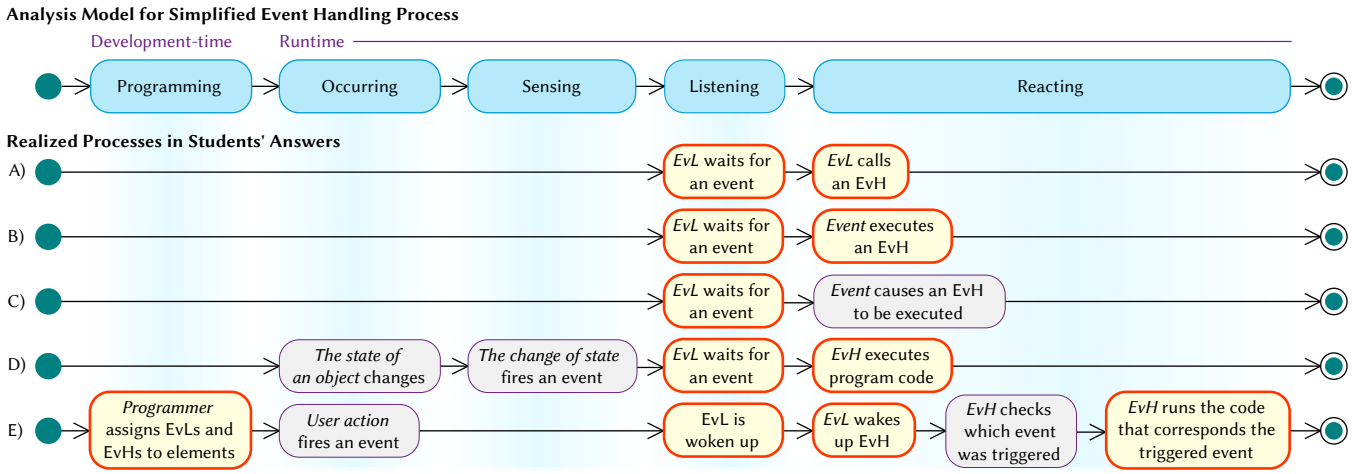


**Figure 6: An analysis model and examples of realized simple process descriptions present in students' answers (see § 5.2.3). In the actions, EvL means event listener and EvH means event handler. Actions containing misunderstandings have a thicker (red) border. The subject, where known, is emphasized for clarity.**

When analysing the 16 answers that conveyed a clear step order for the event-handling process, an analysis model (Figure 6, top) emerged. It contains two stages: *Development-time* and *Runtime*. Under them, there are five phases. The only phase in the Development-time stage is *Programming*. The others, which belong to the Runtime stage, are *Occurring*, *Sensing*, *Listening*, and *Reacting*. Under the model, the five example processes (A–E) illustrate (1) whole processes described in students' answers, (2) the positioning of the described process steps in relation to the analysis model, and (3) the misunderstandings that were present in the described processes. The first example essentially corresponds processes of three answers; the others are from individual answers.

All the examples in Figure 6 demonstrate misunderstandings related to *event listener:* Examples A–D explicitly state that (instead of *an event loop*) the listener would wait for events; in the last example, the listener is woken up. Then, examples A and E state that the listener would *call* or *wake up an event handler*; in example C, it is the event that somehow causes the handler to be executed, and example B claims that the event itself directly executes the event handler. Furthermore, the last example states that the programmer would assign *both* listeners and handlers (instead of just one of them) to Document Object Model elements. The correctness of the second step of example C is debatable. If the student were to use (only) the term *event handler*, the description itself would be correct. Still, they have also mentioned the *listener* in the first step, which—despite the lack of an explicit causal relation from the listener to the handler—arguably invalidates the second step.

A possible misunderstanding visible in examples D and E has been mentioned earlier, namely that the event handler would not be the final step of this simplified event-handling process and that there would be yet another subprogram—a level of abstraction—that is called by the event handler. This might be just a terminological issue, or it might be a misunderstanding about, for instance, the event-handling process or the entity that generally runs or executes program code (the processor).

**Event Queue and Event Loop.** Only one answer discussed *event queues* by stating that events are placed into it after being triggered, not elaborating on that. *Event loop* was present in seven answers. These include an answer that did not use the term itself

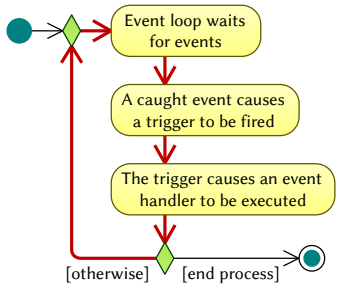


**Figure 7: A process from student's answer that incorporates the notion of event loop (see § 5.2.3). The effective event loop is emphasized with thicker (red) arrows.**

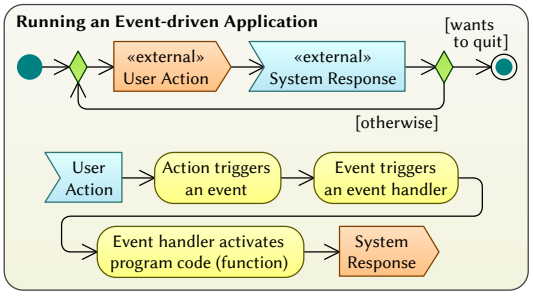


**Figure 8: A process from a student's answer that models both the user's interaction with the computer and the event handling inside the computer (see § 5.2.3).**

but implied its existence by using a similar looping structure in their diagram as in Figure 7. Thus, the student had understood the idea of repeatedly waiting for events until a specified condition occurs, although they failed to make its existence, function, and relationships explicit in their answer. A variation of this was the answer, whose process we show in Figure 7. In their diagram, the student had both clearly stated that the event loop waits for events (step 1) and used a loop with an ending condition. The diagrams of both students were essentially flow diagrams, and the looping process flow effectively becomes the true event loop.

In addition to the two cases above, one answer stated that the event loop executes event handlers, and another one had specified a dependency from *event* to *event loop* as well as from *event loop* to *user action*, but did not elaborate further. The rest three answers mentioned *an event loop* but without clear relationships to other concepts. However, one of these three answers conveyed the best general process description in this study (Figure 8). The diagram was essentially a flow diagram, and while the exact process and steps in it were ambiguous, it conveyed two concurrent looping and partly overlapping processes: One for the user as an event producer, and the other for the computer as an event consumer.

*5.2.4 Special Cases.* Unfortunately, not all of the answers suited well for this study. Here we describe the five special cases (S1–S5) that were excluded (§ 5.2.1) from the conceptual analysis.

To start with, two first paragraphs of one textual answer (S1) were essentially plagiarized from a Wikipedia article after changing some words—hence the rejection from further study. Interestingly, the Wikipedia's version was modified to reflect the student's misunderstanding about the terms *event listener* and *event handler*: "*– – main-loop, which listens for events, listener triggers a callback function, known as an event handler which triggers function for executing desired actions.*" Another similar case (S2) was a concept map, drawn mainly based on the same text in Wikipedia. This concept map contained a plain-language statement divided to a chain of edges and nodes, as well as an *event—event loop—callback* chain twice with different terms and one of them without explanations for the relations between the terms.

In addition, three answers were more or less off-topic. The first one of these (S3) was an exemplary process-oriented concept map—quite similar to our example—but unfortunately about concepts related to movies. The second answer (S4) was submitted by a student, who explained in the course feedback that they did not understand the two assignments and consulted their spouse about them. This resulted in an attempt to explain how the concept map we presented as an example would be related to event-driven programming. The third answer (S5) consisted of a simple concept map and a textual explanation regarding dogs and their environment. This answer was more about the essentials of object-oriented modeling, although it contained the basic idea of reacting to events.

## 5.3 Code Comprehension Exercise (E2)

In total, 26 students wrote an explanation for what the code in Listing 1 does once a button on the page is clicked. The answers to the second question about effects on the page content provided explanations comparable to those for the first question. Therefore, we decided to merge answers for the two questions to one analysis.

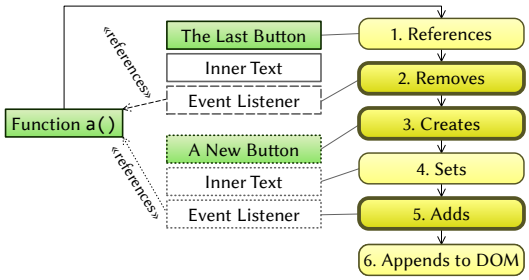


**Figure 9: The steps of executing function `a()` after the event loop has received a click that the function was registered to handle (see Listing 1 and § 5.3). The steps with thicker borders are most frequently included in students' explanations. Dashed lines mean removal, and dotted lines creation.**

All students explained some code or features of function `a()`. Twelve explanations (46%) explicitly mentioned that the function is executed to handle a click event, and six more explanations (69%) imply towards handling an event.

Next, we examined program execution step by step (Figure 9). 6 explanations (23%) included all the steps, and 15 more explanations (81%) skipped at most the first step, in which a reference is acquired for the clicked button. All but three answers (88%) included the execution steps that manage event listeners and create a new button, presented with thicker borders in Figure 9.

Two short and severely limited explanations started from creating a new button and setting its text (steps 3 and 4 in Figure 9). The second one additionally included appending the button to a parent element (step 6). One more substantial explanation included more (steps 2, 3, and 4), yet ignored adding an event listener to the new button (step 5). We consider these explanations problematic in that they do not support explaining and understanding how the function `a()` relates to the buttons and how the behaviour of the buttons changes over time.

A similar problem appeared also in the more complete answers commented above. Four explanations did not explicitly link removing and adding event listener to a function in the program. However, 2 of those explanations, and in total 12 explanations (46%), described that only the lastly added button remains clickable and will repeat the same procedure when clicked. Additional four descriptions (62%) implied to this behaviour with words such as *previous, last, next,* and *newly created.* When describing this program, such argumentation supports the hypothesis that the student understands the execution and management of event handlers.

Some answers did simultaneously display understanding of how the program works and confusion on how its execution internally proceeds. Two of them argued that calling `addEventListener()` waits for or detects clicks. A third one described that event listener returns the program back to function `a()`. A fourth one more specifically described blocking behaviour in claiming that the function continues until the button is clicked.

Finally, one description stated that the repetition of the same procedure when clicking a new button creates an event loop. It is evident that this student did not understand the concept of an event loop. Additionally, 8 explanations included a surplus description of

attaching the event handler for the first button when the program starts, whereas the task itself included explaining the code and its effects only after a button was clicked.

### 5.4 Course Feedback

14 students with research consent gave feedback (§ 4.2.4) about E1. It was positive from nine students, neutral from two, and negative from three. One student reflected about the usefulness of realizing the incompleteness of their knowledge and having to think what the concepts really mean: *"I think it was useful, because you really had to think about it and by doing so might realize that you don't actually understand it (yet). I don't LIKE doing these types of exercises, but that doesn't make them any less useful :)"*

For E2, 11 students with research consent gave feedback: nine positive answers as well as one neutral and one negative answer; three students did not respond. One of the students thought the same way as the above one did about E1: *"Very useful as I noticed that I wasn't sure e.g. the terminology even if I thought I understood the consepts."* Another student highlighted the importance of correct terminology and deep understanding in (professional) communication: *"It was good to have to do some own research here, it enables a programmer to explain to a non-programmer some of the logics going on behind the code."*

The negative and neutral feedback to these questions contained complaints about the uselessness of these types of exercises compared to coding and that the grading was not as transparent as in exercises that were graded automatically.

## 6 DISCUSSION

### 6.1 Findings

The results of our study illustrate that the participating students had fundamental confusions related to event-driven programming (EDP). Below, we answer to our third research question by discussing the most apparent ones of them.

**Event Listeners vs. Event Handlers.** Our earlier study [16] suggested that misunderstandings regarding the terms *event listener* and *event handler* could exist. The results of this study clearly confirm this and demonstrate them. In general, an event listener was overwhelmingly regarded as an active participant that *listens to* or *waits for* events as well as *calls* or *wakes up* a next subprogram. An event handler, in turn, was generally perceived as a passive target or a callback function that needs to be started by the preceding subprogram. Often the students used both terms and explained that a listener calls a handler, as if they would have assigned the *listener* to mean *an event loop* and then used *event handler* to pick up the meaning of *event listener*.

The course material states that *event listener* and *event handler* can be used synonymously. We hypothesize that a factor for the occurrence of the misunderstandings was a paragraph that the course material quoted from MDN Web Docs; it claims that "strictly speaking," a listener and a handler differ from each other.

**Registration and Execution of Event Listeners.** The above earlier study [16] also suggested that students might not understand what it means to register an event listener as well as when and by whom will the registered listeners be called. In addition to the claims in E1 about the event listener calling the event handler, four answers to E2 stated or implied that the `addEventListener()` method would wait for events instead of just adding the given function to the element's event listener list and returning. In the earlier study, a few students stated that the same method would call the event listener given to it as an argument.

These findings support the hypothesis that students might be confused about (1) both the idea and the internal details of registering an event listener as well as (2) exactly who is calling the listeners, when, how the program execution proceeds after the code of a listener has been executed, and how long the various parts of the program and the framework are being executed. When compared to a program that has no "dead code" and is executed from the beginning to the end, the event listeners registered to Document Object Model (DOM) elements appear as dead code, as the student's program itself does not call them. However, the fact that they still *are* being called implies that the execution model of such a program is different and thus the notional machine in the student's mind should be as well. Also, the misunderstandings suggest that the students might have difficulties in applying their existing notional machines to EDP, or in augmenting them for it.

**Terminology.** The terminology that the students used in their answers was inconsistent in general (see also [16]) and the exact meanings behind words were often debatable and open for interpretation. As an example, some of the answers gave the impression that the event listener, when listening events, would be actively running in the same sense as the engine of a car might be idling when the car is not being driven. In addition, an insufficient command of English might have influenced in some of the answers, such as the special cases S3–S5 (§ 5.2.4).

### 6.2 Threats to Validity

The target population in our study differs significantly from typical university-level introductory programming courses, as it represents a wide range of age groups, educational backgrounds, and programming experience. On the other hand, this work's primary approach is qualitative and we seeked to identify different types of understandings, including misunderstandings of EDP concepts. Such results shed light into this area regardless of the size and characteristics of the target group. The numerical results of frequencies should be considered describing this data set only and we do not claim any generalization for them.

The categorization of the students' answers is subject to interpretation, as their use of terminology was not consistent and the presentation of the answers varied. We addressed this challenge by having another researcher to review the analysis of the primary analyst, after which these two negotiated the unclear cases to find a consensus concerning the categorization.

## 7 CONCLUSIONS

This study explores students' misunderstandings when studying the basics of event-driven programming (EDP). It contributes to CS teachers' knowledge of potential issues when teaching EDP and also gives one example of analysing students' understanding of EDP-related concepts. In addition, we give suggestions below for improving the teaching of EDP.

Based on the misunderstandings and varying terms in students' answers, we conclude that the choice of terminology is important. The already-familiar meanings of descriptive words such as *listener* and *handler* affect students' self-explaining and understanding of new material. For each concept, only one term should be used; synonyms should not be mixed. Supposedly, a factor contributing to the misunderstandings related to *listeners* and *handlers* was the term *listener* in the `addEventListener()` method, while the course material used both *handler* and *listener* interchangeably.

Students' programming experience from different environments might further the emergence of misunderstandings. For instance, the concepts and technical details of event handling might differ in comparison to the course's chosen programming environment. When the earlier programming environments of the course's population are known, it might be useful to offer points of comparison and spell out the differences between similar concepts of the environments, possibly as optional info boxes for those who have used a specific environment earlier.

When quoting external sources or referring to them, it should be evaluated if the students have enough knowledge to properly understand them and set them into a proper context. These sources—"definitive" or not—might contain unfamiliar jargon and unnecessary details, which might make them unsuitable for novices. Such sources can confuse students and contribute to the emergence of misunderstandings. While learning to understand technical documentation is an essential task for novices, the question is, how can we best prepare the students for it.

Our results show that the concept of an event in general is well understood, but the difference between the occurrence and its representation in computer should be made clear. Furthermore, other concepts in § 2, including their activities and relationships with their environment, should be explicitly explained while clarifying and reinforcing the correct ways of thinking. Concept maps might help in clarifying and helping students to remember the substance. The essence of the Observer design pattern [6] should be explained to clarify the idea of manipulating event listener lists. Potentially confusing points, such as the fact that at a specific time during the program's execution, not a single line of the code that the student wrote is necessarily being executed, should be explicitly clarified with examples. All in all, the teaching should try to proactively ensure that known misunderstandings do not happen.

## REFERENCES


[1] Marc Berges, Andreas Mühling, and Peter Hubwieser. 2012. The Gap between Knowledge and Ability. In *Proc 12th Koli Calling* (Koli, FI). ACM, New York, NY, USA, 126–134. https://doi.org/10.1145/2401796.2401812

[2] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, et al. 2021. A Curated Inventory of Programming Language Misconceptions. In *Proc 26th ITiCSE* (Virtual Event, Germany). ACM, New York, NY, USA, 380–386. https://doi.org/10.1145/3430665.3456343

[3] Martin Davies. 2011. Concept Mapping, Mind Mapping and Argument Mapping: What Are the Differences and Do They Matter? *High Educ* 62 (2011), 279–301. https://doi.org/10.1007/s10734-010-9387-6

[4] Martin J. Eppler. 2006. A Comparison between Concept Maps, Mind Maps, Conceptual Diagrams, and Visual Metaphors as Complementary Tools for Knowledge Construction and Sharing. *Inf Vis* 5, 3 (2006), 202–210. https://doi.org/10.1057/palgrave.ivs.9500131

[5] Diana M. Franklin, Charlotte Hill, Hilary A. Dwyer, et al. 2016. Initialization in Scratch: Seeking Knowledge Transfer. In *Proc 47th SIGCSE* (Memphis, TN, USA). ACM, New York, NY, USA, 217–222. https://doi.org/10.1145/2839509.2844569

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Obj-Or. Sw.* Addison-Wesley, Boston, MA, USA.

[7] Yu Guo, Aditi Wagh, Corey Brady, et al. 2016. Frogs to Think with: Improving Students' Computational Thinking and Understanding of Evolution in a Code-First Learning Environment. In *Proc 15th IDC* (Manchester, UK). ACM, New York, NY, USA, 246–254. https://doi.org/10.1145/2930674.2930724

[8] Jeisson Hidalgo-Céspedes, Gabriela Marín-Raventós, Vladimir Lara-Villagrán, et al. 2018. Effects of oral metaphors and allegories on programming problem solving. *Comput Appl Eng Educ* 26, 4 (2018), 852–871. https://doi.org/10.1002/cae.21927

[9] Peter Hubwieser and Andreas Mühling. 2011. Knowpats: Patterns of Declarative Knowledge – Searching Frequent Knowledge Patterns about Object-orientation. In *Proc 3rd KDIR (IC3K)*. INSTICC, SciTePress, Setúbal, PT, 350–356. https://doi.org/10.5220/0003689203580364

[10] Iyolita Islam, Kazi Md. Munim, Shahrima Jannat Oishwee, et al. 2020. A Critical Review of Concepts, Benefits, and Pitfalls of Blockchain Technology Using Concept Map. *IEEE Access* 8 (2020), 68333–68341. https://doi.org/10.1109/ACCESS.2020.2985647

[11] Joint Task Group on Computer Engineering Curricula, ACM, and IEEE-CS. 2016. *Computer Engineering Curricula 2016: Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering.* ACM, New York, NY, USA. Retrieved July 16, 2021 from https://www.acm.org/education/curricula-recommendations

[12] Cazembe Kennedy and Eileen T. Kraemer. 2019. Qualitative Observations of Student Reasoning: Coding in the Wild. In *Proc 24th ITiCSE* (Aberdeen, UK). ACM, New York, NY, USA, 224–230. https://doi.org/10.1145/3304221.3319751

[13] Jeroen Keppens and David Hay. 2008. Concept Map Assessment for Teaching Computer Programming. *Comput Sci Educ* 18, 1 (2008), 31–42. https://doi.org/10.1080/08993400701864880

[14] Shriram Krishnamurthi and Kathi Fisler. 2019. Programming Paradigms and Beyond. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press, Cambridge, UK, Chapter 13, 377–413. https://doi.org/10.1017/9781108654555.014

[15] Edurne Larraza-Mendiluze and Nestor Garay-Vitoria. 2013. Use of Concept Maps to Analyze Students' Understanding of the I/O Subsystem. In *Proc 13th Koli Calling* (Koli, FI). ACM, New York, NY, USA, 67–76. https://doi.org/10.1145/2526968.2526976

[16] Teemu Lehtinen, Aleksi Lukkarinen, and Lassi Haaranen. 2021. Students Struggle to Explain Their Own Program Code. In *Proc 26th ITiCSE* (Virtual Event, DE). ACM, New York, NY, USA, 206–212. https://doi.org/10.1145/3430665.3456322

[17] Aleksi Lukkarinen, Lauri Malmi, and Lassi Haaranen. 2021. Event-driven Programming in Programming Education: A Mapping Review. *ACM Trans Comput Educ* 21, 1, Article 1 (March 2021), 31 pages. https://doi.org/10.1145/3423956

[18] Sandra Madison and James Gifford. 2002. Modular Programming: Novice Misconceptions. *J Res Technol Educ* 34, 3 (2002), 217–229. https://doi.org/10.1080/15391523.2002.10782346

[19] Scott R. Rosas. 2016. Group Concept Mapping Methodology: Toward an Epistemology of Group Conceptualization, Complexity, and Emergence. *Qual Quant* 51, 3 (April 2016), 1403–1416. https://doi.org/10.1007/s11135-016-0340-3

[20] Steven D. Sheetz, Gretchen Irwin, David P. Tegarden, et al. 1997. Exploring the Difficulties of Learning Object-Oriented Techniques. *J Manag Inf Syst* 14, 2 (1997), 103–131. https://doi.org/10.1080/07421222.1997.11518167

[21] Ven Yu Sien. 2010. Implementation of the Concept-Driven Approach in an Object-Oriented Analysis and Design Course. In *Proc MODELS'10* (Oslo, NO). Springer-Verlag, Berlin, Heidelberg, DE, 55–69. Retrieved July 19, 2021 from https://dl.acm.org/doi/10.5555/2008503.2008511

[22] Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education.* Ph.D. Dissertation. Dept. of Comput. Sci. and Eng., Aalto U., Espoo, FI. Retrieved July 29, 2021 from http://urn.fi/URN:ISBN:978-952-60-4626-6

[23] The Joint Task Force on Computing Curricula, ACM, and IEEE-CS. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science.* ACM, New York, NY, USA. Retrieved July 16, 2021 from https://www.acm.org/education/curricula-recommendations

[24] The Joint Task Force on Computing Curricula, IEEE-CS, and ACM. 2001. *Computing Curricula 2001 Computer Science: Final Report.* ACM, New York, NY, USA. Retrieved July 16, 2021 from https://www.acm.org/education/curricula-recommendations

[25] The Joint Task Force on Computing Curricula, IEEE-CS, and ACM. 2020. *Computing Curricula 2020: Paradigms for Global Computing Education.* ACM, New York, NY, USA. Retrieved July 16, 2021 from https://www.acm.org/education/curricula-recommendations

[26] Web Hypertext Application Technology Working Group. 2021. DOM Living Standard. Retrieved Jul. 13, 2020 from https://dom.spec.whatwg.org/

[27] Wei Wei and Kwok-Bun Yue. 2017. Concept Mapping in Computer Science Education. *J Comput Sci Coll* 32, 4 (April 2017), 13–20. Retrieved July 16, 2021 from https://dl.acm.org/doi/abs/10.5555/3055338.3055341

[28] Xudong Yu and Steve Klein. 2008. Enhancing Student Learning Using Concept Mapping and Learning by Teaching Environment. *J Comput Sci Coll* 23, 4 (April 2008), 271–278. Retrieved July 16, 2021 from https://dl.acm.org/doi/10.5555/1352079.1352129