Teaching Event-driven Programming for Novice Programmers: Challenges and Advances

Aleksi Lukkarinen
Teaching Event-driven Programming for Novice Programmers: Challenges and Advances

Aleksi Lukkarinen

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2 of the school on 26 August 2022 at 12:00.

Aalto University
School of Science
Department of Computer Science
Learning + Technology Research Group (LeTech)
Many computer applications receive information about external events and produce responses. Such applications include web server software, many services used via Internet browsers, as well as most graphical user interfaces in devices from mobile phones and home appliances to general-purpose computers and industrial applications. The development of such applications is often called event-driven programming (EDP).

Despite the continuously increasing importance of event-driven applications in the modern society, only a little published experimental research has been targeted at teaching and learning EDP. This dissertation addresses this research gap.

The dissertation has two major parts, of which the first one addresses understanding EDP. As an introduction, it reflects on the difficulty of defining the term event-driven programming and continues by presenting a mapping review regarding published research on teaching and learning EDP. From there, it proceeds to explore students’ perceptions of concepts such as a button, an event, and an event handler. This discussion is based on three studies that involved questionnaires and exercises for students of two online courses. A few questions included reading program code, and in one exercise, the participants were asked to answer using a concept map. The analysis of these answers revealed fundamental misunderstandings regarding EDP and user interfaces. Based on them, the dissertation offers practical suggestions for improving the teaching practice.

The second major part of the dissertation addresses teaching EDP from two perspectives. First, it explores contextualization of teaching EDP and focuses on tools and exercises of an approach called media computation. Second, it presents a prototype of a tool that can be used for visualizing high-level concepts in many online learning materials. This tool supports logging users’ actions, plays well together with version control systems, has a relatively low learning curve for teachers of computer science, as well as is open source and free to use. The tool was evaluated in a small-scale pilot study, which demonstrated its suitability for its intended usage environment.


Avainsanat: tapahtumapohjainen ohjelmointi, käsitteet, väärinkäsitykset, viitekehyksellistetty opetus, visualisointi, ohjelmoinnin alkeisopetus

ISSN (painettu) 1799-4934 ISSN (pdf) 1799-4942
Julkaisupaikka Helsinki Painopaikka Helsinki Vuosi 2022
Preface

Unbeknownst to me, my work towards this dissertation started in 2015 when I began work on my master’s thesis project for LeTech research group in Aalto University: The fifth publication discussed in this dissertation was my first conference paper and is based on my work for that thesis. After graduating and working at the university for a while, I began doctoral studies on Computing Education Research. Although this dissertation represents the end of my formal studies, both my curiosity and my field of profession mean that for me, studying and learning will never end.

I am grateful to my supervisor, professor Lauri Malmi, for accepting me as a doctoral student after supervising my master’s thesis, actively guiding me in my studies, coauthoring three of my publications, and letting me to fulfill my desire to further educate myself on science and research as well as to contribute to the body of scientific knowledge through my research. Also, Lassi Haaranen was highly encouraging, supportive, and helpful as my advisor, coauthored five of my publications, and let me to collect research data from the course he was teaching.

Big thanks to Juha Sorva for the ideas for both my master’s thesis and my doctoral research project as well as for coauthoring my first conference paper. I thank Juha and Otto Seppälä for being my advisors regarding my master’s thesis, for originally hiring me to work at Aalto University as a research assistant, and for sharing their office with me for several years (although the Covid-19 epidemic emptied also that room for a long time). Thank you, Otto, for the long discussions regarding events and event-driven programming as well as for all your help during the years.

I also feel gratitude towards Arto Hellas and Teemu Lehtinen, both of whom I met the first time during my master’s thesis project. Arto has always been extremely encouraging, had the idea for the second publication discussed in this dissertation, allowed his course to be used for data collection, and coauthored the resulting publication. Teemu has helped me several times with the A+ learning management system and also let me to participate in writing his paper, which then became the third publication in this dissertation. Another Teemu—Teemu Sirkiä—shared an office with Teemu L., Lassi, and me while I was working towards my master’s thesis. He graduated from the research group when I was beginning my doctoral studies, but I always felt his encouragement. I thank each member of the
Learning + Technology research group (LeTech) for the positive spirit that there has been during the years.

I thank my pre-examiners, assistant professor André L. Santos and associate professor Nickolas Falkner, for their time and helpful feedback on improving my dissertation. Moreover, I am honored to have associate professor Petri Ihantola as my opponent for the public defense.

Lots of other people have enabled me to get this far. For instance, I have had the great luck of having many good teachers in the past at all stages of education. From the time of my studies towards a bachelor’s degree in software engineering, I especially want to thank Erja Nikunen for her help and encouragement, for jointly supervising my bachelor’s degree, and for hiring me as a project assistant when I began my thesis work.

I want to thank many others, including my parents, other close family, and friends, for cheering me up as well as encouraging and supporting me during my research. Thanks to Heli, Kauko, Anna, Nicholas, Julius, Isabella, Petteri, Lauri, Minna, Nguyễn Hạnh, Huỳnh Quỳnh, Trần Nguyên, con chó Lucky, Puppy, Whiskey, and Bấp. Finally, I am grateful to my fiancée Lê Giàu for believing in me and being patient with my studies. As your name implies, you are rich in love, and I am so lucky to have met you.

Espoo, Finland, June 26, 2022,

Aleksi Lukkarinen
Contents

Preface i
Contents iii
List of Publications v
Author's Contribution vii
List of Figures ix
List of Tables xi
Abbreviations xiii

1. Introduction 1
   1.1 Motivation 1
   1.2 Overview 3
   1.3 Scope 4
   1.4 Research Questions and Methods 4
   1.5 Relationship to Computing Education Research 12
   1.6 Structure of the Dissertation 13

2. Background 15
   2.1 Research on Introductory Programming 15
   2.2 Misunderstandings 19
   2.3 Visualizations and Notional Machines 20
   2.4 Aspects of Program Design 22
   2.4.1 Inversion of Control 22
   2.4.2 The Observer Pattern 23

3. Understanding Event-driven Programming 29
   3.1 Concepts and Nature of Event-driven Programming 29
      3.1.1 Perspectives to Defining EDP 30
      3.1.2 The Subarea of EDP Regarding PIII and PIV 35
   3.2 Past Research (PI) 36
   3.3 Students' Conceptions of Buttons (PII) 42
   3.4 Students' Conceptions of EDP (PIII & PIV) 46
      3.4.1 The Initial Clues: The First Publication (PIII) 47
      3.4.2 Digging Deeper: The Second Publication (PIV) 47
      3.4.3 Conclusions 49
4. Supporting Teaching

4.1 Contextualization (PV) ............................. 51
   4.1.1 Contextualization Tools in General (Class. I) .  52
   4.1.2 Tools for Media Computation (Class. II) .......  55
   4.1.3 Exercise Types of Media Computation (Class. III) .  57
   4.1.4 Conclusions ..................................  59

4.2 CSMV for Visualizing High-level Concepts (PVI) ....  59

5. Discussion ........................................  65

5.1 Contributions ....................................  65
5.2 Primary Outcomes ................................  66
5.3 General Observations ...............................  69
   5.3.1 The Difficulty of Defining EDP .................  69
   5.3.2 Experimental Research on EDP .................  70
   5.3.3 Notional Machines for EDP ....................  71
   5.3.4 When to Teach EDP ............................  72
5.4 Implications for Computing Educators .................  72
   5.4.1 Consider Everyday Meanings of Terms ..........  73
   5.4.2 Explain Similar Terms and Environments .......  74
   5.4.3 Have a 1–1 Match Between Terms and Concepts ..  74
   5.4.4 Consider the Suitability of External Resources ..  75
   5.4.5 Consider to Explain Even the “Obvious” ........  76
   5.4.6 Evaluate Actual Learning Gains .................  76
5.5 Limitations .......................................  76
5.6 Future Work ......................................  78

6. Conclusions .......................................  81

Bibliography ........................................  83

Appendices

A. Textbooks That Teach EDP in Connection with GUIs  111
B. Screening in the Study of PII ..........................  113
C. Screening in the Study of PIV .......................  115
D. The Meaning of the Verb “to Listen” ...............  117

Errata ..................................................  119

Publications .........................................  121
List of Publications

This dissertation consists of an overview and the publications listed below. In the text, the publications are referred to by their Roman numerals I–VI.


Author’s Contribution

Publication I
Event-driven Programming in Programming Education: A Mapping Review
I performed the searching and the rapid screening phases; all authors collaborated in detailed screening and data collection. I was responsible for the article and wrote most of the introduction and the methodological description, as well as the results concerning bibliometrics, publication streams, learning outcomes, and orderings of learning content. The other authors contributed evenly to writing the other sections and participated in finishing the whole article.

Publication II
A Kingdom for a Button: Students’ Thoughts about Buttons
The study was carried out based on the idea of the 2nd author, who planned survey questions Q2–Q4; Q1 was my addition to them. I made an initial analysis of the students’ answers, created an online questionnaire that all authors then used while analyzing the answers, and compiled final results based on the analysis data. I was responsible for the paper and wrote most of the methodology, results, and discussion sections. The other authors collaborated evenly on the introduction and the background as well as participated in finishing the whole paper.

Publication III
Students Struggle to Explain Their Own Program Code
The other authors planned the study and collected and analyzed the data; the 1st author was responsible for the paper. My contributions were to describe the exercise related to event-driven programming (E3) and, on the basis of analyzing students’ answers to that exercise, to discuss answering challenges in general and confusions related to event-driven programming. The other parts of the paper were written by the other authors.
Author’s Contribution

Publication IV

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

I planned the study with the 3rd and the 4th author; the former supported
the data collection. I analyzed the students’ answers regarding the con-
cept comprehension exercise (E1); the other authors analyzed the answers
for both the code comprehension exercise (E2) and the enrollment survey.
I was responsible for the paper and wrote most of the content on concepts of
event-driven programming; data collection and analysis; results regarding
concept comprehension and course feedback; findings; and conclusions.
I also contributed to introduction and related work. Other authors wrote
most of the other content and participated in finishing the paper.

Publication V

Classifying the Tools of Contextualized Programming
Education and Forms of Media Computation

This paper was written based on my master’s thesis, for which the 2nd
author served as one of the advisors. I was responsible for the paper, and
my primary contributions are the classifications of contextualization tools,
tools for media computation, and varieties of media computation. The other
parts of the paper were mostly written by the 2nd author.

Publication VI

Scripted Step-based Visualizations: A Pilot Study

I implemented the visualization tool based on the idea from the 2nd author,
analyzed the collected data, was responsible for the paper, and wrote most
of the content. The other authors contributed fragments of content and
participated in finishing the paper.
List of Figures

Introduction
1.1 Positioning of the publications in regards of approaches to teaching; tools and frameworks; and learning outcomes ........ 3
1.2 Structure of this dissertation ................................................... 13

Background
2.1 The Observer pattern ............................................................... 25
2.2 The Observer pattern applied to listening events in DOM ........ 26
2.3 Event listeners being added and called with DOM ................. 27

Understanding Event-driven Programming
3.1 Author-based publication streams (PI) ................................. 38
3.2 Annual counts of first publications (PI) ................................. 39
3.3 Educational levels of publications (PI) ................................. 40
3.4 Students referring to the essence of a button (PII) ................. 44
3.5 Relationships between DOM, event listener, and event handler in students’ answers (PIV) ................................. 49

Supporting Teaching
4.1 Tools for contextualized programming education (PV) .......... 53
4.2 Types of tools for media computation (PV) ............................ 56
4.3 Visualizing client–server communication using CSMV (PVI) .... 61
4.4 Visualizing event-driven programming using CSMV (PVI) .... 63

Appendices
B1 Screening of students’ responses (PII) ................................. 113
C1 Screening of students’ responses (PIV) ................................. 115
List of Tables

Introduction
1.1 Simon’s themes related to the publications .................. 13

Understanding Event-driven Programming
3.1 Examples of scaffolding program code libraries ............. 41

Supporting Teaching
4.1 Examples of software-based contextualization tools ........ 55
4.2 Types of media computation exercises (PV) ................ 58
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets.</td>
</tr>
<tr>
<td>CS</td>
<td>Computer Science.</td>
</tr>
<tr>
<td>CSEd</td>
<td>Computer Science Education.</td>
</tr>
<tr>
<td>CSMV</td>
<td>Client–server Messaging Visualizer.</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets.</td>
</tr>
<tr>
<td>DMX</td>
<td>Digital MultipleX.</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model.</td>
</tr>
<tr>
<td>ECTS</td>
<td>European Credit Transfer and Accumulation System.</td>
</tr>
<tr>
<td>EDP</td>
<td>Event-driven Programming.</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface.</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language.</td>
</tr>
<tr>
<td>I/O</td>
<td>Input / Output.</td>
</tr>
<tr>
<td>ISCED</td>
<td>International Standard Classification of Education.</td>
</tr>
<tr>
<td>MIDI</td>
<td>Musical Instrument Digital Interface.</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-oriented Programming.</td>
</tr>
<tr>
<td>RQ</td>
<td>Research Question.</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus.</td>
</tr>
</tbody>
</table>
1. Introduction

1.1 Motivation

In the modern society, computers and computer systems have become a commonplace both in people’s private lives and in realizing the functionalities and processes necessary for supporting and developing the society as a whole. Many of the applications running on these computers—and the value-adding processes they realize—are not running in a vacuum in the sense that they would be started once and run independently to the end regardless of what happens around them. Instead, these software and processes are endlessly waiting for input from their stakeholders to tell them what to do, until they are told to stop waiting and to reach the end. In other words, their execution does not generally proceed identically every time, because they rely on runtime input to direct their execution.

Developing the kind of software that endlessly waits for instructions until it is told to stop is often called event-driven programming\(^1\) (EDP; see § 3.1). It is based on the idea of receiving, processing, dispatching, and creating information about events. An event is an occurrence that happens at a specific point of time. In programming jargon, however, that term often metonymically refers to some kind of representation or other knowledge of an occurrence that potentially is of interest to some observing parties and has been passed to them to enable them to react to it.

Graphical user interfaces\(^2\) (GUI) are often event-driven. They are common in a variety of modern devices from computerized wrist watches, mobile phones, televisions, and high-end home appliances to tablets, laptops, desktop computers, and industrial embedded systems. Especially, modern web applications available through Internet browsers are often designed to be event-driven.

Given the contemporary ubiquitousness of both GUIs in general and web pages as an application of the Internet in particular, it is necessary\(^1\) Also terms event-based and event-oriented have been used.
\(^2\) Naturally, corner cases, such as whether an application taking textual commands via a text field in a GUI is really based on a GUI, are arguable.
for software developers to become well-versed not only with event-driven programming (EDP), but with web development technologies as well. Even developers not working with web pages as such should know the basics of technologies such as *Hyper-Text Markup Language* (HTML) [171], *Cascading Style Sheets* (CSS) [172], and *JavaScript*—at least to understand their own discipline on a general level.

A more practical reason for being familiar with at least rudimentary web development technologies is that some development tools and frameworks enable the usage of technologies such as HTML, CSS, and JavaScript in developing user interfaces for applications that either do not look like web pages or are not such in the first place. They can be used to produce platform-independent user interface content and to allow application developers to use already-familiar technologies instead of forcing them to learn new ones. Moreover, JavaScript [165] and its derivatives, such as *TypeScript* [339], have become more important for web developers to learn, because, in addition to browser programming, they are now feasible options for back-end development as well.

Considering the widespreadness of potential event-driven systems, it becomes important for software developers to understand event processing. Regarding especially novice programmers, this importance is highlighted by software frameworks that call subprograms to handle events while encapsulating the low-level details. Often, novices begin studying programming by developing programs, in which every subprogram is called directly by the program itself. However, the ways of reasoning about the execution of these two types of programs differ because of the calls made by the frameworks. Consequently, the above kind of EDP requires the novice to learn and adapt to, for instance, (1) manipulating registrations of event handlers, (2) event handlers being called from outside the program, and (3) the fact that at a specific moment, not a line of an event-driven program might be being executed.

This dissertation will demonstrate that there has not been much research on (1) how students understand the concepts of event-driven programming and the runtime characteristics of event-driven programs, or (2) how to teach them. To alleviate this problem, this dissertation discusses the current state of research reported in journal articles and conference papers. It also reports empirical research on students’ understanding of event-driven concepts and has a constructive part regarding a visualization tool for conveying high-level ideas to students in online learning materials.
1.2 Overview

To help to advance the software industry and the society through it, this dissertation targets the education of future software developers. Its goal is to help teachers in improving their teaching of event-driven programming (EDP) in the following three ways:

1. By discussing students’ understanding of EDP-related concepts as well as runtime behavior of event-driven programs.
2. By describing and classifying
   a) ways to contextualize teaching of EDP
   b) approaches to one such contextualization method.
3. By providing a visualization tool that can be used to visualize high-level concepts, including those of EDP.

The dissertation contains six publications, often denoted as PI–PVI. Publication I is a journal article, and the five other publications are conference papers. Figure 1.1 presents these publications in terms of three non-mutually-exclusive knowledge areas: Approaches to Teaching; Tools and Frameworks; and Learning Outcomes.

- PI is a mapping review concerning journal articles and conference papers that address teaching and learning of EDP; it belongs to the intersection of all three areas.
- PII, PIII, and PIV, as studies on students’ understanding, belong solely to the Learning Outcomes area.
- PV is about contextualized teaching and tools suitable for it; that places it at the intersection of Approaches to Teaching and Tools and Frameworks areas.
- Finally, PVI presents a visualization tool that is not related to any specific teaching approach; thus, it belongs purely to the Tools and Frameworks area.

Figure 1.1. Positioning of the publications belonging to this dissertation considering three partly overlapping areas: Approaches to teaching; tools and frameworks; and learning outcomes (see § 1.2). The sizes of the areas are not meaningful.
Introduction

As Figure 1.1 shows (see also § 1.6), this dissertation describes the six publications in two parts: Chapter 3 discusses the mapping review and three papers concerning students’ understanding, whereas Chapter 4 presents two papers related to tools and approaches for teaching event-driven programming.

1.3 Scope

The spectrum of computer systems that can be argued to be event-driven is wide and eclectic, especially because it seems that there is not an unambiguous and generally-accepted definition for what event-driven programming (EDP) exactly is and what it is not (see § 3.1.1).

The studies in this dissertation concerning students’ understanding of EDP target a subarea of EDP with the following characteristics:

1. It focuses on graphical user interface (GUI) frameworks.
2. Event loops, event queues, and similar details are hidden by the framework.
3. When events occur, the framework passes information of the events to subprograms that the programmer has explicitly registered for that purpose.

An example of the above kind of situation is front-end web development using JavaScript and Document Object Model [170].

Moreover, this dissertation discusses GUIs mainly from the computer’s and the programmer’s perspectives. Furthermore, it is not concerned with any related areas, such as human–computer interaction, user experience design, concurrency, operating systems, interrupt handling, or other kinds of programs or systems that could be interpreted to be event-based, such as enterprise applications using message queues and programs written in a hardware description language. Finally, it does not take positions on the exact relationships between EDP and possibly related concepts, such as inversion of control (see § 2.4.1).

1.4 Research Questions and Methods

As the number of computerized processes and everyday applications used on mobile devices and via the Internet increases in the society, it becomes important for the software development student to acquire sufficient skills in the related technologies. Such technologies often contain event-drivenness, which highlights the importance of educators to ensure that their students are proficient in event-driven programming (EDP). For educators teaching EDP, it is helpful to know what kinds of literature the
scientific community has published in relation to teaching and learning event-driven programming (EDP). In addition, it is useful for researchers to be aware of possible research gaps that could form new research areas. These motivations are driving the first research question:

**RQ 1**

*What kinds of journal articles and conference papers are available on teaching and learning event-driven programming?*

Previous published work includes reviews and taxonomies regarding introductory programming education on numerous topics. Some efforts are general [e.g., 32; 114; 122], whereas some concentrate on its specific subareas, such as programming languages for novices [34], notional machines [45], software testing [134], as well as teaching and learning recursion [26]. Tool reviews include general discussion [e.g., 7; 19; 38; 115; 160] as well as more focused reports on topics such as learning programming online [1], automatic assessment [2] and plagiarism detection [31], algorithm [43] and program [46] visualization, games [44], and K–12 education [116].

Despite the large number of reviews, none of them are known to focus specifically on teaching and learning event-driven programming (EDP); the mapping review reported in Publication I is the first such study known to be published. It has six research questions [PI, page 4], all of which are repeated below as subquestions of RQ 1. The questions are answered in § 3.2 and form the base level for the EDP-related research in this dissertation. Pages 1–5 of PI discuss the background of the study further.

**RQ 1.1**

*Who has written about teaching and learning event-driven programming, when, and where?*

Information on authors as well as publication times and channels reveals insights into the research field. For instance, who are the most prominent researchers, what are the crucial publication channels to follow, and how has the research discipline been developing over the years.

**RQ 1.2**

*Which research methods have been used to study teaching and learning event-driven programming?*
Introduction

Knowledge of the earlier-used research methods helps to see what kinds of research in terms of research methods have been common or rare, what kinds of research are perhaps missing altogether, and how reliable, or trustworthy, the existing body of research on the field is as a whole.

**RQ 1.3**

*In which educational contexts has event-driven programming been discussed?*

Teaching and learning programming happen in a wide range of educational contexts, including courses on varying educational levels from kindergarten to career changers and lifelong learners. Educators, though, are primarily interested in research that is relevant to their own teaching. Knowledge of what educational contexts the literature covers and how well is helpful in finding relevant research to study. Furthermore, considering varying kinds of relevant contexts can reveal new approaches for improving one’s own teaching further. Finally, researchers can address any research gaps that there might be.

**RQ 1.4**

*What kind of pedagogical approaches have been reported to support learning event-driven programming?*

Publication I divides the pedagogical offerings of reviewed publications into three classes that are represented by the three latter subquestions (RQs 1.4–1.6). The first class—pedagogical approach—refers to teaching and learning practices and resources regardless of their involvement with event-driven programming (EDP). Considering and incorporating better approaches can potentially lead to better learning outcomes.

**RQ 1.5**

*What kind of software tools have been developed to support learning event-driven programming?*

The second class of pedagogical offerings (see RQs 1.4 and 1.6) are software and hardware tools that are designed to facilitate learning EDP. Taking advantage of better tools might produce better results, so it is important for both educators and researchers to stay informed of new and potentially better technologies for supporting education. Also, researchers might want to develop and try approaches that the literature does not cover yet.
RQ 1.6

**What empirical results of learning event-driven programming have been reported?**

The third class of pedagogical offerings (see RQs 1.4 and 1.5 above) are empirical results concerning both learning and motivational, emotional, and attitudinal changes. Such results are most valuable, because they can help to make well-informed educational decisions based on scientific evaluation. They also can direct future research by demonstrating, which approaches might not work well and which ones might be worth of further research and development. Some publications offer discussion, opinions, and anecdotal evidence without empirical evaluations, which makes their generalizability questionable.

As discussed earlier, event-driven programming (EDP) is an important area for students to learn, but Publication I demonstrates that little research exist on how students understand it. This research gap motivates the second top-level research question:

RQ 2

**What kinds of misunderstandings do students have related to event-driven programming?**

Many prior publications report on students’ misconceptions related to programming. For instance, the list of Sorva [164, Appendix A] includes 162 incorrect ways to think on various areas (see § 2.2) of introductory programming education. More recent work includes a literature review of Qian and Lehman [35] as well as the discussion of Chiodini et al. [71] on describing misconceptions and structuring collections of them.

Focus areas of earlier studies on students’ understanding regarding programming-related concepts include: Variables [89], object-oriented programming [133], parallelism [108], input/output subsystems [109], and the effects of alternative introductory courses [148] and modality (textual versus visual block-based languages) [152].

Unfortunately, earlier publications on conceptual understanding are not known to focus either on EDP or user interface controls (which often use events) in computing education. This dissertation addresses these two research gaps with subquestions RQ 2.1 and RQ 2.2:
RQ 2.1

*What kinds of conceptions do students have related to the workings of buttons in graphical user interfaces?*

One path that this dissertation takes towards helping programming teachers to improve their teaching is to explore students’ misunderstandings and assumptions regarding buttons as elements of graphical user interfaces. This research, reported in *Publication II* (§ 3.3), is qualitative and based on surveying participants of programming courses.

RQ 2.2

*In which different ways do students understand the concepts of event-driven programming and runtime behavior of event-driven programs? What kinds of misunderstandings do they have on these concepts?*

Another way this dissertation helps teachers is to report students’ misunderstandings regarding event-drivenness. This is approached from two perspectives: (1) concepts of event-driven programming (EDP) and (2) runtime aspects of event-driven programs.

This line of study is reported in *Publications III* and *IV* (§ 3.4), in which all the relevant research is qualitative and based on surveying participants of an introductory programming course. The course was relatively light and practical instead of discussing concepts theoretically and deeply on abstraction levels above the actual program code. Thus, students with no previous exposure to EDP might understand the concepts as they saw them in actual programs in the development environment chosen for the course. Consequently, there would not be a mismatch between theory and practice in students’ answers, and the above two perspectives would support each other both in the answers and as research targets.

Contextualized programming education is a method for decreasing dropout rates of computer science courses and, on the other hand, exciting the interest of potential new computer science students. Given the importance of learning event-driven programming, it becomes crucial to ask the third top-level research question:
Introduction

RQ 3

In which different ways has teaching of event-driven programming (EDP) been contextualized to increase students' motivation? What types of tools have been developed for media computation and how can media computation be used for contextualizing EDP?

Earlier literature on computing education discusses contextualized teaching both generally [e.g., 14; 100; 103; 111; 113] and regarding specific courses, tools, and themes [e.g., 10; 53; 62; 125; 127; 147]. Furthermore, some introductory programming textbooks [e.g., 185; 191; 192; 193] have been contextualized with some themes and tools. Still, known publications have not discussed contextualized programming education from the viewpoint of teaching event-driven programming (EDP).

Publication V discusses contextualized programming education by dividing it into themes, tools, and practices, of which contextualization tools are classified more extensively. Moreover, it focuses on media computation [e.g., 85], which is a teaching approach that uses multimedia as a contextualization theme. Regarding media computation, PV classifies (1) suitable teaching tools according to their technical nature as well as (2) genres and (3) exercise types found in four related textbooks [185; 191; 192; 193]. Based on these three classifications, § 4.1 discusses contextualizing the teaching of EDP.

Visualization in general is a broad field that includes software visualization—in other words, illustrations of computer software regarding, for instance, its structure, behavior, or evolution. For educational purposes, such visualizations concentrate primarily on the execution of (1) algorithms [43] and (2) program code [46] (see § 2.3). On these areas, numerous visualization tools\(^3\) have been developed, but they are highly specialized and realize only specific types of visualizations. As such, they are neither intended nor suitable for visualizing arbitrary high-level concepts. This gap in research and tooling leads to the fourth top-level research question:

\(^3\) [For instance, 23; 51; 84; 92; 93; 98; 99; 105; 118; 119; 136; 143; 144]
Introduction

RQ 4

What kind of a tool is suitable for visualizing high-level concepts of event-drivenness in online learning materials in such a way that

1. for a computing educator, the learning curve of the tool is relatively low
2. the tool and the visualizations play well together with version control systems
3. the visualizations can be easily shared with colleagues
4. students’ activities with the tool can be logged for research purposes
5. the source code of the tool can be customized by educators, and
6. the tool suits for environments, in which expenses and commercial software are not acceptable by principle?

In addition to algorithm and program visualization systems, previous work related to the above research question include many existing approaches that could fulfill a part of the criteria given in the question. Such approaches include the following:

- **Browser programming.** It is possible to develop visualizations by programming them oneself using various web development technologies. The basic languages are Hypertext Markup Language [171], Cascading Style Sheets [172], and JavaScript/EcmaScript [165]. Furthermore, there are languages\(^4\) that can be transpiled into some of these basic languages as well as JavaScript libraries\(^5\) that offer functionality for realizing animation and visualizations.

- **Presentation graphics applications.** It is a standard practice for educators to create learning materials using presentation graphics software\(^6\) and export them to formats that can be delivered to the students. Furthermore, the presentations could be shared via online services.\(^7\)

- **Dedicated applications for authoring eLearning materials.** There are applications\(^8\) that are purposefully created for authoring learning materials. Some of them enable educators to create non-linear course materials with several types of exercises, and the materials might be exportable to learning management systems and to Internet browsers.

\(^4\) For instance, Syntactically Awesome Style Sheets [351], TypeScript [339], Dart [316], CoffeeScript [306], ClojureScript [322], Scala.js [352], and Elm [308].

\(^5\) For instance, jQuery [344], Anime.js [315], Chart.js [305], and D3.js [303].

\(^6\) For instance, Google Slides [317], LibreOffice Impress [358], Microsoft PowerPoint [335], and Apple Keynote [299].

\(^7\) For instance: Scribd [353], SlideShare [354], SlideServe [355], Speaker Deck [357], Google YouTube [318], and Vimeo [363].

\(^8\) For instance: Adobe Captivate [298], Articulate Storyline 360 [300], Lectora [311], and iSpring Suite [325].
From the viewpoint of the research question, though, the above technologies have their limitations. Regarding browser programming, the favorable sides include having the greatest possible customizability, but it comes with a cost. Educators are often busy with their teaching and research, and even for professionals, it is tedious and time-consuming to first search for all the available tools, try and compare the tools to find the best ones for the problem to be solved, then learn properly how to use them, and finally develop something that solves the problem. Furthermore, although computing educators should be familiar with the basics of web development, not all of them are experts in it. Finally, it is not reasonable to have every educator to reinvent the wheel by developing their own solutions for the same problem.

Presentation graphics applications can be used to create visualizations, and some such applications are even free to use. However, when the files produced by them have to be used with software external to the learning materials, they are not really integral parts of the materials and introduce dependencies to external software. The files are often in some binary formats, which makes version control systems unable to present modification histories of those files. Video files are often large and thus inconvenient to store and deliver. Streaming services could be used to alleviate that problem, but they are often commercial and, again, make the learning material dependent on those services. Furthermore, the technologies used to view the presentations do not offer much customizability, the presentations themselves are customizable only in the limits of the features of the software used to create them, and logging students’ activities with the visualizations might be difficult or practically impossible.

eLearning authoring solutions offer more flexibility than presentation graphics software, but they also have larger learning curves and are often commercial. Their features limit the customizability of the materials created with them, and activity logging might be difficult or practically impossible. As with presentation graphics applications, they might save their presentations in binary formats, which makes them unfriendly for version control systems.

To answer the above problem, Publication VI presents a JavaScript-based open-source visualization tool that can be used to compose text and images into stepwise visualizations by writing simple scripts that describe the visualizations. These visualizations can then be embedded as integral parts of the web pages of learning materials, and they are instrumented to support activity data collection out of the box. As the scripts are textual, the tool is well-suited to situations where the source code of learning materials is kept in a version control system. Learning curve is relatively low for anyone who is familiar with the basics of web development, and still the program code of the tool is available for customization. The tool does not introduce dependencies to online services or other external software,
and as the file sizes are relatively small, the visualizations can be easily delivered to learners and educator colleagues.

The nature of the work described in Publication VI is primarily constructive, but a light empirical evaluation as a pilot study answers the following subquestion of RQ 4:

**RQ 4.1**

*How does such a visualization tool work in its intended environment, and what are students’ perceptions of such a tool?*

As an answer, the pilot study described in Publication VI provided practical experience from the educator's point of view as well as activity log data and feedback from students.

### 1.5 Relationship to Computing Education Research

To position this dissertation within the surrounding field of science, we will compare it to three classifications of subfields of research. First, Fincher and Petre [156] listed ten broad areas of motivation for computer science education (CSEd) researchers. Of these, the most relevant areas for the dissertation are discussed below.

- **Student understanding**
  
  Publications I, III, and IV contribute to the knowledge of students’ perceptions and misunderstandings of event-driven programming (EDP). In addition, the study in PII elicits knowledge of students’ misunderstandings and assumptions regarding graphical buttons.

- **Teaching methods; Educational technology**
  
  Publications III and IV further the knowledge of how EDP should be taught. Furthermore, PI, PII, PV, and PVI present approaches and tools for teaching EDP.

- **Animation, visualization, and simulation**
  
  Publication VI presents a visualization tool that can be used for teaching high-level concepts of EDP. Moreover, Publication V discusses media computation [e.g., 85], a teaching approach that uses, for instance, visual elements to represent the results of programs that students develop.

- **Transferring from campus-based teaching to distance education**
  
  Many teaching approaches and tools presented in this dissertation—especially CSMV in PVI—can be applied to distance education, for instance, via online learning materials.

In addition to the above areas, a newer classification of Simon [163] defines a set of common themes as a dimension in a classification for computing
education papers. The themes that are most relevant considering this dissertation closely correspond to the motivational areas of Fincher and Petre. Table 1.1 presents the relationships between them and the publications.

Table 1.1. The most relevant themes in Simon’s classification for computing education papers [163] from the viewpoint of the publications in this dissertation.

<table>
<thead>
<tr>
<th>Theme / Publication</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
<th>V</th>
<th>VI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ability / Aptitude / Understanding</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Online / Distance Delivery</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Teaching / Learning Techniques</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Teaching / Learning Tools</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Educational Technology</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Finally, Pears et al. [33] present four broad classes that partly map to the areas of Fincher and Petre. In terms of this classification, this dissertation contributes to Studies in Teaching, Learning, and Assessment; Institutions and Educational Settings; and Problems and Solutions.

1.6 Structure of the Dissertation

The primary parts of this dissertation are six publications (see § 1.4) and six chapters; they are being shown in Figure 1.2 below.

The next chapter discusses work related to this dissertation. Chapter 3 discusses past research related to event-driven programming (RQ 1), students’ conceptions of buttons (RQ 2.1), as well as students’ understanding regarding concepts of event-driven programming and run-time behavior of event-driven programs (RQ 2.2). Chapter 4 supports teaching event-driven programming by discussing contextualized teaching (RQ 3) and providing a visualization platform for high-level concepts (RQ 4). Finally, Chapter 5 summarizes and discusses the results, limitations, and future research, and Chapter 6 concludes the dissertation.
2. Background

This chapter briefly discusses the background and related work on topics related to the studies presented in this dissertation. It begins by positioning this dissertation within the research on introductory programming (§ 2.1). The following sections focus on specific themes from the viewpoint of this dissertation: misunderstandings (§ 2.2), visualizations and notional machines (§ 2.3), and aspects of program design (§ 2.4)—inversion of control (§ 2.4.1) and the Observer design pattern (§ 2.4.2).

2.1 Research on Introductory Programming

As the ubiquitousness of both computers and processes realized by them has increased in the society, the persons on the street have to interact with them in their everyday lives for both themselves and their employers—even if they were not developing computer applications themselves. Thus, it has become important for everyone to have an elementary understanding of what computers are and how they operate. This emphasizes the importance of Computing Education Research (CER) as a research discipline in finding effective educational approaches and tools.

Although CER is a relatively new research area in comparison to many other branches of science, it is nothing new when considering the history of practical computer devices themselves: In the late 1960s and during 1970s, programming was already being studied both as a human skill and in educational settings [157, p. 12–13]. Because the difficulty of learning to program was already known, researchers studied the nature of the issues as well as ways to alleviate them [157, p. 13].

One example of efforts towards helping novice programmers has been to develop simplified programming languages. An early example of such a language is Basic [331], which was developed in the first half of 1960s and can be seen as a derivative of Fortran (Formula Translator) [245, 15].

1 An acronym for Beginner’s All-purpose Symbolic Instruction Code.
Background

During the following decades, licenses for its various flavors were sold along many microcomputers, and educational children’s books on Basic programming [e.g., 189; 202; 246] were published. Later, Basic influenced the design of Visual Basic, which was tried for teaching graphical user interfaces and event-driven programming [e.g., 42; 59; 95; 155].

After the first version of Basic, other programming languages, such as Logo [242], Pascal, and Smalltalk, were developed to aid and teach novices both to program in its own right and to solve problems by programming [157, pp. 15–20]. Moreover, such tools as Boxer and the physical Logo turtle [158, p. 663; 242] as a programmable toy contributed to the emergence of modern visual block-based languages [157, pp. 19–20], such as Scratch [291], Snap! [293], and App Inventor [250]. Today, such languages are being used to introduce novices to (event-driven) programming.

Little by little, the researchers of computing education have been getting organized, as the following examples\(^2\) of first meetings, first journal issues, and establishments, ordered by year, demonstrate:

1970 • Technical Symposium of the Special Interest Group in Computer Science Education (SIGCSE) of the Association of Computing Machinery (ACM) started after SIGCSE was formed at the end of the 1960s [324].

1985 • Consortium for Computing Sciences in Colleges (CCSC) [323].
• Journal of Computing Sciences in Colleges (JCSC) [323].

1986 • Empirical Studies of Programmers group\(^3\) [157, pp. 24–25].

1987 • Psychology of Programming Interest Group (PPIG) [350].
• The first conference of CCSC [323].

1988 • Routledge’s Computer Science Education journal [18].

1996 • Australasian Computing Education (ACE) conference [141].
• Innovation and Technology in Computer Science Education (ITiCSE) conference [324].

2001 • Koli Calling International Conference on Computing Education Research [140].
• Transactions on Computing Education (TOCE) [4].

2005 • The International Computing Education Research (ICER) conference [324].
• The International Conference on Informatics in Schools: Situation, Evolution and Perspectives (ISSEP) [117].

2019 • Conference on Global Computing Education (CompEd) [73].
• The UK and Ireland Computing Education Research Conference (UKICER) [63].

\(^2\) Some entities have been renamed. For instance, CCSC was originally Consortium for Computing in Small Colleges (1985–2002) [323], and TOCE was Journal on Educational Resources in Computing (JERIC, 2001–2008) [47].

\(^3\) Meetings ceased after 1997 [157, pp. 24–25].
In 2015, Simon [163] compared the research field of Computing Education Research to Fensham’s proposed criteria [238] for recognizing science education as a field of research. Although the criteria are not authoritative and serve only as a checklist, the result of 13 of the 14 criteria being satisfied suggests that Computing Education Research (CER) can be called as a research discipline in terms of that criteria.

One broad area of CER is to study how to best teach programming for novices. A recent literature review of Luxton-Reilly et al. [114] regarding introductory programming reported topics of publications between 2003 and 2007 (inclusive). The included 1666 publications were assigned hierarchically to 4 primary groups, 17 secondary groups, and some of them further to 9 tertiary groups. The following discussion places this dissertation among these groups and serves as a glance at some research topics related to introductory programming.

**Students.** The first primary group focuses on students through three lenses as secondary groups: content, sentiments, and subgroups.

- **Content to be learnt.** This group contains four tertiary groups: theory (modeling student’s understanding), code literacy (reading, writing, and debugging program code), behavior (students’ activities) and ability (student’s performance on programming courses). As PII, PIII, and PIV all discuss students’ misunderstandings regarding both concepts and program code, they are related to theory and code literacy. Furthermore, because a pilot study described in Publication VI measures student’s usage of visualizations, it is related to behavior. Finally, Publication I presents earlier research on learning outcomes, which is primarily related to theory, behavior, and ability groups.

- **Sentiments of students.** This group contains four tertiary groups: attitudes, engagement, and experience. This dissertation is not directly related to either students’ attitudes or experiences, but there are connections to student engagement. First, contextualized teaching, discussed in Publication V and § 4.1, might increase students’ engagement to course materials. Second, improving the capabilities of the visualization tool described in Publication VI might increase students’ engagement to it. Finally, improving learning materials as discussed in § 5.4 could make them more understandable. When students understand the content, they might feel satisfied and safe—even excited. This could contribute to higher engagement with the leaning material.

- **Subgroups of students.** This group is concerned with specific student subgroups, such as those underrepresented in computing education, or the students who might be in the danger of failing courses. This dissertation is not directly related to such groups, but contextualized teaching, discussed in Publication V and § 4.1, could be used to make the learning content more interesting, approachable, and engaging for specific groups. Also, some publications reviewed in Publication I discuss groups such as non-majors.
Teaching. This dissertation is related to four of the five secondary groups of the second primary group teaching:

- **Models, taxonomies, and theories.** Both Publication V and § 4.1 discuss contextualized teaching and three related classifications. If the classifications are considered comparable to taxonomies, then this dissertation contributes to the secondary group called theories.

- **Tools that support teaching/learning.** The first two of the above classifications are concerned with tools that could be used to contextualize teaching, and Publication VI discusses a new visualization tool. In addition, Publication I discusses earlier research on tools and frameworks for teaching event-driven programming (EDP). Thus, this dissertation contributes to the secondary group called tools.

- **Techniques/activities used to teach.** § 5.4 intends to help in improving the teaching of EDP, and the third of the above classifications discusses exercise types of one contextualization method. Moreover, Publication I discusses earlier research on pedagogical approaches for EDP. These are contributions to a third secondary group: delivery.

- **The overall course structure.** Although this dissertation does not discuss course structures as such, Publication I presents earlier publications regarding various orderings of learning content, including objects-first, design-patterns-first, and events-first. In addition, some of the publications reviewed in PI present implementations of courses. Consequently, the dissertation is related to a fourth secondary group: orientation.

This dissertation has no direct connections to the fifth secondary group under teaching: Infrastructure (institutional and environmental support).

Curriculum. One of the two remaining primary groups is concerned with the curriculum, especially with the choice of programming languages and paradigms. § 3.1.1 discusses both defining EDP and areas to teach about it. In addition, § 5.3.4 offers remarks on positioning of EDP in the computing curricula. These associate this dissertation with the curriculum primary/secondary group.

If EDP were to be considered as a programming paradigm, this dissertation might be considered to be indirectly linked with the secondary group of paradigms. Even in that case, however, the dissertation would not be concerned with the choice of a paradigm to be used, and a discussion about a paradigm as such without an educational context does not fit to the groupings of Luxton-Reilly et al.

Assessment. The final primary group is concerned with ways of assessing students; theories and models of assessment; exam questions and responses; tools and methods that enhance feedback; as well as academic integrity and plagiarism. Some of the publications mentioned in Publication I are related to assessment. However, neither PI nor this dissertation explicitly discuss these research areas.
2.2 Misunderstandings

Learning to program in itself, not to mention learning the programming languages and tools to be used, involves a broad range of new concepts, terminology, and skills to be learned. One potential reason for students’ low success on programming courses are flaws in their understanding of the content they are supposed to learn. The literature describes a multitude of such flaws. For instance, regarding introductory programming courses, Sorva [164, Appendix A] catalogues 162 flaws with their source publications since the 1980s.

Researchers have used varying terminology to discuss flawed understandings. Sorva as well as Qian and Lehman [35] list terms such as bugs, difficulties, errors, incorrect understandings, misconceptions, mistakes, partial understandings, and student-constructed rules [35; 164, Appendix A]. Some authors have used term alternative conception to avoid possible negative connotations of term misconception [35]. Furthermore, terms naïve conceptions, naïve knowledge, and intuitive knowledge have been used to refer to prior ideas that prevent students from correctly understanding concepts being taught to them [35].

Although many of the above terms could be considered to be roughly synonymous, subtleties might exist in the meanings their users have associated with them. This dissertation relates to flaws in understanding through the studies presented in Publication II and Publication IV. To avoid claiming that the flaws discussed in them align with some nuances that might be associated with the existing terminology, this dissertation uses term misunderstanding when referring to those flaws.

The catalogue of Sorva [164, Appendix A] has been arranged into groups by topic. By choice, it covers only flaws that are common in introductory programming courses. It seems not to include misconceptions related to event-driven programming (EDP), but a few such misconceptions are reported in other sources. Two such sources are given below.

First, a dedicated website for misconceptions [71] features a draft [321] for a JavaScript-related misconception that executing `setTimeout(f, 0)` would equal to a synchronous call to `f()`. Although this misconception does not involve events that are visible to the programmer, it is related to events through a possible way of implementing the method by internally placing an event to the event queue of the JavaScript engine.

The topics in the catalogue of Sorva include: Variables, assignment, and expression evaluation; flow of control, selection, and iteration; subprogram invocations and parameter passing; recursion; references and pointers, reference assignment and object identity; object–class relationship and instantiation; object state and attributes; methods and method calls; other topics specific to object-oriented programming; and the overall nature of programs and program execution.
Second, a website [327] that gives advice on computer science education includes more examples of confusions related to event-driven programming. For instance, students might be confused about how event-related program code is being called [330; 329]. Furthermore, students might think that, in Scratch, a Repeat 1 block in an event handler (a script) would cause the event handler to be executed twice [326].

2.3 Visualizations and Notional Machines

The term visualization in general means5 making something visible, interpreting in visual terms, or forming visual images in one’s mind.6 A common application of visualization is to present any numerical information, such as statistics, in a graphical form. Another area is software visualization—in other words, visualization of computer software in terms of, for instance, its structure, behavior, and evolution. Whereas such visualizations are used by computing professionals in their everyday work, computing educators apply similar approaches in their teaching.

An ordinary source for visualizations is metaphors that are used in spoken languages and literature for conveying meanings by implicit comparisons.7 Although they can be visualized in the observer’s mind, they might also lend themselves well to the production of concrete visual representations. Computing educators have discussed metaphors in relation to, for instance, object-oriented programming [e.g., 40; 83; 149] and event-driven programming [17].

For a long time, various diagrams, such as flowcharts, have been used to describe processes, including the functionality of computer programs. Whereas even a quick sketch with a stick on sand can help in visualizing and conveying ideas, contemporary options include broad graphical modeling languages that provide devices for visualizing various kinds of computing-related aspects. For such languages, including Unified Modeling Language [343] and Business Process Model and Notation [342], there might be dedicated modeling software applications8 that might also generate program code and even try to update their models based on modifications in the code (known as round-trip engineering [e.g., 135]).

In addition to exploiting linguistical devices and self-drawn diagrams, computing educators have developed various kinds of software tools to visualize the material to be learned. Educational software visualization has

---

6 For instance, in meditation: Matko and Sedlmeier [25].
7 For instance: “You are my sunshine” and “I’m feeling blue.”
8 For instance: Visual Paradigm [364], MagicDraw [309], and Enterprise Architect [356].
two broad subareas: *algorithm visualization* (AV) and *program visualization* (PV) [e.g., 43; 46; 50; 114]. AVs usually appear on higher abstraction levels than PVs, which address concrete programs on the abstraction level of the program code. Both kinds of tools might have animation capabilities, and tools that support *visual program/algorith Simulation* [e.g., 99; 105; 144; 164] can let the student to practice the workings of the visualized content by interacting with it.

Program visualization can address both static and dynamic aspects. The former kind could target, for instance, block structures in the program code as well as relationships between subprograms, classes, modules, and applications. Visualizations of programs’ dynamic aspects (i.e., runtime behavior) include memory usage as well as execution of algorithms and programs, perhaps animated line by line. An example regarding event-driven programming is given by Čuvić, Maras, and Mladenović [76], who visualize (1) how events transform program state as well as (2) the execution of the program code in individual event handlers.

This dissertation is related to visualizations through a visualization tool discussed in Publication VI. It is helpful for the educator who desires to include visualizations of abstract concepts in the learning materials to help students to learn more effectively. In itself, the tool is neither related nor restricted to computing and could be used to visualize other subjects as well. However, this dissertation presents two usage examples related to program visualization. One of them visualizes runtime behavior of client–server systems in terms of their communication protocol, and the other one demonstrates visualizing the process of writing program code—that is, program’s static structures.

Regarding the abstraction level of the above protocol visualization, comparable tools include NetPrIDE [75] and GRASP [137; 138]. The visualizations for the tool of Publication VI are to be composed by the teacher by writing a script that describes the content of the visualization. Similar idea of scriptability has been present earlier in SALSA [92], but in contrast, that tool is intended to be used by the students.

One application of visualizations in teaching programming is to discuss *notional machines* [e.g., 45; 46; 77]. They fade away programs’ runtime environment, simplifying mechanics of executing programs to the abstraction level of program languages’ structures. Thus, the exact nature of a notional machine depends on the programming language being used. Programming languages and styles can have differing notional machines, and a single language can have many notional machines [46].

Dickson, Brown, and Becker [77, Figure 1] illustrate the problem of the novice programmer having an insufficient and perhaps incorrect model of a notional machine, which might cause them to have misunderstandings regarding the execution of programs. To alleviate this issue, educators teaching introductory programming courses should include learning about
a notional machine as an explicit learning goal [46]. Their simplicity, compared to the actual hardware, could result in more effective teaching [77]. This, combined with the generality of event-drivenness in computer systems, highlights the importance of researching into notional machines that would support understanding event-driven programming (EDP). Although the literature discusses notional machines for object-oriented programming [e.g., 41; 58; 112], it does not seem to address notional machines for EDP, save for the discussion of Čuvić et al. mentioned above.

2.4 Aspects of Program Design

As the two last areas of background information for this dissertation, the following sections will present fundamental ideas related to event-driven programming: Inversion of Control and the Observer pattern.

2.4.1 Inversion of Control

Inversion of control\(^9\) (IoC) [211, pp. 286–289; 241, p. 157; 314] is an abstract concept that can be applied on many levels in designing both individual programs and larger architectures. The fundamental idea is that something that would otherwise happen inside of some entity, is externalized to happen outside of it. In other words, the situation, in which the entity would control the behavior in question, is reversed so that the external party obtains the control instead.

One aspect of IoC is called dependency injection (DI) [209, pp. 128–136; 211, pp. 256–299; 223, p. 115; 241, p. 157], which is a crucial practice in writing loosely-coupled and testable program code: Instead of letting an entity to create or obtain (significant) resources it depends on, those resources will be arranged to be created or obtained by external parties that then provide them for the entity—inject them into it. Thus, the concern of creating and finding dependencies is separated\(^{10}\) into external parties, and the entity only declares what it needs, expecting someone to give it the necessary resources.

Another aspect of IoC is (partially) transferring the ownership of the control flow of a program to an external party. One example of this are

---

\(^9\) IoC is not to be confused with the Dependency-Inversion Principle (DIP) [240, pp. 127–134] that also involves the idea of inversion but targets dependencies between entities such as packages, modules, and layers.

\(^{10}\) This separation of concerns advances applying the Single Responsibility Principle (SRP) [240, pp. 95–98; 241, p. 157]. It also means that external parties, such as programmatic tests, can control what resources the entity uses to realize its behavior. In case of testing, the test code can inject resources that enable proper testing by, for instance, providing information in a controlled fashion or verifying that they are being called correctly during the tests.
callback functions [237, p. 208; 314]. As a small-scale example, a `map()` function might accept a function as its argument and call that function for each element of a data collection while passing the current element to the function as an argument. By using the `map()` function, its caller hands over—or abstracts away—the control regarding iterating the collection while retaining control over what happens regarding each element. Other small-scale examples of reversing the ownership of the control flow include design patterns Plugin [217, pp. 499–503], Strategy [218, pp. 315–323], Template Method [218, pp. 325–330; 314], and Observer (see § 2.4.2).

For broad architectural needs, there are frameworks\(^{11}\) that can take over responsibilities related to managing the control flow as well as injecting dependencies and assembling object graphs. Such frameworks might be configurable via files that declare, for instance, what objects to create and how. An example related to this dissertation is a JavaScript application running in an Internet browser as a part of a web page. The browser, effectively acting as an IoC/DI framework, will pass the control flow to the application code when appropriate. Especially, it will call functions that the application has registered for processing events, when it receives or generates events that the application is interested in.

In general, inversion of control (IoC) can be seen to describe and generalize some characteristics of event-driven programming (EDP) in case there is a framework that calls subprograms that are provided by the developer for handling events [314]. However, when the application code itself actively listens to and dispatches events instead of relying on a third-party framework, it becomes questionable if using the concept of IoC is reasonable anymore. This dissertation focuses on EDP and does not take a position on exactly how much of EDP could be generalized towards IoC.

### 2.4.2 The Observer Pattern

*Design patterns (hereinafter patterns)* are descriptions of common problems with proposed solutions [210, p. 5] that are based on experience and thus proven in practice. Grand [221, pp. 4–5] mentions two books about urban planning [234; 235], from which the idea of patterns has been applied to the field of computer science [e.g., 218; 247; 302]. Often, patterns are presented using a predefined structure\(^{12}\) that helps finding and understanding information as well as provides a consistent presentation over a publication.

\(^{11}\) Such frameworks are sometimes (interchangeably) called *IoC containers* or *DI containers* [211, pp. 256–299; 313; 314]. Tools that offer IoC/DI capabilities include .NET Framework [334], Castle Windsor [304], Dig [310], Guice [319], Spring [365] and Spring.Net [359], TSyringe [338], as well as TypeDI [361].

\(^{12}\) For instance, Buschmann et al. [210, pp. 19–21], Fowler [217, pp. 11–12], Gamma et al. [218, pp. 6–7], Grand [221, pp. 2–4], and Malveau et al. [226, pp. 49–60].
Background

Sometimes patterns are classified\(^{13}\) according to some criterion, such as problem type (e.g., creations, structural, and behavioral patterns \(^{218}\)) and abstraction level (e.g., architectural patterns \(^{210}, p. 12\)).

Knowing patterns has several advantages. In addition to providing ideas for improving the designs they address, the names given to well-known patterns can become standard terminology in professional communication \(^{210}, p. 6\) and help to convey the intent of the pattern quickly. Patterns can also support consistency and predictability as well as aid in managing complexity \(^{210}, p. 7\). However, patterns are only tools and do not guarantee anything as such \(^{233}, p. 7\). They should not be applied just because they exist, and if applied, the extent and the variant of the implementation has to suit to the current situation. Vlissides \(^{233}, pp. 3–10\) discusses some misconceptions related to patterns in general.

Some of the most well-known patterns are those described by Gamma et al. \(^{218}\). They are related to object-oriented design on the level of individual classes. In addition to object-oriented modeling, the term pattern has been applied to many other areas related to software development. Such areas include software processes \(^{206; 213; 226}\), use cases \(^{204}\), user interfaces \(^{360; 307}\), general program design,\(^ {14}\) games \(^{341}\), programming environments \(^{205; 208; 211; 219}\), programming language and compiler development \(^{216; 230}\), databases \(^{228}\), and testing \(^{227}\). In addition, Fowler \(^{239}\) as well as Ambler and Sadalage \(^{236}\) present refactorings using predefined structures similar to patterns in general, and Kerievsky \(^{225}\) discusses refactoring program code towards, and away from, specific patterns. Moreover, the term antipattern\(^ {15}\) has been used \([e.g., 213; 226]\) to discuss nonideal approaches.

Publication III and Publication IV in this dissertation are related to the Observer pattern.\(^ {16}\) It provides mechanics for maintaining one-to-many relationships between classes, so that classes acting as observers can listen updates from a class acting as a subject. Figure 2.1 presents the essentials of the pattern as presented by Gamma et al. \(^{218}\). Abstract class Subject maintains a collection of instances of classes that inherit from abstract class Observer. It also provides methods for both attaching and detaching\(^ {17}\) observers as well as to notifying all attached observers by calling their update() method defined by Observer. ConcreteSubject class,

---

\(^{13}\) This dissertation refrains from classifying patterns.

\(^{14}\) [For instance: 207; 209; 210; 211; 214; 215; 217; 218; 221; 222; 223; 224; 232].


\(^{17}\) In other words, adding and removing, or registering and unregistering.
which inherits from the abstract class `Subject`, maintains some state that
the observers are interested in. When that state changes, the `notify()`
method can be called, which causes the attached observers’ `update()`
methods to call the subject’s `getState()` method to retrieve the changes
in the state and to update the observers’ own states accordingly.

Figure 2.1. The Observer pattern [adapted from 218, p. 294] (see § 2.4.2).

Regarding event-driven programming in the studies of PIII and PIV, the
essence of the Observer pattern becomes important when considering the
addition and removal of event handlers. One method for the programmer
to associate event handlers with their callers is to pass the target of the call
(e.g., a reference to an object or including a subroutine) as a parameter to
a subroutine provided for the purpose. This is the method provided by Docu-
ment Object Model (DOM) [170], and it is presented as an application of the
Observer pattern in Figure 2.2. In the figure, `ConcreteEventListener`
instances are registered via `addEventListener()` method and are be-
ing called when information about events arrive. Compared to Figure 2.1,
there are two main differences: First, the abstract classes are replaced with
interfaces, which causes changes in the mutual relationships of the mod-
eled entities. Second, implementation details include (1) the arrival and
dispatching of events, including the counterpart of the `notify()` method,
as well as (2) the exact nature and location of the data structure that holds
information about added event listeners.

Figure 2.3 illustrates how the classes in Figure 2.2 work together runtime.
First, the event listener collection of the `ConcreteNode` is empty, and
when information about an event arrives, there are no listeners to call.
Afterwards, two instances of `ConcreteEventListener` get created and
added to the event listener collection. Consequently, the two event listeners
will be called when the next event arrives, if the criteria, including the
event type, for calling each listener is fulfilled.

The Observer pattern has also been known as Publisher–Subscriber,
Publish–Subscribe, and Dependents [210, p. 339; 218, p. 293]. Its named
variations and extensions include Multicast and Typed Message [233,
Background

Figure 2.2. The principle of the Observer pattern [218, p. 293] applied to listening events when working with Document Object Model [170] (see § 2.4.2). Non-essential details are omitted, and the actual calling mechanism of the event listeners as well as the exact nature and location of the data structure that holds information about added event listeners are implementation details.

pp. 123–144], Publisher–Subscriber (also known as Publish–Subscribe and Pub/Sub) [e.g., 210, p. 339–343; 211, pp. 442–463; 214, p. 217; 222, p. 106–110; 336], as well as Producer–Consumer, Gatekeeper, and Event Channel [210, p. 341–343]; some of these are for messaging between distributed systems. An example of another pattern that uses Observer is Model–View–Controller\(^{18,19}\) (MVC) for structuring interactive programs. Essentially, MVC divides the program code into three parts and applies a degree of decoupling between them by having them to listen updates from each other using the Observer pattern.

\(^{18}\) Discussed, for instance, by Buschmann et al. [210, pp. 125–143], Dovgal, Noriskin, and Olechko [214, pp. 391–392], Fowler [217, pp. 330–332; 312], and Kasampalis [224, pp. 93–101].

\(^{19}\) Other patterns similar to MVC include Presentation–Abstraction–Control (PAC) [e.g., 210, pp. 145–168], Model–View–Presenter (MVP) [e.g., 214, p. 392; 349; 312], and Model–View–View Model (MVVM) [e.g., 214, p. 393; 219].
Figure 2.3. An example of event listeners being added and called when working with Document Object Model [170] (see § 2.4.2). The classes correspond those in Figure 2.2. Non-essential details are omitted, and the actual calling mechanism of the event listeners is an implementation detail.
3. Understanding Event-driven Programming

As discussed in § 1.1, event-driven programming (EDP) is an important area to learn for a software developer. This importance is emphasized by the inclusion of EDP to the model curricula of Association for Computing Machinery and IEEE Computer Society: In Computing Curricula 2001 [175], EDP was included as a core topic under the knowledge focus group Programming Fundamentals. Both Computer Science Curricula 2013 [174] and Computer Engineering Curricula 2016 [173] address EDP under their knowledge areas, and Computing Curricula 2020 [176], highlighting not only EDP but also web development and graphical user interfaces, 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.”

This chapter will lay the foundation for teaching EDP—especially for novice programmers. It begins by discussing the concepts and nature of EDP (§ 3.1). Then, it will review the current state of research regarding EDP (§ 3.2) leaning on Publication I and answering RQ 1. The discussion will continue with novice programmers’ understanding about buttons (§ 3.3) on the basis of Publication II and answering RQ 2.1. Finally, the chapter ends by examining novices’ understanding of concepts of EDP and runtime characteristics of event-driven programs (§ 3.4), answering RQ 2.2 by drawing on Publications III and IV.

3.1 Concepts and Nature of Event-driven Programming

The term event-driven programming (EDP) seems to be difficult to define in an unambiguous and useful way, and this thesis refrains from trying to provide a general, overarching, and exhaustive definition for it. Instead, this section first discusses the difficulty of defining EDP (§ 3.1.1) and then presents the subarea of EDP that Publication III and Publication IV are concerned with (§ 3.1.2).
3.1.1 Perspectives to Defining Event-driven Programming

A simple attempt to define event-driven programming (EDP) could rely on concepts that are in its primary focus. Such concepts could include the following ones [adapted from PIV]:

- **Event**
  - (1) An occurrence that is meaningful for an observer, or (2) a set of information about such an occurrence for being passed around and processed in the computer. Examples of events include key presses, mouse movements, and changes in application life cycle. In other contexts, the second meaning is also known as a *message*.

- **Event handler**
  - A subprogram 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 listener*, and in some cases, as a *callback function*.

- **Adding (removing) an event handler**
  - In case the development environment requires or enables explicit registration of event handlers: Having an event handler registered (unregistered) with the environment so that the environment calls (stops calling) it when an event of, for instance, a specific event target and/or type is to be processed.

- **Event queue**
  - A data structure that holds information about events. Often it essentially resembles a first-in-first-out list, but in some environments, there might be priorities and other special conditions on event handling. In other contexts, it is also known as a *message queue*.

- **Event loop**
  - In general, a loop that waits for events to be available and then dispatches them to be processed by the possibly-added event handlers. In other contexts, it is also known as an *event-dispatching loop*, a *message loop*, a *message pump*, and a *main loop*.

In addition to listing concepts related to EDP, another way for approaching a definition is to elicit characteristics of event-driven systems. Hansen and Fossum [88] describe three such characteristics: (1) Loose coupling between events and their handlers (runtime registration of event handlers; multicasting and multiplexing of events in regard to their handlers; inverted control between high- and low-level units in programs), (2) state-based control, as well as (3) concurrent and distributed computing.

Although the above kinds of heuristics might hold in some environments and offer a rough picture of what EDP might mean for someone, there should be a practical and overarching definition that exactly and exhaustively states what EDP is and what is not. However, such a definition does not seem to exist, and trying to formulate one might not be easy. If event-
drivenness and event-driven programming (EDP) cannot be given such definitions and thus can mean, or be twisted to mean, almost anything depending on the user of the terms, one might justifiably ask whether the terms create confusions in communication and if it is beneficial to use them at all. They might be only buzzwords or all-encompassing misnomers for some more descriptive terminology, such as individual characteristics of the system being discussed.

The discussion below does not intend to define event-drivenness or EDP. Instead, it aims to provide various perspectives to the difficulties in giving such definitions. In other words, it does not intend to provide answers but rather to pose questions that could aid in formulating useful definitions for event-drivenness and EDP.

**Paradigm for Organizing Program Code in GUIs.** EDP has been called a *programming paradigm*, and some introductory programming textbooks (see Appendix A) bring up EDP when discussing graphics or graphical user interfaces (GUI) to present how to write subprograms to handle user input. The term *paradigm* is used even in Computer Science Curricula 2013 [174]:

> 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.

However, Krishnamurthi and Fisler [159] argue that event-drivenness is *a behavioral characteristic* and not about arranging the program code in a specific way. Thus, it would be orthogonal to *organizational characteristics* such as object-oriented and functional programming styles. The same authors also point out that we do not have a clear definition for the concept of a programming paradigm and thus cannot be sure if EDP really is one. Moreover, as Hansen and Fossum [88] argue, the idea of events is not restricted to graphics or GUIs but is visible in many computer applications including web servers, reactive systems, and operating systems.

**Waiting for Runtime Input.** Event-drivenness could be defined in such a way that the execution order of a program depends on runtime input that the program is waiting until a specific condition becomes true and the program ends. In this case, fundamentally, every piece of software that waits for input from its environment could be argued to be event-driven. Consequently, even reading console input with a one-liner program, such as `10 INPUT A$` (in GW-BASIC), `main = getLine` (in Haskell), `input()` (in Python), and `gets` (in Ruby), would be event-driven regardless of the method used to implement the reading of input. Goldwasser and Letscher [190, p. 494] present a similar Python example that can be reduced to the `raw_input()` call by omitting the production of output.
Understanding Event-driven Programming

**Abstraction Levels of Event-drivenness.** We could discuss event-drivenness on more than one abstraction level, and the interpretation of event-drivenness can depend on the level, on which it is observed and considered. For instance, in case of the above one-liner applications and the software framework that supports the programming language, there would be at least two such levels:

- **Inside or below the software framework.**
  To read input, the framework might call services of lower levels, such as the operating system and the firmware, or it might implement handlers for hardware interrupts. Furthermore, it might use an event loop itself for reading characters and returning a string, or it might rely on an event loop implemented on lower abstraction levels. From the perspective of the framework, definition for the term *event* could be either one character or one string of characters that has a known length or ends with the newline\(^1\) or NUL\(^2\) characters. Similarly, we could consider event-drivenness on the level of, for instance, the operating system and the device drivers, if these exist.

- **Inside the application using the software framework.**
  Disregarding everything below the application, we could define that *an event* means one input string from the above kind of one-liner and define a loop that reads strings that would be interpreted as commands or data where needed. This would correspond the event loop in applications for modern graphical user interfaces (GUI). From a list of given options, the user could then choose the function that they want to be executed next and then give the necessary parameters for it, like in modern GUIs.
  
  For instance, in case we wanted to emulate moving the mouse, we could enter a command `MM`, followed by a horizontal offset and a vertical offset. If we want to emulate a data entry dialog window, we could have one command per text field for entering/modifying data, or several commands per field where more complicated functionality is necessary. We could also have a command for entering a separate data entry mode and two others for exiting from it with acceptance or canceling; these would correspond opening and closing the dialog.

If we would not want to be artificially limited to reading input one string at a time, we could, for instance, use another kind of application programming interface, including the event loop in modern GUIs, or we could disregard the software framework for reading input and implement hardware interrupt handlers and event queuing ourselves.

---

1 Line Feed (LF) & Carriage Return (CR): ASCII/Unicode 10 (0xA) & 13 (0xD) [169].
2 Null (NUL): ASCII/Unicode 0 (0x0) [169].
**Graphical versus Text-based User Interfaces.** The above example illustrates the similarity between applications that have graphical user interfaces and console inputs. The programming style of reading events in an event loop and dispatching them to event handlers can be implemented regardless of whether the display device happens to be in a text mode or in a graphics mode, and regardless of whether the user interface is based on text lines or graphical objects such as windows, menus, and buttons.

**Events and Their Handlers.** The term *event* could be interpreted and characterized in many ways and reasoned about on many abstraction levels. Generally, it could be defined as an occurrence at a point of time. However, what exactly counts as such an occurrence? A broad interpretation would be that even the start-ups of a computer and a software application would be events, but then practically every piece of software would be event-driven, because they would be event handlers that handle the start-ups.

One way to define an event would be to think it as a change of a state. For instance, in a program that follows the Model–View–Controller design pattern (see § 2.4.2), the model could send events for any interested parties whenever its state changes. However, without further narrowing, the above definition could refer to any state wherever in the world and would not be more helpful than when referring to an occurrence.

Events could be also defined to be runtime input from the user, but events concerning matters such as timers, network, and application lifecycle are not necessarily immediate reactions to users’ actions. We could also define events to arrive from outside the program code that the programmer wrote for the application, but it seems unreasonable not to allow applications to create and dispatch events internally.

Situations related to programming techniques contain candidates for applying the notion of event as well. These include the following:

- **Machine language instructions.**
  We could also choose to treat the executions of individual machine language instructions as events. This could be especially feasible (1) when debugging a program in the assembly language of a physical processor and (2) when executing a program in a simulator or a virtual machine.

- **Method calls.**
  In object-oriented programming, calls of methods could be interpreted as messages to the related objects and thus as events arriving to those objects. Method calls could be feasible as events in aspect-oriented programming as well.

- **Iterations, repetitions, and callbacks.**
  Code blocks to be repeated in structures such as for and while statements as well as callback functions given as arguments to sub-programs such as map(), forEach(), and EnumWindows() could be
considered as event handlers that get called for each iteration. This extends to application programming interfaces (API) such as Simple API for XML that calls subprograms provided to them while iterating over an Extensible Markup Language (XML) document.

- **Exception handling.**
  Thrown exceptions caught with try and catch statements could be interpreted as records of events, and the code blocks in catch statements as event handlers.

- **Promises.**
  The callbacks for handling the success and the failure of a promise could be interpreted as event handlers and their calls as events.

- **Reactive programming.**
  In processing data streams, the chunks of data could be understood as events, and the parties observing and reacting to the streams would then be event handlers.

**Characteristics of Development Environments.** For various abstraction levels from interfacing hardware to simplified languages for introductory programming education, the development environments that enable or require event-driven programming (EDP) have varying EDP-related characteristics. The spectrum of variations in these characteristics contribute to the difficulty of formulating an unambiguous and usable definition for EDP. These characteristics might include some of the following ones:

- **The visibility of the details of event dispatching.**
  For instance, Scratch [291] and Visual Basic 6 hide these details, whereas developing with C language against Win32 application programming interface (API) requires the programmer to provide code for the event loop of the application [e.g., 243, p. 34; 244, p. 72; 332].

- **The registration of subprograms that process events.**
  For instance, Scratch and Visual Basic 6 do not require the programmer to explicitly manipulate registrations of subprograms that process events, whereas Document Object Model (DOM) [170] and Java Swing [346] generally do.

- **The mapping of events to the subprograms that process them.**
  For instance, when developing for Windows against Win32 API using C language, each window has one subprogram—a window procedure [e.g., 243, p. 14; 244, p. 75; 333]—that receives all events (messages) sent to that window. On the other hand, in Visual Basic 6, forms and controls can have one subprogram—an event procedure [e.g., 337]—per event type. To customize this mapping, the programmer can have subprograms to call other subprograms.
• The visibility of events’ meta information.
An environment such as Scratch might just call an event handler or otherwise tell that an event has occurred without passing any complementary information. As a contrast, environments such as Win32 API and DOM might provide information such as a timestamp, the source of the event, and the states of the mouse buttons and special keys on keyboard.

• The approach of passing events’ meta information.
These approaches include hardware input/output ports, processors’ registers, sets of arguments for subprograms, as well as data structures, including objects.

Other varying characteristics might include concurrency and parallelism related to receiving and processing events, but these are outside of the scope of this dissertation.

Event-drivenness vs. Event-driven Programming. Finally, we could ask about the relationship between event-drivenness and event-driven programming (EDP). Is an event-driven program always a result of EDP? For instance, if we claim that EDP is about writing event handlers but the programmer implements an event loop, the act of development might not be thought to be EDP, even if the program is event-driven.

To take the above idea further, what about developing a program that, for instance, implements a keyboard interrupt handler to read and enqueue input and has an event loop that reads events from the queue and calls event handlers to handle those events? We could argue that such a program contains event handlers on two abstraction levels (the interrupt handler [cf., 27; 88] and the ones called from the event loop), but because the programmer implements all the event-handling infrastructure themselves, the programming effort might not constitute as EDP.

3.1.2 The Subarea of Event-driven Programming Regarding the Studies on Students’ Understanding

As mentioned in § 1.2, this dissertation’s empirical studies PIII and PIV on students’ understanding of event-driven programming (EDP) focus on a subarea of EDP that could be characterized as follows:

• It focuses on graphical user interface (GUI) frameworks.
• The GUI framework hides event loops, event queues, and similar details of dispatching events to the receivers.
• When events occur, the GUI framework passes information about them to subprograms that it calls and that the programmer has explicitly registered for this purpose.

One example of the above kind of subarea is front-end web development using JavaScript and Document Object Model [170], which was the context
Understanding Event-driven Programming

for the data collection in PIII and PIV. Java Swing, for instance, is similar except that the subprograms that process events are methods of handler objects and not functions, as is often the case in JavaScript.

In the above subarea, the fundamental concepts for students to learn about event-driven programming include the following:

- Creating skeletal subprograms to act as event handlers.
- Filtering the received event stream by choosing the events for which to add event handlers.
- Adding and removing event handlers.
- Inspecting and exploiting the available information about the received events.
- Using event handlers to manipulate program state that is saved outside of them.

In other kinds of environments, the above list might be extended with other tasks. For instance, when programming using Java Swing framework, creating skeletal subprograms would include implementing a listener interface with the necessary methods or inheriting an adapter class and overriding the necessary methods. On the other hand, when programming with C language against Win32 application programming interface (API), the list might include writing an event loop in the program code included in (or called from) the WinMain() function [e.g., 243, p. 28; 244, p. 48; 340]—the entry point of Windows applications.

Finally, it would be useful to have knowledge of areas such as how the GUI framework executes the event handlers and what kind of path the event information roughly takes from, for instance, a mouse to the processing applications.

3.2 Past Research (Publication I)

As discussed in § 1.1, event-driven programming (EDP) is an important area for software developers to learn. However, teachers have expressed opinions on the novice programmers’ difficulty of learning EDP (see Publication I). To support both teaching and researching EDP, this section answers the first research question of this dissertation:

**RQ 1**

*What kinds of journal articles and conference papers are available on teaching and learning event-driven programming?*

The answer to the above question consists of answering its subquestions (below) on the basis of the study described in Publication I—a mapping
review that explores past research in teaching and learning event-driven programming (EDP). The following sections of this dissertation augment the review via new studies that target teaching and learning EDP.

**Publication I**, with its appendices, describes searching eight search engines and getting 858 search results, downloading 643 (75%) of them, narrowing the result set to 104 publications (12%), and accepting one additional publication that was suggested by a peer reviewer. These 105 publications were then analyzed, and the primary results were summarized. Some of the publications had become obsolete in terms of, for instance, their usage/discussion of technologies such as Java Applets [72] and Windows 3.0 [145]. Although these publications were classified as historical or semi-historical, they might still be relevant otherwise.

**RQ 1.1**

Who has written about teaching and learning event-driven programming, when, and where?

Bibliometrics and publication streams are discussed in detail in § 3.1 and § 3.2 of **Publication I**. Here it suffices to give an overview of publication streams, publication times, and publication channels.

**Publication Streams.** Among the publications, four publication streams were identified on the basis of having common authors and at least four publications (Figure 3.1). The first one was authored by one or more of Kim B. Bruce, Andrea P. Danyluk, and Thomas P. Murtagh. These publications discuss, for instance, (1) the suitability of EDP for an introductory programming course [66], (2) scaffolding the learning of EDP by program code libraries that simplify the usage of application programming interfaces used by more advanced programmers [65; 120], and (3) EDP as a facilitator of learning classes, parameters, and loops [64].

The second publication stream is involved with **Scratch** [291] and **LaPlaya**. The common author of the four publications is Diana M. Franklin. These publications discuss, for instance, curricula, state initialization, and possible issues that the usage of block-based languages might cause later computer science education. The third stream also has four publications. It is concerned with Scratch and has Deborah A. Fields as the common author. This stream investigates, for instance, programming profiles and learning trajectories, as well as EDP, parallelism, and initialization [79].

The fourth publication stream consists of five publications related to **App Inventor** [250] and linked together by authors Mark Sherman, Andrey Soares, and Franklyn Turbak. For instance, one of these publications [49] discuss App Inventor's event model.
Understanding Event-driven Programming

Bruce et al. 2005
Why Structural Recursion Should Be Taught before Arrays in CS 1
Murtagh 2007
Squint: Barely Visible Library Support for CS1
Bruce et al. 2010
Introducing Concurrency in CS 1
Bruce et al. 2001a
Event-Driven Programming Is Simple Enough for CS 1
Bruce et al. 2001b
Library to Support a Graphics-Based Object-First Approach to CS 1
Bruce 2004
Controversy on How to Teach CS 1: A Discussion on the SIGCSE-Members Mailing List
Bruce & Danyluk 2004
Event-Driven Programming Facilitates Learning Standard Programming Concepts
Franklin et al. 2013
Assessment of Computer Science Learning in a Scratch-Based Outreach Program
Franklin et al. 2016
Initialization in Scratch: Seeking Knowledge Transfer
Fields et al. 2014
Programming in the Wild: Trends in Youth Computational Participation in the Online Scratch Community
Fields et al. 2015
The Programmers’ Collective: Fostering Participatory Culture by Making Music Videos in a High School Scratch Coding Workshop
Fields et al. 2016
Combining Big Data and Thick Data Analyses for Understanding Youth Learning Trajectories in a Summer Coding Camp
Fields et al. 2017
Youth Computational Participation in the Wild: Understanding Experience and Equity in Participating and Programming in the Online Scratch Community
Harlow et al. 2018
Ecological Design-Based Research for Computer Science Education: Affordances and Effectivities for Elementary School Students
Weintrop et al. 2018
Starting from Scratch: Outcomes of Early Computer Science Learning Experiences and Implications for What Comes Next
Soares 2014
Reflections on Teaching App Inventor for Non-Beginner Programmers: Issues, Challenges and Opportunities
Turbak et al. 2014
Events-First Programming in App Inventor
Soares & Martin 2015
Teaching Non-Beginner Programmers with App Inventor: Survey Results and Implications
Kim & Turbak 2015
Adapting Higher-Order List Operators for Blocks Programming
Sherman & Martin 2015
The Assessment of Mobile Computational Thinking

Figure 3.1. Author-based publication streams identified in the mapping review (Publication I, Appendix B, Figure B4). The red color represents the first stream, which is mostly related to ObjectDraw and Squint. Programming in Scratch is also a major theme. The green stream is related to Scratch, and the purple stream is related to LaPlaya. The orange stream is related to App Inventor.
Publication Times. The included publications were published between 1993 and 2018 (inclusive). Figure 3.2 presents the annual publication counts without republications. As can be seen, the number of publications begun to increase at the turn of the millennium.

![Figure 3.2. Annual counts of first publications among all the publications included in the mapping review [Publication I, Figure 4]. The dotted line represents a three-year moving average.](image)

Publication Channels. The five most-used publication channels among all the included publications were ACM SIGCSE Technical Symposium (SIGCSE), Innovation and Technology in Computer Science Education conference (ITiCSE), Journal of Computing Sciences in Colleges (JCSC), Information Technology Education Conference (SIGITE), and ACM Transactions on Computing Education (TOCE).

RQ 1.2

**Which research methods have been used to study teaching and learning event-driven programming?**

The most important finding was that almost half of the publications included in the mapping review were descriptive—they did not contain clear data analysis or collection. This reflects the need for empirical evaluations of the learning outcomes and other suitability of teaching approaches and tools. Amongst the rest of the included publications, plain quantitative methods were most common, followed by mixed methods at the second place and plain qualitative ones at the third place. More of them can be found from § 3.4 of Publication I.

RQ 1.3

**In which educational contexts has event-driven programming been discussed?**
The mapping review presents the educational levels of the included publications using the International Standard Classification of Education (ISCED) levels [362]. All but one of the ISCED levels were represented (Figure 3.3), but the clear majority of the publications were related to Level 6, which means studies that aim for receiving a bachelor's degree.

![Figure 3.3. Educational levels of the publications included in the mapping review [Publication I, Figure 7], according to the ISCED levels [362]. Many publications address more than one level, and for ten publications (the N/A group) it was not possible to define a clear ISCED level.](image)

The most common course context on the university level was introductory programming (often “CS1”), and the most common themes were mobile, graphics, and graphical user interface (GUI) programming. More about educational contexts can be found from § 3.5 of Publication I.

### RQ 1.4

**What kind of pedagogical approaches have been reported to support learning event-driven programming?**

Whereas many publications presented *some* pedagogical approach, only a few publications focused purely on teaching and learning event-driven programming (EDP). As an example from the latter set, Hansen and Fossum [88] discuss a senior-level course that discusses various aspects of event-drivenness, including GUIs, concurrency, distributed systems, testing and debugging, hardware interrupts, simulation, and games.

Many publications contextualized (see Publication V) the teaching of EDP with, for instance, graphics [81], graphical user interfaces [72], web development [125], and sensor motes [52]. Other approaches discussed include computer game development [11], social multi-user virtual worlds [78], specification languages [146], and tangible devices [12]. In addition, several publications discussed orderings of learning content, such as *imperative-first*, *objects-first*, *design-patterns-first*, and *breadth-first*. For this dissertation, the most interesting discussion concerned introducing events early in the introductory programming course [e.g., 49; 64; 66; 67; 181].
More about teaching approaches, including orderings of learning content, can be found from § 3.6 and § 3.9 of Publication I.

**RQ 1.5**

What kind of software tools have been developed to support learning event-driven programming?

A visible line of work amongst the included publications was to develop program code libraries (see Table 3.1) for shielding novice programmers from possible difficulties and thus enabling them to create simple programs to achieve tasks that would otherwise be above their skills.

**Table 3.1.** Examples of scaffolding program code libraries that were mentioned in relation to event-driven programming in introductory programming courses.

<table>
<thead>
<tr>
<th>Library</th>
<th>Programming Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>BreezyGUI</td>
<td>Java</td>
</tr>
<tr>
<td>cs1graphics</td>
<td>Python</td>
</tr>
<tr>
<td>DoodlePad</td>
<td>Java</td>
</tr>
<tr>
<td>NGP</td>
<td>Java</td>
</tr>
<tr>
<td>ObjectDraw</td>
<td>Java</td>
</tr>
<tr>
<td>Squint</td>
<td>Java</td>
</tr>
<tr>
<td>Views</td>
<td>C# / .NET (primarily)</td>
</tr>
</tbody>
</table>

As a part of discussing tools, the mapping review also presented statistics on programming languages that the included publications were related to. In addition to textual written programming languages such as Java, C++, Python, and Visual Basic, also visual block-based languages/environments such as Scratch [79], LaPlaya [15], App Inventor [49], and Alice [5] were addressed. More about tools, frameworks, and programming languages can be found from § 3.7 of Publication I.

**RQ 1.6**

What empirical results of learning event-driven programming have been reported?

Unfortunately, only about a quarter of the analyzed publications presented results related to learning outcomes. Many of these studies are connected by the theme of visual block-based languages, and present small fragments of information. For instance, some students using Scratch were found to initialize their projects in wrong types of events [80], and when analyzing
the types of blocks students used, one group was found to use broadcasts—that is, sending events—more frequently than others [9]. Also, learning programming in a Scratch-style environment with parallelism might cause issues later in other environments. One such potential issue is the desire of having several actions to happen in parallel in an environment without parallelism [151]. A few publications discussed students’ learning with other programming languages, such as Java, C++, and Visual Basic [21; 24]. More about empirical results can be found from § 3.8 of Publication I.

Conclusions. Although event-driven programming (EDP) has become important in many application areas, empirical research on teaching and learning it to advise instructors in an evidence-based and practical manner seems not to have been reported much. For instance, the study in Publication I did not find empirical studies on what exactly did the students actually learn about EDP in a certain environment and how well, or studies that would indicate that one teaching method or tool is better than the other. Instead, it was usual (1) to discuss EDP in passing or on the side of some primary topic and (2) to take it for granted that students will learn EDP just by using some tool or environment. There are many directions available for future research; some of them are discussed in § 5.6.

3.3 Students’ Conceptions of Buttons (Publication II)

One common application of event-driven programming (EDP) are graphical user interfaces (GUI). To approach research on EDP from their perspective, Publication II describes a study that explores students’ thinking about the characteristics and functioning of an elementary user interface element: a button. In addition to learning about possible confusions in students’ thinking, the study was to tease out mismatches between teachers’ expectations of students’ knowledge and what the students actually know. The results of this study complement our knowledge regarding the second research question of this dissertation:

RQ 2

What kinds of misunderstandings do students have related to event-driven programming?
More specifically, the study is concerned with its first subquestion:

**RQ 2.1**

What kinds of conceptions do students have related to the workings of buttons in graphical user interfaces?

The study was performed on a course called *Internet and Browser Programming*. It was an introductory course for lifelong learners, offered in Finnish, and introduced its participants to Internet, Hypertext Markup Language, Cascading Style Sheets, and Dart. It was continuously open for new students, and the participants finished it at their own pace. As a part of the online learning material (in Finnish) in a bespoke learning management system, the students were presented a link to a voluntary survey with four substantive questions. The survey was opened in June, 2020, and students were not rewarded for participation. After it was decided to stop collecting data, screening the responses (see Appendix B) received so far resulted in 185 accepted answers from June 14, 2020 to May 21, 2021. These were then analyzed using thematic analysis.

**The Survey Questions.** The survey had four substantive questions:

- One about the concept of a button in general and the rest about the effects of a button in three scenarios.

**Question 1: The Essence of a Button**

How would you describe a thing that is commonly called as “a button” or “a synonym”?†

*Original in Finnish:*

Miten kuvailisit asiaa, josta käytetään nimeä “nappi” tai “painike”?†

† The English translation has the marking “a synonym”, because Finnish has several commonly-used terms for the concept of a button, but English seems not to have them: The meanings of knob, switch, and toggle, for instance, are different.

Not surprisingly, answering this question was not easy. The essence of a button was referred to in various ways (Figure 3.4). Majority of the answers used some vague term, such as *something, it, a thing, an object, an element, and a gadget*. The next frequent group of terms were related to interaction and user interfaces, and some students referred to a button in terms of its appearance. For the purpose of a button, most students stated that it causes something to happen. Other purposes included giving a possibility to choose, sending a command or some input, interacting, progressing forwards in a process, and confirming something. Other classes of charac-

---

† The course (FI: Internet ja selainohjelmointi), 2 ECTS in scope, belonged to a study module called *FITech 101: Digi & Data* (FI: Tietotekniikan perusteet) that was offered free of charge as online teaching by Aalto University.
Figure 3.4. Examples of terms that students used for referring to the essence of a button [Publication II, Figure 1]. The terms are translated from Finnish.

<table>
<thead>
<tr>
<th>VAGUE</th>
<th>INTERACTION</th>
<th>APPEARANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Something</td>
<td>Enabler of Interaction</td>
<td>Area</td>
</tr>
<tr>
<td>It</td>
<td>A Call for Action</td>
<td>Part</td>
</tr>
<tr>
<td>Thing</td>
<td>Functionality</td>
<td>Visual</td>
</tr>
<tr>
<td>Object</td>
<td>Function</td>
<td>Graphic</td>
</tr>
<tr>
<td>Element</td>
<td>User Interface</td>
<td>Shape</td>
</tr>
<tr>
<td>Gadget</td>
<td>Link</td>
<td>Icon</td>
</tr>
<tr>
<td></td>
<td>Choice / Selection</td>
<td>Box</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Switch</td>
<td></td>
</tr>
<tr>
<td>Trigger</td>
<td>Command</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Physical Object/Button</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A Button of a Keyboard</td>
<td></td>
</tr>
</tbody>
</table>

**Question 2: Shopping Mall Scenario 1**

You are in a shopping mall and see a screen that has a button on it. The button has a text “Touch to Start.” What do you think will happen when the button is pressed? Why?

*Original in Finnish:* Olet kauppakeskuksessa ja näet ruudun, jossa näkyy nappi. Napissa lukee teksti “Kosketa aloittaaksesi.” Mitä ajattelet, että napin painamisesta tapahtuu? Miksi?

To the second question, students most frequently stated the obvious assumption: something begins or starts. Other answering approaches included mentioning the process that the button affects as well as describing the user interface or the features that would become available after pressing the button.

**Question 3: A Form for Personal Information Scenario 2**

You are filling a personal information form on a web page. At the end of the form is a button that says “Save.” What do you think will happen when the button is pressed? Why?

*Original in Finnish:* Olet täyttämässä nettisivulla olevaa henkilötietolomaketta. Lomakkeen lopussa on nappi, jossa lukee “Tallenna.” Mitä ajattelet, että napin painamisesta tapahtuu? Miksi?

The answers to the third question varied in details about, for instance, saving and submitting information, the place of storing the information, and the process related to giving and processing the information. A few students speculated about the possibility of having a Submit button alongside or without the Save button.
There is a weather information application in your mobile phone. The application has a button with a text “Fetch Weather Data.” Pressing the button causes execution of the program code listed below. What do you think will happen when the button is pressed? Why?

```javascript
var button = getButtonByText("Fetch Weather Data");
button.setText("Fetching Data");
var fetch = await HttpRequest.get("https://weather-service.com/weather");
textField = getTextFieldById("weatherinfo");
textField.setText(fetch);
button.setText("Fetch Weather Data");
```

In the fourth question, most answers were given on higher abstraction levels than the given program code. When discussing the code, fetching references to user interface controls was rarely mentioned. Also, there were some misunderstandings concerning the functioning of the code.

Conclusions. This study indicated that the respondents had rough ideas about what buttons are and what their effects might be in the three scenarios. However, often the answers were partly ambiguous or incorrect. One of the most common reasons the students mentioned for their thinking in questions 2–4 was the caption of the button. Curiously, despite seeing the given program code in the fourth question, a few students still stated that they expected something on the basis of the button’s caption instead of telling, for instance, that the program code does this and that, or that they do not understand what the given program code does. One hypothesis is that they forgot or did not care to mention about falling back to the caption after not understanding the program code, but would this hold for all of these answers?

The answers often implied assumptions, such as that the caption sufficiently describes the button’s effects; that the button does not (mistakenly) trigger some other functionality; that the button actually causes something to happen in the first place; and that the effects of the button can actually be performed in the current situation. For some students, the button and the effects of pressing it seemed to be inseparable. Moreover, for some, the button was actively performing the effects of pressing it instead of just triggering the execution of some program code.

This observation was noted also in the study in Publication IV (§ 3.4) concerning another course, population, and program code.
Understanding Event-driven Programming

The other commonly-given reason for expecting something in questions 2–4 was experience: All the three situations are common in the contemporary society, and some students indeed mentioned having experienced something similar, such as a self-service point-of-sale system in a restaurant. Also, in the second question, the environment of the screen was also mentioned to have had an effect.

All in all, some respondents might not have enough fundamental knowledge and understanding regarding interactive applications. As some children start to use graphical user interfaces of mobile phones, gaming consoles, and other computers already in primary school age or even earlier, it is easy for an educator in the university to take this kind of knowledge for granted. However, if students have not been exposed enough for that kind of reasoning, it might be beneficial to increase the amount of exposure to fundamental principles of interactive applications and the components of graphical user interfaces.

3.4 Students’ Conceptions of Event-driven Programming (Publications III & IV)

When teaching a topic, it is crucial to ensure that the learner knows the proper terminology as well as understands the concepts and their relationships. Regarding event-driven programming (EDP), the students should understand the concepts introduced at the beginning of § 3.1.1. This section is related to two studies that are concerned with students’ understanding concepts related to EDP. The results of these studies complement our knowledge regarding the second top-level research question:

RQ 2

What kinds of misunderstandings do students have related to event-driven programming?

More specifically, the results are related to its second subquestion:

RQ 2.2

In which different ways do students understand the concepts of event-driven programming and runtime behavior of event-driven programs? What kinds of misunderstandings do they have on these concepts?

The first study (Publication III) is relevant in respect of the qualitative discussion of both challenges in answering questions in its § 5.1 and confusions about EDP in its § 5.2. It indicated the possible presence of two
results that were confirmed in the second study (Publication IV), which is directly concerned with understanding of both concepts and program code related to event-driven programming (EDP).

Both of the studies were carried out on a course called Introduction to Web Development and Programming.\textsuperscript{5} It was an introductory course for lifelong learners and introduced its participants to both client and server programming using Hypertext Markup Language, Cascading Style Sheets, JavaScript, and Node.js. Learning material of the course was offered online in a bespoke learning management system. The material had eight chapters [PIII, Figure 1], of which the fourth discussed elementary browser programming in JavaScript—especially EDP. The data for the first study (PIII) was collected during the second implementation of the course on fall 2020, and the data of the second study (PIV) is from the third implementation on spring 2021.

3.4.1 The Initial Clues: The First Publication (PIII)

The study described in Publication III was concerned with three exercises, in which students developed some program code and were then presented questions about that code. The relevant parts for this dissertation are the third exercise (E3 in PIII) and the qualitative analysis based on it. The exercise asked the students to create event handling code to a project template and then to answer three open-ended questions (E3.1–E3.3 in Table 1 of PIII). Analysis of the answers for confusions regarding event-driven programming (EDP) revealed the initial clues regarding two possible issues:

1. There might have been misunderstandings regarding the synonymous terms event listener and event handler (see § 5.1.2 in PIII).

2. Confusions seemed to exist regarding the meaning of adding an event handler for a Document Object Model element, the time when the event handlers are executed, and who calls them. Especially, some students explained that addEventListener() method would call the event handler it is given as an argument (see § 5.2.2 in PIII).

Both of these issues are linked to the observations made in the second study (Publication IV, § 3.4.2), which was dedicated to teasing out students’ misunderstandings regarding EDP.

3.4.2 Digging Deeper: The Second Publication (PIV)

After the study in Publication III revealed cues to the existence of the issues described above, the study described in Publication IV was performed

\textsuperscript{5} The course, 3 ECTS in scope, was offered free of charge as online teaching by Aalto University.
to explore students’ understanding of concepts related to event-driven programming (EDP). In this study, the students were presented two exercises:

1. The **concept comprehension exercise** asked the students to explain (1) what concepts are related to event-driven programming, (2) what do those concepts mean, (3) what kinds of relations do those concepts have with each other, and (4) what kinds of activities do they perform alone and together. The accepted answering methods were concept maps [e.g., 6; 8], text, or both.

2. The **code comprehension exercise** presented a short code for a web page in Hypertext Markup Language with JavaScript embedded in it (Listing 3.1), encouraged the students to try the page on their own computers, and then describe (1) what the JavaScript code does after a button on the web page is clicked and (2) what are the effects of such a click regarding the web page.

```html
<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.innerHTML = parseInt(c.innerText) + 1;
    d.addEventListener("click", a);
    document.getElementById("a").appendChild(d);
  }
  document.getElementById("b").addEventListener("click", a);
</script>

**Listing 3.1.** The JavaScript code to be explained in the code comprehension exercise (see § 3.4.2) of the study that is presented in Publication IV. The surrounding boilerplate HTML code is omitted.

Screening (see Appendix C) resulted in 24 concept comprehension exercise answers and 26 code comprehension exercise answers to be analyzed. The concept comprehension exercises were normalized into class and activity diagrams in Unified Modeling Language, and those reductions were then analyzed regarding relationships and process. The code comprehension exercise answers were analyzed for the presence of specific steps.

The concept comprehension exercise resulted in detailed information about the many ways students had understood the details of EDP. For instance, **Figure 3.5** presents students’ descriptions of the relationships between Document Object Model and the terms *event listener* and *event handler* (the latter two should be synonymous). In particular, it clearly demonstrated that many of the students had the incorrect idea that *event handler* and *event listener* refer to separate concepts, often so that a listener actively “listens to” events and calls a handler when needed. This confirms the first observation presented in § 3.4.1.
The code comprehension exercise provided details regarding students’ understanding and possible difficulties. For instance, majority of the students omitted the first step of fetching a reference to the clicked button. Also, a few students claimed that `addEventListener()` would wait for mouse clicks, which is linked to the second observation in § 3.4.1.

### 3.4.3 Conclusions

The results of the second study (Publication IV) demonstrated that although the concept of an event in itself is well understood in general, the difference between the occurrence and its representation in computer as well as other concepts and their relationships should be made clear to the students. For implications to educators, please see § 5.4.
4. Supporting Teaching

The previous chapter discussed students’ thoughts and understanding regarding event-driven programming (EDP). This chapter approaches EDP from a different angle: the approaches and tools available for teachers. First, it answers RQ 3 by discussing contextualized teaching of EDP (§ 4.1) on the basis of Publication V, which discusses approaches to classify tools of contextualized programming education as well as tools and exercise types of media computation. Second, it answers RQ 4 on the basis of Publication VI (§ 4.2), which describes construction and small piloting of a general-purpose visualization tool that has been used to teach EDP and client–server communication.

4.1 Contextualization (Publication V)

One method for increasing students’ motivation on introductory computing courses is to contextualize the teaching. In other words, the concepts to be learned can be presented in a context that is naturally meaningful for the students and thus provides a source for intrinsic motivation towards studying. By discussing options for contextualization of teaching event-driven programming (EDP) on the basis of Publication V, this section answers the third top-level research question of this dissertation:

**RQ 3**

*In which different ways has teaching of event-driven programming (EDP) been contextualized to increase students’ motivation? What types of tools have been developed for media computation and how can media computation be used for contextualizing EDP?*

This research question is answered on the basis of three classifications that are presented in Publication V: Tools for contextualized programming education (§ 4.1.1), tools for media computation (§ 4.1.2), and types of media
computation exercises (§ 4.1.3). Naturally, similar classifications could be constructed in various ways [e.g., 19; 115]; the approaches taken here were intuitive for the author while initially writing his master’s thesis [162]. Because they are practical and can be internalized quickly, they could be useful for someone familiarizing themselves with the spectrum of tools and exercises for the first time.

The classifications are not intended to be fully complete in any way, but they form a basis for more extensive work. They were constructed on the basis of a literature review and recommendations from colleagues. For simplicity, they are hierarchical in a single dimension, but can be extended as necessary when new information is needed or becomes available. If, for instance, learning goals, theories, difficulty levels, targeted education levels, or the effort needed for adopting a specific approach become interesting, they can be added later, perhaps as new dimensions. Finally, the classes are not mutually exclusive in general: A single item can often simultaneously belong to several classes.

4.1.1 Contextualization Tools in General (Classification I)

One way to structure the components of contextualized programming education is to divide them into three classes (Publication V, Figure 1): themes (1), practices (2), and tools (3). A theme is the subject matter or field that usually is the primary motivating factor for students to learn. These include everyday experiences, real-world data, and school subjects, as well as games [37] and multimedia [70; 85; 86; 142]. Tools [e.g., 160] are technologies that students use in the learning process; the first two classifications are concerned with them. Finally, practices are students’ and teachers’ activities that promote a specific contextualization approach. Such practices include problem-based learning, sharing creative results, and collaboration with industry partners.

The contextualization tool classification (Figure 4.1) is hierarchical and based on practical aspects of using the tools. On the first level, the tools are divided on the basis of whether they are essentially a combination of software and hardware (2.1), essentially software (2.2), or essentially hardware (2.3).

2.1 Solutions Essentially Combining Software and Hardware.

These hybrid tools are divided into three classes: The first class (2.1.1) contains hardware that is actively driven by a general-purpose computer (e.g., via USB [70; 104], MIDI [91; 168], and DMX [167] interfaces), whereas hardware in the second class\(^1\) (2.1.2) run inde-
**Figure 4.1.** Branch tools (2) of Classification I (§ 4.1.1): Tools for contextualized programming education [Publication V, Figure 2].
Supporting Teaching

independently after being programmed with a general-purpose computer. Some devices, such as robots [74; 153], could be placed in either one of these classes, depending on how they are used. The third class (2.1.3) is for other hybrid solutions, such as Puzzlets [285], in which an external controller is used to send command sequences to applications running on a general-purpose computer.

The above three classes are further divided on the basis of whether the hardware is internal or external to the general-purpose computer.

2.2 Essentially Software-based Solutions.

Software tools are divided based on the theme of the tool, of which Table 4.1 presents some examples. Graphical user interfaces (2.2.8) are further divided to essentially course-specific templates (2.2.8.1) and third-party tools (2.2.8.2).

2.3 Essentially Hardware-based Solutions.

Hardware tools, including many tangible tools [e.g., 158], are further divided on the basis of their need for electricity (2.3.1) or the lack of it (2.3.2). Electrical devices (2.3.1) are divided to embedded systems (2.3.1.1), such as Cubetto [262], and other devices (2.3.1.2). Non-electrical tools include board games, such as c-jump [254], Code Monkey Island [255], Robogem [288], and Robot Turtles [289].

The above classification presents many possibilities to contextualize teaching of event-driven programming (EDP). Development environments of both embedded systems [e.g., 123] and general-purpose computers often enable the programmer to react to some kinds of sensor inputs from simple switches and keypads to rarer sensor types, such as distance and temperature sensors. Some operating systems send applications events to inform them about phases of their lifecycle and changes in their operating environment (such as rotating the screen and changing the color theme). Furthermore, interfacing other devices enable reacting to arriving data. Naturally, applications can define their own events as appropriate.

EDP can be easily included in many contextualization themes. User interfaces and games could process events corresponding users’ input, and the data from the sensors of robots and measurement devices could be processed as events. Simulations, microworlds, games, and story-telling programs could offer events related to the occurrences inside the simulation/world: For instance, the pressure exceeding the safety level, a water tank becoming empty, colliding with or approaching an enemy or an obstacle, picking up an object, triggering a special combination of blocks, using all the energy, the darkness falling in the evening, and some given time expiring. Moreover, event streams can be processed and refined to new and small-scale computers, such as Arduino [3; 61; 62; 96; 97; 124; 251], BBC micro:bit [139; 253], and Raspberry Pi [154; 286].

54
Table 4.1. Examples of essentially software-based tools for contextualized programming education by class and theme in accordance with Classification I. For media computation, please see Classification II (§ 4.1.2) below.

<table>
<thead>
<tr>
<th>Class</th>
<th>Theme</th>
<th>Tool Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.1</td>
<td>Microworlds [94]</td>
<td>Guido van Robot [270], Jeroo [273], Karel the Robot [57], Logo [28; 161; 242; 294].</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Story-telling</td>
<td>Alice [5; 101; 153; 249], Etoys [267], Looking Glass [102; 279], Scratch [69; 291], ScratchJr [292], Snap! [293], ToonTalk [296].</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Games</td>
<td>AgentCubes [128; 248], Baltie [252], CodeAdventure [56; 256], CodeCombat [257], Codemancer [258], CodeMonkey [259], Code Spells [260], Construct [261], GameMaker Studio [268], Greenfoot [269], Hour of Code Activities [272], Kodu Game Lab [275], Lightbot [278], Nancy Drew: Codes &amp; Clues [283], Stencyl [295].</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Robots</td>
<td>Robocode [16; 24; 287], Robot Virtual Worlds [290].</td>
</tr>
<tr>
<td>2.2.8.2</td>
<td>Third-party GUI Libraries</td>
<td>ACM Java Task Force’s GUI library [29; 131], Java Power Tools [126], Portable Graphics Library [132].</td>
</tr>
</tbody>
</table>

See also: Table 3.1.

events, such as key presses into characters or commands and too many signals from some instrumentation in a specified time span into alarms.

Contextualization themes that have been presented in the literature in relation to event-driven programming (EDP) include embedded systems [20; 90], games [24; 30], graphical user interfaces [22; 55; 95], graphics [39; 65; 81], mobile devices [49; 107; 130], music [20], networks [20; 120], robots [54; 82; 121], web pages [125], and virtual reality [78]. Furthermore, EDP has been used as a context for introducing concurrency [67].

4.1.2 Tools for Media Computation (Classification II)

In the first classification regarding contextualization tools (§ 4.1.1 above), Class 2.2.3 represents a software-based contextualization approach called media computation [e.g., 10; 85; 87; 129; 142] that is based on professional concepts such as creating and manipulating images, videos, sound, and web pages programmatically. It has been suggested to both increase student retention and to make studying more interesting [e.g., 86]. The second classification (Figure 4.2) organizes software tools suitable for supporting media computation. The first level of the classification divides these tools to program code libraries and integrated development environments (IDE).
Figure 4.2. Types of tools for media computation [Publication V, Figure 3].

1 **Program Code Libraries.**

Libraries that contain executable program code are further divided on the basis of their approach to inputting and outputting media in relation to the programming environment, in which they are being used: Some (e.g., the Guzdial–Ericson multimedia class library [271]) perform input and output of the processed media independently (1.1), whereas others (e.g., DrRacket’s 2htdp/image [265]) take advantage of facilities provided by IDEs (1.2).

2 **Integrated Development Environments (IDE).**

IDEs are divided on the basis of their support for input and output (I/O) of media. Many IDEs, such as Atom, DrJava [263], Eclipse, GNU Emacs, Jython Environment for Students [274], MS-DOS Edit, Notepad++, Processing Development Environment, vi, Visual Studio, and Windows Notepad, are focused on editing text and do not participate in media I/O without being extended.

IDEs that do support media I/O are further divided on the basis of the relationship between media and program code. Some IDEs, such as Maplesoft Maple [280] and MathWorks MATLAB [282] with its Live Editor, allow media and program code to alternate in a window as input and output (2.1.1). More advanced IDEs, such as DrRacket [264] and Wolfram Mathematica [281], allow mixing media with the program code (2.1.2). Finally, the lowest level of support (2.1.3) is to

---

2 Here, the concept of a window is loosely defined as a rectangular area on the screen. However, it can be considered from various points of view and there are variants and related concepts, such as application windows, parent and child windows, dockable windows, frames, panes, and other user interface components—both self-managed (lightweight) and system-managed (heavyweight). This dissertation refrains from defining a window exactly and exhaustively.
display media and program code in separate windows. IDEs in this
class include EarSketch [266], Kojo [276], Pixly [284], Snap! [293],
Tunely [297], and the traditional MATLAB.

This classification provides a perspective to teaching event-driven pro-
gramming (EDP) using both program code libraries and IDEs. Considering
receiving events, the best approaches might be those, in which the pro-
gram being developed uses either full screen or its own application window
(Classes 1.1 & 2.2). This is because the IDE is not intervening in the event
flow that the program receives. On the other hand, when an IDE is a part
of the event handling of the program, it might restrict and interfere with
the event flow. This might happen when the IDE participates in inputting
and outputting media or when the program is run in the context of the
IDE, for instance, as a plugin (Classes 1.2 & 2.1).

An additional inconvenience is that often development tools that operate
as worksheets\textsuperscript{3} or read-evaluate-print loops (REPL) are not designed to
support EDP inside them, and REPLs are not designed for developing
large applications in the first place. However, both styles might still be
extended to support running programs that require continuous input.
These approaches include using external windows, full screen, or applet-
style embedded areas. If the usage of external windows or a full screen
mode is acceptable, a library that presents media in such ways might help
to evade the need for support in the development tool itself.

\subsection*{Exercise Types of Media Computation (Classification III)}

The second classification (\S 4.1.2) organized tools of media computation.
The third and last classification (Table 4.2) explores the types of media
computation exercises. It is based on identifying common themes in the
ways media are being programmatically used in the exercises of four
related textbooks [185; 191; 192; 193].

The first level of the classification divides the exercises on the basis of
whether the information handled in the exercise is created or transformed,
or if only its representation is changed. Creating information (1) includes
turtle graphics, sound synthesis, composing music, and creating dynamic
web pages. Transforming information (2) is further divided further divided
on the basis of whether the information is filtered (2.2) or composed (2.3).
The lower levels of these two classes contain examples of exercise types
for various kinds of information. Finally, transforming representation (3)
includes examples, in which the information remains essentially the same,
but its representation is changed.

The exercise types in this classification might seem not to contain or
need event-driven programming as such. However, many of them offer

\textsuperscript{3} For instance, a Scala worksheet [366].
Table 4.2. Types of media computation exercises in four introductory programming and data structure textbooks [185; 191; 192; 193] that apply creating and transforming media as a motivation for programming [Publication V, Table 1].

<table>
<thead>
<tr>
<th>1. Creating (Original) Information</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1. Turtle Graphics</td>
<td></td>
</tr>
<tr>
<td>1.2. Sample-based Sound Synthesis: Additive generation and mixing of sine, square, and triangle waveforms</td>
<td></td>
</tr>
<tr>
<td>1.3. Note-based Music Composing: MIDI music</td>
<td></td>
</tr>
<tr>
<td>1.4. Web Page Creation: Dynamically based on database content</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2. Transforming Information</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1. Filtering Information</td>
<td></td>
</tr>
<tr>
<td>2.1.1. Pixel-based Bitmap Filtering; Uniform/Gradual, Entirely/Partly</td>
<td></td>
</tr>
<tr>
<td>2.1.1.1. Color Adjustments: Separation and exchange of color channels, color replacements and gradients, brightness adjustments, negation, black-and-white and sepia conversions, red-eye reduction, and posterization</td>
<td></td>
</tr>
<tr>
<td>2.1.1.2. Geometrical Transformations: Rotating, scaling, flipping, and mirroring</td>
<td></td>
</tr>
<tr>
<td>2.1.1.3. Convolution-based Effects: Blurring and sharpening</td>
<td></td>
</tr>
<tr>
<td>2.1.1.4. Miscellaneous Effects: Oil-paint</td>
<td></td>
</tr>
<tr>
<td>2.1.1.5. Edge Detection</td>
<td></td>
</tr>
<tr>
<td>2.1.1.6. Steganography: Hiding information into bitmaps</td>
<td></td>
</tr>
<tr>
<td>2.1.2. Sample-based Sound Waveform Filtering</td>
<td></td>
</tr>
<tr>
<td>2.1.2.1. Volume Adjustments: Normalization, fade-ins and fade-outs, as well as distortion by inducing clipping</td>
<td></td>
</tr>
<tr>
<td>2.1.2.2. Pitch Adjustments</td>
<td></td>
</tr>
<tr>
<td>2.1.2.3. Structural Adjustments: Reversing, mirroring, and removing unnecessary parts</td>
<td></td>
</tr>
<tr>
<td>2.1.3. Frame-based Video Filtering, including time-parameterized bitmap filters</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2.2. Compositing Information</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.1. Primitive Shapes — More Complex Shapes</td>
<td></td>
</tr>
<tr>
<td>2.2.2. Bitmaps — Visual Collages; possibly with transparency and chroma-keying</td>
<td></td>
</tr>
<tr>
<td>2.2.3. Dynamically Changing Bitmap Collages — Videos</td>
<td></td>
</tr>
<tr>
<td>2.2.4. Sound Waveforms — Audio Collages</td>
<td></td>
</tr>
<tr>
<td>2.2.5. MIDI Clips — Audio Collages</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3. Transforming Representation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1. Writing image’s color components as values into a text file, modifying them in a spreadsheet application, and reading them back into an image</td>
<td></td>
</tr>
<tr>
<td>3.2. Writing samples of a sound waveform into a text file and representing those values as a line chart in a spreadsheet application</td>
<td></td>
</tr>
<tr>
<td>3.3. Interpreting sound waveform’s sample values as colors to create a bitmap representation of the sounds</td>
<td></td>
</tr>
<tr>
<td>3.4. Retrieving a web page in source code (HTML) form and parsing some data out of it</td>
<td></td>
</tr>
</tbody>
</table>
possibilities for including event-driven programming. Both creating and transforming information might happen on the basis of (real-time) input from sources such as keyboard, mouse, joystick, other local sensors, files, applications, networks, and user interface controls. Application programming interfaces for reading such inputs might be event-based. Especially time-related media, such as video [e.g., 70] and sound, could offer interesting contexts for controlling parameters of real-time syntheses and transformations.

4.1.4 Conclusions

The three classifications presented above offer a general view to the tools of contextualized programming education and two more detailed views to media computation. They make it easier to familiarize oneself with the spectrum and characteristics of the options available and to relate single options to the whole. Furthermore, they serve as tools for identifying suitable contexts for teaching event-driven programming.

4.2 CSMV for Visualizing High-level Concepts (Publication VI)

In teaching materials, it might be helpful for the learners if abstract concepts were visualized. Specialized tools (see § 1.4 and § 2.3) help in their use cases, but they are not suitable for visualizing arbitrary topics. This leads to the fourth top-level research question:

RQ 4

What kind of a tool is suitable for visualizing high-level concepts of event-drivenness in online learning materials in such a way that

1. for a computing educator, the learning curve of the tool is relatively low
2. the tool and the visualizations play well together with version control systems
3. the visualizations can be easily shared with colleagues
4. students' activities with the tool can be logged for research purposes
5. the source code of the tool can be customized by educators, and
6. the tool suits for environments, in which expenses and commercial software are not acceptable by principle?
This section answers the above research question by discussing a visualization tool prototype, CSMV\(^4\) [106] (an acronym for Client–server Messaging Visualizer), based on Publication VI. The prototype was constructed for embedding visualizations of arbitrary high-level concepts into online learning materials in learning management systems. It possesses the following characteristics:

- Visualizations are composed of frames called *steps* and can be browsed through step by step at the student's own pace. Each step can contain text, Hypertext Markup Language (HTML) code, and images. A visualization is created by (1) preparing the necessary images, (2) writing a textual *script* that describes the visualization, (3) referring to the necessary files from the HTML file of the parent web page, and (4) adding a placeholder element to the same HTML file.
- Learning curve for creating visualizations is relatively low: Basics of HTML, JavaScript, and Cascading Style Sheets are often enough to create a script and to embed the visualization to a web page.
- The textuality of the visualization scripts supports versioning using version control systems. Thus, the tool is well-suited to environments that automatically fetch the learning material from repositories.
- The relatively small size of the visualizations makes them easy to share between colleagues.
- The tool generates events of users’ activities for logging and research purposes. These events can be captured and sent to, for instance, a learning management system for storage and further processing.
- Dependencies to external (commercial) services only for delivering the visualizations are not needed. If so desired, even the only runtime dependency, JavaScript library called jQuery [344], can be delivered as a part of the learning material instead of downloading it from an external server, such as those of some content delivery network.
- As being open-source, the tool can be augmented and customized for varying usage scenarios. It is also suitable to environments that prefer or require using open-source software.
- The tool is free to use, so it suits for environments where expenses are not acceptable or commercial software is forbidden by principle.

Figure 4.3 presents the steps of a short visualization created using CSMV. As can be seen, the user interface consists of a title, a description, a visualization area, and the controls for browsing the visualizations. Various aspects of the user interface, such as the size of the visualization and the visible texts, can be configured in the script. Also, Cascading Style Sheets can be used to customize the appearance of both the user interface and the content of the visualization.

\(^4\) On Jan. 4, 2022 and for the time being, CSMV is available at https://github.com/aleksi-lukkarinen/Client-Server-Messaging-Visualizer.
CSMV was developed using EcmaScript 6 [166], and Babel [301] is used to transpile it into EcmaScript 5 for compatibility with Internet browsers. As mentioned earlier, it uses a JavaScript library called jQuery [344], and its build is automatized with Gulp [320].

Visualizations are described for CSMV via a JavaScript file\(^5\) (Listing 4.1) that is to be included to the web page containing the visualizations with CSMV and jQuery by using the `<script>` tag. The file consists of a single variable that contains a list of visualization scripts.

Each visualization script consists of a name (a unique identifier), a title, and a description as well as four sections: debug and environmental settings, definitions of actors that are used to compose the steps, and definitions of the steps. Each of the steps is defined by setting the positions and visibilities of the actors defined in the script. Internally, CSMV operates essentially as a state machine with a linear list of states that correspond the steps of the visualization.

\(^5\) An example of a full configuration file can be seen in CSMV’s source repository.
Listing 4.1. A skeletal configuration file of CSMV (a shorter version is presented in Publication VI, Listing 1) that defines a single actor and a single step only. More are to be added to describe the visualization.
In addition to its original purpose of visualizing protocols, CSMV can be used for visualizing other high-level concepts, such as those related to event-driven programming (EDP). As an example, Figure 4.4 presents a step of a visualization that illustrates adding an event handler for a button on a web page. Other potential EDP-related visualizations include an event vs. a record of an event, routing of event records in the computing equipment, execution model of event-driven programs, essence of the Observer design pattern, and storing state outside the event handlers.

Figure 4.4. A step [presented in 106] from a CSMV-based visualization that illustrates the process of adding an event handler for a button on a web page (used on the course described in § 3.3). The visualization gradually adds blocks of code (with a highlighted background) and describes each addition in the red rectangle at the top. The program code is pseudocode, and the explanations, the identifiers in the program code, as well as the user interface are in Finnish:

- Tapahtumakäsittelijän rekisteröiminen [Registering an Event Handler]
- Lisätään koodilohkoon tapahtumakäsittelijäksi funktio, joka arpoo luvun ja sen perusteella esittää taulukossa olevan vitsin.

```html
<html>
<head><-- Ohitetaan tässä esimerkissä --></head>
<body>
<input type="button" id="painikeKerroVitsi" value="Kerro vitsi">
<script>
vitsit = taulukko;
  "Mistä äänekäs kissa tunnetaan? Mausta.",
  "Millaista on työttömän muusikon elämä? MÖllivoittoista.",
  "Mistä tietää kahvinkeitittäjän epäonnistuneen? Mokasta."

funktio naytaVitsi = {
  i = satunnaislukuValilta(0, vitsit.pituus - 1)
  ilmoita/vitsit[i])
}
</script>
</body>
</html>
```
CSMV was evaluated in a small pilot study during spring 2020 on the first implementation of the course described in § 3.4. For the pilot study, the learning material of the course was augmented with three CSMV-based visualizations, and button presses from the visualizations were logged. One of the visualizations was similar to the one in Figure 4.3. This pilot study answers the subquestion of RQ 4:

RQ 4.1

*How does such a visualization tool work in its intended environment, and what are students’ perceptions of such a tool?*

From the technical point of view, the data logging worked while CSMV was a part of the course material, and technical problems that could originate in CSMV were not experienced. These facts support the expectation that the CSMV implementation functions well in its intended environment at least to the extent of the used visualizations (that act as test data).

On the basis of analyzing the log data, the students seemed to consider the individual steps of the visualizations, and some students moved back and forth in the visualizations instead of quickly “skimming through” all the steps once. Student feedback was collected as a part of an end-of-course survey via two questions. The first question was answered by using a Likert scale, and the second one requested open-ended textual feedback (see Publication VI, Section 4). To the first question, only one of 43 respondents gave a negative assessment, whereas 35 answers were positive on some level (seven were neutral). The second question gathered only 12 answers, of which seven were neutral and four positive.

As a conclusion, the log data and the student feedback suggest that the concept of visualizing abstract concepts is worth developing further. Furthermore, the functioning of the prototype suggests that CSMV as an implementation offers a good baseline for further development.
5. Discussion

The two previous chapters discussed new research on teaching and learning event-driven programming (EDP) on the basis of the six publications that belong to this dissertation. This chapter starts by briefly listing the dissertation's contributions (§ 5.1). They are followed by the primary outcomes of this dissertation by research question (§ 5.2), the implications of this dissertation for computing educators (§ 5.4), and some general observations regarding EDP (§ 5.3). The next section discusses the validity and the limitations of this work (§ 5.5), and this chapter ends with suggestions regarding possible directions for future work (§ 5.6).

5.1 Contributions

Regarding event-driven programming (EDP), this dissertation makes the following contributions to the field of Computing Education Research:

1. A review of the past research on teaching and learning EDP (PI).
2. Discussion about the difficulty of defining EDP (§ 3.1.1).
3. Students perceptions, including misunderstandings, about
   • buttons and their applications (PII)
   • concepts of EDP (PIII and PIV).
4. Approaches for activating students in learning EDP through
   • contextualized teaching, including media computation (PV)
   • interactive visualizations of high-level concepts (PVI).
5. Practical suggestions for computing educators to improve the practice of teaching EDP (§ 5.4).
6. Opportunities for future research (§ 5.6).

The primary outcomes as well as the implications for computing educators and researchers are presented in § 5.2, § 5.4, and § 5.6.
5.2 Primary Outcomes

The research questions (RQ) posed in § 1.4 were answered in Chapter 3 and Chapter 4. This section recapitulates that treatise by discussing the primary outcomes of individual studies.

The answer to the first research question forms a basis for further research by clarifying the current state of Computing Education Research regarding teaching and learning event-driven programming (EDP) in the published literature:

RQ 1

What kinds of journal articles and conference papers are available on teaching and learning event-driven programming?

This research question was answered in § 3.2 on the basis of Publication I, which describes a mapping review regarding teaching and learning EDP. It included 105 publications, of which 104 were identified as relevant from a total of 858 search results; the one additional publication was added later based on a suggestion. Several kinds of data were collected from those 105 publications, including bibliometrics, research methods, educational contexts, as well as their pedagogical offerings: approaches, tools, and/or outcomes (please see § 3.2 as well as Publication I, including its appendices).

The most conspicuous observation is that even though EDP has an important role in the computing industry, the research community has not published much empirical research that would focus on some aspect of teaching and learning EDP: In fact, almost a half of the included publications did not present a clear data collection or analysis. Several publications present personal opinions and only anecdotal evidence, but generalizability of these statements to other contexts can be questioned. Furthermore, several publications treat learning EDP as an automatic result of, for instance, using a specific tool, instead of experimentally evaluating the extent and qualities of the supposed learning. This leaves many paths for future research, some of which are discussed in § 5.6.

The second research question was intended to increase understanding of students’ difficulties regarding EDP:
This dissertation addresses the above research question from two viewpoints. The first of them is graphical user interfaces and their controls—an important and often ubiquitous application area of event-driven programming (EDP). It is represented by the following subquestion that addresses a single but fundamental user interface control:

**RQ 2.1**

*What kinds of conceptions do students have related to the workings of buttons in graphical user interfaces?*

This research question was answered in § 3.3 on the basis of Publication II. Some students considered buttons as actors that actively perform their effects instead of starting them, and some expected that instead of the program code given in the exercise, the label of a button describes what the button does. The reasons for these two observations are not completely clear and require further studying. In general, it was found that the students might not have enough exposure to fundamental knowledge regarding the workings of graphical user interfaces and interactive applications. Consequently, it is possible that teachers expect the students to have a deep understanding of more topics than they actually do.

The another viewpoint to students’ difficulties is their conceptual understanding regarding EDP, represented by another subquestion:

**RQ 2.2**

*In which different ways do students understand the concepts of event-driven programming and runtime behavior of event-driven programs? What kinds of misunderstandings do they have on these concepts?*

This question was answered in § 3.4 on the basis of the studies in Publication III and Publication IV. In them, it was found that despite the students had a basic understanding about the concept of an event, they had misunderstandings regarding adding event handlers as well as dispatching events, including the meanings of the terms *event handler* and *event listener*. 
Discussion

The third research question is the first of the two questions that target tools for teaching event-driven programming (EDP). More specifically, it seeks tools that are suited for contextualizing the teaching of EDP to make students feel that studying it is more meaningful than when its concepts are presented either without context or in a context that is not personally or professionally meaningful for the students.

RQ 3

In which different ways has teaching of event-driven programming (EDP) been contextualized to increase students’ motivation? What types of tools have been developed for media computation and how can media computation be used for contextualizing EDP?

This question was answered in § 4.1, which presented various possibilities to contextualize the teaching of EDP on the basis of three classifications from Publication V. The first classification is concerned with tools of contextualized programming education in general, the second one deals with teaching tools related to media computation, and the third one explores various exercise types of media computation.

The fourth (and final) research question also addresses tools for teachers. It seeks to help teachers to convey high-level concepts of EDP to students in a more effective way:

RQ 4

What kind of a tool is suitable for visualizing high-level concepts of event-drivenness in online learning materials in such a way that

1. for a computing educator, the learning curve of the tool is relatively low
2. the tool and the visualizations play well together with version control systems
3. the visualizations can be easily shared with colleagues
4. students’ activities with the tool can be logged for research purposes
5. the source code of the tool can be customized by educators, and
6. the tool suits for environments, in which expenses and commercial software are not acceptable by principle?

As an answer to the above question, § 4.2 presents a prototype for a new visualization tool on the basis of Publication VI. The tool enables and em-
powers teachers to visualize high-level concepts so that the visualizations can be integrated into many online learning materials. The visualization tool has a relatively low learning curve for a computing educator, it is well-suited for version control systems, the visualizations can be shared with colleagues easily, and the tool supports logging for learning analytics purposes. Furthermore, the source code of the tool is open for inspection and customization, and the tool is free to use.

5.3 General Observations

The following subsections discuss the difficulty of defining event-driven programming (EDP), the lack of experimental research on teaching and learning EDP, notional machines for EDP, as well as positioning of EDP in the computer science curricula.

5.3.1 The Difficulty of Defining Event-driven Programming

As discussed earlier, event-driven programming (EDP) is important on many fields of computing. Especially many kinds of software, such as graphical user interfaces, games, services, and control programs of embedded systems often rely heavily on reacting to events. However, giving EDP an exhaustive and overarching definition seems not to be easy (§ 3.1.1). The root cause for this might be the ambiguousness of the term EDP itself.

A term should be chosen based on the exact meanings that its words convey. Otherwise, the meaning of the term will be unnecessarily unclear and the usage of such a term will hinder the communication. Thus, using such terms would be unwise. To define the term EDP clearly, exactly, and exhaustively, it has to be considered against, for instance, English dictionaries, the developer’s common sense, and any imaginable technology.\footnote{A really insightful and overarching definition would not be constrained even to technologies. Instead, it would give a philosophical view that is applicable to the whole known universe and conveys a broad understanding of applying the defined idea to anything there exists.}

When considering the “real” meanings of the three words in the term EDP at the broadest, the term becomes ambiguous and can arguably refer to many sets of criteria, some of which might include program’s runtime behavior, specific features of program code, and specific programming environments and their features. In addition, the term event can refer to practically any kind of specified occurrence, and programming does not have to mean the usage of traditional text-based programming languages, either. Such ambiguity enables the users of the terms to use them as they desire in their own contexts, which serves to muddle the meaning that should be conveyed.

1 A really insightful and overarching definition would not be constrained even to technologies. Instead, it would give a philosophical view that is applicable to the whole known universe and conveys a broad understanding of applying the defined idea to anything there exists.
A part of the ambiguousness around the term event-driven programming (EDP) is that EDP has been called a programming paradigm in the same sense as, for instance, procedural, functional, and object-oriented programming. However, Krishnamurthi and Fisler [159] remind that there is no clear definition for a programming paradigm, which makes it impossible to verify EDP’s status in this respect by comparing it to some commonly-accepted criteria. Furthermore, as Krishnamurthi and Fisler note, EDP seems to characterize the program’s behavior rather than the organization of the program code. In other words, event-driven behavior can be implemented in various program code organizations, including procedural, functional, and object-oriented styles.

If there were different kinds of programming paradigms, then perhaps EDP could be a behavioral paradigm instead of an organizational one. However, given the ambiguousness around EDP, it would be better to replace the term EDP with terminology that directly and clearly describes the exact characteristics that are being referred to, such as reacting to runtime input, a specific organization of program code, the existence of an event loop, writing interrupt handlers, or the visibility of an event record to the programmer. Unfortunately, after being used and taught decades around the world, the terminology is not going to change overnight. A humbler request than asking for an immediate change is that everyone would consider the terminology more closely and intentionally choose to use terms that describe the intention of the communicator accurately. Everyone can influence on their own part—even if it were against the established practice—and perhaps the result can be seen in time as better practices being established.

5.3.2 Experimental Research on Event-driven Programming

The mapping review described in Publication I demonstrated that the amount of experimental research that focuses on teaching and learning event-driven programming (EDP) is small. Given both the importance of EDP as well as the time the term has been used in the academia and the industry, one would expect that teaching and learning EDP would have been extensively researched already. Considering generalizable results based on experiments, however, this is not the case.

The reason for the above research gap is unclear. The existing literature contains publications on tools and approaches for teaching EDP as well as some discussion on positioning EDP in the computer science (CS) curriculum. EDP has also been included to the model curricula since 2001 [173; 174; 175; 176]. These facts suggest that the reason cannot be the unawareness of CS educators or the related research community.

Unfortunately, arranging comprehensive experimental studies might not be feasible in all educational settings due to, for instance, practical or
Discussion

ethical reasons. In addition, teachers are often busy with teaching and developing their courses, whereas carrying out extensive studies requires extra effort on the top of their other work. Some practical aspects of research could be given to students on doctoral and master’s levels, but the amount of workforce is still limited.

One natural factor contributing to the lack of experimental studies might be the competition between research areas and foci in the field of computing. To start with, EDP is only a part of development tools and programming languages, such as any other feature or programming style. Consequently, focusing on EDP is just one possibility of a multitude of perspectives, instead of studying teaching and learning of, for instance, the application of higher-order functions \( \text{fold}(\), \( \text{reduce}(\), and such. Also, although event-driven programming is a fundamental application and style of programming that is constantly present in many flavors of computer applications, it has no pivotal high-profile influence in the industry or the society. More generally, studying teaching and learning introductory programming is basic research that competes with more popular and visible research areas.

5.3.3 Notional Machines for Event-driven Programming

Although notional machines are not directly related to this dissertation, the results of Publication IV suggest that students might have difficulties in adapting their existing notional machines for event-driven programming (EDP). Compared to notional machines that support programs, in which all subprograms are called from within the same program code, the mechanics of notional machines for EDP should support at least the following situations (see also § 5.4):

1. An event handler is registered or unregistered with the programming environment.
2. Information about an event arrives from outside the program and causes an event handler to be called.
3. Execution of the thread waiting for events is somewhere else (e.g., in a hidden main program or event loop) than on a line of the program code written by the student themselves.
4. There is neither a main program nor the corresponding entry point in the student’s program code (e.g., there are only event handlers that are registered outside the program code).

Apart from the discussion of Čuvić et al. [76], there seems not to be much coverage of notional machines for EDP in the published literature. Thus, more research is needed on this area (see also § 5.6).
5.3.4 When to Teach Event-driven Programming

One fundamental question regarding teaching event-driven programming (EDP) is its positioning in the computing curriculum. It has been taught on dedicated courses [e.g., 88], but some approaches [e.g., 13; 22; 39; 49; 55; 66] have integrated it into the introductory programming course.

An advantage of having a dedicated course for EDP after the introductory programming course would be the possibility for the course content to concentrate on various applications of EDP instead of competing with other fundamentals of programming. Related to that, EDP might be easier to comprehend after other fundamentals are learnt first on earlier courses. There are potential disadvantages as well. For instance, it might be desired to apply EDP-related knowledge earlier in the curriculum in areas including graphical user interfaces as well as server and low-level (hardware) programming. Furthermore, the students’ motivation might decrease while waiting to learn something that they might consider to be interesting. Finally, in case the course were elective, perhaps not all students would enroll on it, which in turn would diminish the reach of the education on an important subject area.

In contrast to the above advantages of a later and broader exposure, one could hypothesize that an earlier exposure to the fundamental ideas of EDP might produce better learning outcomes in time. This would be more likely if the ideas were initially understood correctly and then gradually repeated, broadened, and explained in more detail in later courses. On the other hand, an insufficient or incorrect initial comprehension of the ideas might hinder later learning of both EDP and other related or similar subject areas.

Yet another aspect to consider is how learning one style of programming might hinder learning another style. For instance, Weintrop et al. [151] discussed potential problems when transferring from visual block-based languages to textual sequential languages. The teaching should take measures to alleviate such problems for instance by scaffolding students in proper ways of reasoning about the problematic areas.

In addition to the above remarks, this dissertation does not take a position on which approach is better. The computing education literature seems not to contain experimental studies on the matter, so new studies in various well-described contexts would be needed to gain indications on which approach, if any, is more beneficial.

5.4 Implications for Computing Educators

This dissertation offers practical insights on improving the everyday practice of teaching event-driven programming. Although the concept of event
was generally understood well in the study of *Publication IV*, educators should pay more effort to explicitly clarify (1) the difference between the occurrence and its representation in the computer as well as (2) other concepts and their relationships. The areas of clarification might include

1. adding (registering) an event handler—the essence of the *Observer* design pattern (see § 2.4.2)
2. event handling from the occurrence to the call of the event handler
3. storing program state outside the event handlers
4. executing event-driven programs (see also § 5.3.3).

The execution model of event-driven programs might be challenging to grasp for novices. For instance, if the event loop is hidden inside the framework, not a single line of the student’s own program code is necessarily being executed at a certain moment, when the program that contains the event loop is being executed.

In addition to the above, another potential difficulty is that the student’s own “program” might not contain a main program, which also causes it to lack a visible entry point. This is the case if, for instance, JavaScript methods are called declaratively via the event-handling attributes (e.g., `onclick`) of Document Object Model elements and there is no program code outside the called methods. The program code in such methods could be thought as tiny but “full” programs, as they are what the student actually sees. However, such a situation seems to be more about extending the program that implements Document Object Model than developing an individual large-scale program oneself.

Visualizations and skeletal code examples might help in illustrating the structure and the execution model of event-driven programs. One tool that can help in visualizing high-level concepts in learning materials is discussed in *Publication VI*. Furthermore, concept maps might help students in understanding and memorizing concepts and their relationships. More suggestions for improving teaching are presented in the following subsections as practical generalized principles.² Implications for computing education researchers are discussed later in § 5.6.

### 5.4.1 Consider Everyday Meanings of Terms

**CONTEXT** Students’ exposure to both spoken and programming languages might affect their understanding of new terminology. When they encounter unknown terms, they might use their intuition to explain those terms to themselves. For instance, the term *event listener* might be understood as something that is *actively* trying to hear or receive something

² Principles are supposed to be followable most of the time as standard approaches, but there might be exceptions, some of which are discussed with the principles described in this section.
(see Appendix D) instead of being a passive subprogram that is called and executed only when a related event needs to be processed [P1V].

**Therefore** Consider the real-world meanings of the terminology being used. When possible, choose to use terms that intuitively give a correct idea regarding the related concepts. Otherwise, be sure to explain the exact meanings of potentially misleading terms to the students.

**Dependencies** The programming environment might use non-ideal terminology. Consequently, the learning environment is forced either to use the same terminology everywhere or to use other terms outside the programming environment with careful explanations.

### 5.4.2 Explain Similar Terms and Environments

**Context** Some terms, such as *event handler* and *event listener*, are synonymous. On the other hand, the meanings of a single term might slightly differ in various contexts, such as that of *an event listener* between JavaScript (often a function) and Java’s *Abstract Window Toolkit*\(^3\) (interfaces and implementing classes that describe/contain methods for various types of events). Multiple terms for one concept as well as one term referring to (slightly) different concepts can potentially confuse students.

**Therefore** To avoid misunderstandings such as those in the study in Publication IV, pay attention to the programming environments that the student populations have used earlier. Then, clarify similarities and differences regarding the topics at hand as well as other known terminology issues. Otherwise, use only one term per concept (§ 5.4.3). Optional information boxes specific to terms or environments are one way to implement these clarifications in learning materials. Also, if a separate terminology section or explaining pop-up windows are used, the clarifications should be added or cross-referenced in those as well.

**Dependencies** The terminology used in the programming environments and information sources (see § 5.4.4) that the student population has used earlier might affect their thinking.

### 5.4.3 Have a 1–1 Match Between Terms and Concepts

**Context** The term *event* is being used in multiple meanings: The occurrence as well as the record that represents such an occurrence in the computer. On the other hand, a subprogram that is called to handle events is sometimes called as *an event handler* and sometimes, for instance, as

\(^3\) *Swing* [348] superseded *Abstract Window Toolkit* (AWT) [347] but is built on top of it and leans on its event interfaces and classes (e.g., the *java.awt.event* package) in handling, for instance, keyboard and mouse input.
Discussion

an event listener. Various information sources and experts, including educators, might use multiple terms for one concept interchangeably, or they might refer to multiple concepts with one term. Although the educator knows the meanings of the terms related to their subject, one-to-many and many-to-one relationships between terms and concepts are potentially apt to confuse the students—especially novices [PIII, PIV].

**THEREFORE** Learning environments (e.g., courses and learning materials) should have one-to-one match between terms and concepts and, by principle, stick to those relationships everywhere.

**EXCEPTIONS** This principle might be momentarily disregarded when explaining terminology, such as synonymous terms and differences in terminology between various contexts, such as programming languages, tools, and environments (see § 5.4.2).

**DEPENDENCIES** The terminology used in the programming environment is likely to dictate the choice of terminology in the learning environment.

### 5.4.4 Consider the Suitability of External Resources

**CONTEXT** Among other things, external resources might contain jargon and explanations that are unsuitable for the novice programmer and might confuse them. This might have been one factor that caused misunderstandings in the studies of Publication III and Publication IV.

**THEREFORE** Before referencing or quoting external resources, consider the possible difficulties, such as terminological conflicts, with the primary learning environment. Consider if the reference or quote is appropriate for the students and avoid using unsuitable resources. That said, in case the students are to become professional developers, sooner or later they have to familiarize themselves with using many kinds of technical documentation.

Whatever the reason for interaction with external resources, the educator should provide scaffolding to ease the students’ possible “culture shock” with the resources such as documentations of programming languages, environments, tools, and application programming interfaces. To assist students’ familiarization process, the educator might provide separate exercises on using them. For instance, as a scaffolding method towards being able to use Javadoc-style documentation, McAvoy [328] suggests creating similar lecture slides with guidance, and then transitioning towards using real documentation through exercises.

**DEPENDENCIES** Using some resources, such as documentation of a class library, might be unavoidable in practice.


5.4.5 Consider to Explain Even the “Obvious”

**CONTEXT** As the study described in Publication II demonstrated, students might not know or understand matters that the educator might think as obvious. For instance, it is easy to expect that everyone understands how graphical user interfaces and interactive applications work. Still, the students might lack adequate exposure to correct terminology, exact definitions, and descriptions of usual operating principles. This might lead them to construct their own incomplete or incorrect ideas by guessing.

**THEREFORE** Consider what matters the students should know and if they actually know them. Then, explain the potentially unclear matters. Regarding user interfaces, it might be beneficial to increase the coverage of related fundamentals on introductory computing courses.

5.4.6 Evaluate Actual Learning Gains

**CONTEXT** The mapping review in Publication I demonstrated that in many publications that mention or discuss teaching or learning event-driven programming (EDP), it is taken for granted that students “learn” EDP just because they are subjected to some teaching tool or approach.

**THEREFORE** Instead of making grand and generalizing assumptions, evaluate the actual learning gains: For instance, what individual concepts and skills are covered in teaching and in what way, compared to how well the students have internalized the content and are able to apply it.

5.5 Limitations

This section summarizes limitations regarding the research presented in the publications that are included in this dissertation. It starts with common aspects and proceeds to discuss the studies individually.

**Researchers.** In all studies in this dissertation, the researchers responsible of the studies were doctoral students. However, more experienced researchers participated in some aspects of the studies as well as gave constant feedback through planning, execution, and reporting of the studies. All the researchers were from a single university, so the research lacks active influences from the diversity of other research cultures.

Publications II, III, and IV included interpretation of students’ answers to exercises and surveys. As qualitative interpretations are results of the researchers’ consideration, the personal experiences, expectations, and knowledge might have affected the results. Also, because all researchers were native Finnish people, a theoretical possibility of misunderstandings and oversights exists in the interpretation of answers given by students.
from other cultures. However, the significance of the researchers’ mindsets has been decreased by having two or more researchers to analyze the same material. Furthermore, the questions and the answers were technical rather than culture-related, so the probability for culture-related misunderstandings and oversights should be small.

**Languages.** All researchers spoke Finnish as their first language, so a theoretical possibility of misunderstandings and oversights exists in the interpretation of students’ answers given in English. However, both the content and the language used in most answers was simple.

A more realistic possibility for misunderstandings exists in the interpretations of students’ vague computing-related terminology—both in English and in Finnish. As a preventive measure, students’ answers were mostly analyzed by two or more researchers who then discussed until reaching a consensus. In Publication IV, the analysis of the concept comprehension exercise answers was partly checked by the second author.

Finally, in the mapping review of Publication I, the search terms were only in English, and a few publications were rejected on the basis of not having been written in English. Consequently, some relevant work might be missing from the set of the reviewed publications.

**Studied Populations.** The two courses that served as contexts for the studies described in publications II, III, IV, and VI were offered by a single university. Furthermore, the student populations of both courses are lifelong learners from wide range of ages, educational backgrounds, and programming experience. These populations do not represent typical university freshpeople, which may threaten the transferability of the results. However, because II and IV as well as the relevant results of III were all qualitative, the diversity of the participants contributes towards the results’ trustworthiness.

**Publication I.** Probably the largest potential limitation of the mapping review is that it is restricted in the scope of literature it covers. First, various terms have been used to refer to event-driven programming (EDP). To lessen this problem, the search criteria contained several variants, such as `event-driven`, `event-oriented`, and `event-based` programming.

Second, plenty of EDP-related literature still exist in various fields of computing, and many topics could be argued to be related to EDP even without EDP being explicitly mentioned at all. Consequently, to keep the scope of the review realistically manageable, the searches had to be limited to journal articles and conference papers that explicitly mention EDP, and some of the searches were further limited to specific metadata fields. This may have caused the review to miss some relevant literature.

Regarding the review process, the *Searching* and *Rapid Screening* phases were performed by only one researcher, which might have caused some publications related to teaching and learning to become overlooked and excluded. However, as the teaching aspect is often reported in the abstracts
of the publications, the risk for this should be low. In the Detailed Screening phase, each publication was considered initially by two of three researchers. If they disagreed, the third researcher also considered the publication, after which the final result was negotiated until a consensus was reached.

Finally, the searches were performed during September 18–22, 2018, so the review does not reflect more recent literature. Moreover, the citation counts reported in the review mostly reflect the data of Scopus and will naturally change over time.

**Publication II.** One limitation related to the study on students’ thinking about buttons was anonymity. Consequently, the detection of duplicate submissions was not completely reliable, and it was not possible to combine a wider selection of personal details into the analysis.

**Publication VI.** The pilot study of the visualization tool prototype had two unfortunate limitations. First, the logging did not record user identifiers, which made it impossible to analyze user behavior over session boundaries. Second, the visualizations used in the study did not have separate starting and ending states, which made it impossible to measure the total time the students viewed the visualizations. Still, the usage of the prototype on two online courses demonstrated its functioning in the extent of the visualizations used on the courses.

The prototype was developed in EcmaScript 6 [166] that, for compatibility with Internet browsers, is transpiled into EcmaScript 5 with Babel [301]. To simplify development and to mask differences between browsers, the project uses a third-party runtime library called jQuery [344]. The build process is automatized with Gulp [320] and leans on Node.js [345]. At the moment of writing, the program code needs refactoring, automatized testing, and documentation. In addition, § 5.6 lists possibilities for developing the features of the prototype further.

### 5.6 Future Work

The mapping review described in **Publication I** demonstrates that not much empirical research is published regarding teaching and learning event-driven programming (EDP). The other studies presented in this dissertation take steps towards filling this research gap, but much more could be done. This section summarizes some directions for future work regarding the topics related to this dissertation.

**Concepts and Skills.** In the core of EDP are the concepts and skills that students should learn. There seems to be little earlier experimental research on the actual learning gains regarding EDP, and some publications have expected that the mere exposure to some tool or approach will cause the students to “learn.” To alleviate the situation, it should be studied what exact concepts and skills have been learned, if any, and how well.
Publication III and Publication IV provide a baseline for new research on how students understand concepts related to event-driven programming (EDP) and what kinds of misunderstandings they might have. However, this line of research should continue with varying student populations and programming environments. Interviews and think-aloud studies could help to discover the root causes of why students think in specific ways. In addition, developing a widely-accepted instrument for assessing students’ learning regarding EDP would help both teachers (see § 5.4.6) and researchers.

**Visualizations.** Studying might be accelerated with visualizations, and § 5.4 proposes topics that could benefit from being visualized. However, not all visualizations might be helpful, and it should be studied how to best visualize EDP-related concepts and what is the actual impact of such visualizations. Publication VI presents a tool that can help in composing visualizations, but the technical capabilities of the tool could be enhanced further. Approaches for that include

1. extending the state machine to enable non-linear visualizations
2. enabling the addition of actions to the visualization elements using a programming language with variables and conditions
3. supporting embedding of various types of exercises
4. improving the reusability of steps and actors
5. improving visual capabilities, such as smooth animation and actors defined using Scalable Vector Graphics
6. developing a graphical editor for composing visualizations.

**Notional Machines.** If the event handlers of an event-driven program are being called from outside the programmer’s own code (that is, by the framework), the execution model and the notional machine of such a program should differ from those of programs, whose subprograms are called only by other code in the same program. However, further research should find out, for instance, exactly what kind of notional machine would be appropriate for teaching EDP and how does that machine differ from other notional machines, such as those for object-oriented programming (see also § 5.3.3 and § 5.4).

**Software Development.** Students’ practices, strategies, and difficulties in designing, programming, and debugging event-driven programs should be studied. Methods for this include observing students’ coding and debugging processes as well as interviewing them about their projects.

**Methods and Tools.** Although some publications compare teaching tools on the basis of theoretical discussion, educators would need experimental studies that compare EDP-related tools and approaches regarding actual EDP-related learning gains. The same applies to studies, in which a new tool or method for teaching EDP is introduced or an existing one is altered.
Programming Environments. There are differences between programming environments (programming languages, program code libraries, integrated development environments, etc.) in terms of practicing event-driven programming (EDP). Also, programming paradigms, such as object-oriented and functional programming, might cause differences regarding learning EDP. This calls for adjusting the teaching approaches according to each environment. Educators and researchers should study what differences there are and how EDP should be taught in various environments. Furthermore, studies should address students’ capability and difficulties in transferring their skills from one environment to another.
6. Conclusions

The term *event-driven programming* (EDP) has been used in the industry and the academia to describe producing program code for computer software that respond to occurrences called *events*. Such technologies, including most web servers and most graphical user interfaces of all kinds of computers and browser-based services, have become mainstream in the modern society. As computer applications developed by the industry continue to enable new processes as well as to streamline and automatize existing ones, it has become crucial for the software developer to have a good comprehension of EDP. Consequently, the topic should have a corresponding importance in the software developers’ education.

EDP has some coverage in the computing education literature. Some publications mention EDP as a part of some course or curriculum. Many present approaches and tools for teaching EDP, and some describe anecdotal evidence and opinions, for instance, the suitability of EDP for the novice programmer. Unfortunately, although EDP has been a part of some empirical studies, not much experimental research that would purposefully focus on teaching and learning EDP seems to have been published. This research gap invites more research in the future.

As any subject area to be studied, EDP is subject to potential misunderstandings. The findings described in this dissertation revealed various flawed perceptions regarding, for instance, dispatching events and registering event handlers. The knowledge of potential misunderstandings helps educators to address them in their teaching to avoid students developing them and to correct them if they do appear. In addition, this dissertation offered practical suggestions for improving the teaching of EDP.

In addition to applying knowledge of potential misunderstandings, teaching of EDP could be improved through contextualized teaching. This dissertation has discussed contextualization approaches and tools, both generally and related to media computation specifically. These approaches can be used to make studying more meaningful and interesting, especially for specific groups of students. This might improve student engagement and reduce the number of students dropping off from courses.
Conclusions

A third approach for improving the teaching of event-driven programming (EDP) is to improve the learning materials by visualizing potentially difficult concepts. To facilitate the creation of such visualizations, this dissertation presented a prototype of a tool for composing visualizations suitable to be integrated into many online learning materials.

The six studies discussed in this dissertation form a basis for further research on teaching and learning EDP both in general and in the specific research lines of the individual publications: conceptual understanding, contextualization, and visualizations regarding EDP. It is unfortunate that teaching, learning, and generally communicating about EDP suffers from ambiguousness in terminology, as a generally accepted definition seems not to exist for the exact meaning of EDP. This should be considered in the future usage of the term. Also, other more exact and descriptive terms could be considered to be used instead.
Publications in Journals


Bibliography


Publications in Conferences and Workshops


Bibliography

Book Chapters


Theses


Bibliography

Standards


Model Curricula


**Textbooks on Introductory Programming**


Bibliography


Textbooks on Design Patterns


Bibliography


Bibliography


Books on Miscellaneous Topics


Bibliography


Magazines


Web Sites on Tools for Contextualized Programming Education


[258] [Codemancer] Important Little Games Inc. *Codemancer – A fantasy game that teaches the magic of code.* 2022. URL: https://codemancergame.com/ (visited on Jan. 11, 2022).


Bibliography


Bibliography


[293] **Snap!** Snap! Build Your Own Blocks. URL: https://snap.berkeley.edu/ (visited on Jan. 10, 2022).

[294] **StarLogo Nova** Scheller Teacher Education Program at Massachusetts Institute of Technology. Welcome to... StarLogo Nova. URL: https://www.slnova.org/ (visited on Jan. 11, 2022).


Web Sites on Miscellaneous Topics


Bibliography


Colleen Lewis et al. *Misconception: Students think that a “repeat 1” block in Scratch will execute the relevant script twice, where the script executes once and then repeats once.* 2022. URL: https://www.csteachingtips.org/tip/misconception-students-think-repeat-1-block-scratch-will-execute-relevant-script-twice-where (visited on Mar. 3, 2022).


Bibliography


A. Textbooks That Teach EDP in Connection with GUls

This appendix illustrates the frequency of introductory programming textbooks that discuss event-driven programming (EDP) in connection with graphical user interfaces. These books include the following ones:

Arnow, Dexter, and Weiss 2004. [177, p. 203]  
*Introduction to Programming Using Java™: An Object-Oriented Approach.*

*Objects First With Java: A Practical Introduction using BlueJ.*

Bishop and Horspool 2004. [179, pp. 170 & 351]  
*C# Concisely.*

Bravaco and Simonson 2010. [180, p. 954]  
*Java Programming From the Ground Up.*

Budd 2002. [182, p. 428]  
*An Introduction to Object-Oriented Programming.*

Charatan and Kans 2006. [183, p. 228]  
*Java in Two Semesters.*

Cohoon and Davidson 2004. [184, p. 290]  
*Java 1.5 Program Design.*

Deitel and Deitel 2016. [186, p. 597]  
*Java™ How to Program.*

Eckel 2003. [187, pp. 360 & 780]  
*Thinking in Java.*

Gaddis 2013. [188, p. 782]  
*Starting Out with Java™: From Control Structures through Objects.*
Textbooks That Teach EDP in Connection with GUIs

**Goldwasser and Letscher 2008.** [190, p. 493]
*Object-Oriented Programming in Python.*

**Guzdial and Ericson 2007.** [191, p. 535]

**Guzdial and Ericson 2011.** [192, p. 347]
*Problem Solving with Data Structures Using Java™: A Multimedia Approach.*

**Horstmann 2010.** [194, p. 331]
*Java Concepts.*

**Lewis and Loftus 2009.** [195, p. 219]
*Java™ Software Solutions, Foundations of Program Design.*

**Lewis 2013.** [196, p. 259]
*Introduction to the Art of Programming Using Scala.*

**Liang 2011.** [197, p. 558]
*Introduction to Java™ Programming: Comprehensive Version.*

**Roberts 2008.** [198, p. 350]
*The Art & Science of Java™: An Introduction to Computer Science.*

**Sanders and Dam 2006.** [199, p. 225]
*Object-Oriented Programming in Java™: A Graphical Approach.*

**Savitch 2016.** [200, p. 983]
*Absolute Java.*

**Sedgewick and Wayne 2008.** [201, p. 437]
*Introduction to Programming in Java™: An Interdisciplinary Approach.*

**Wu 2010.** [203, p. 790]
*An Introduction to Object-Oriented Programming with Java™ 5th ed.*
B. Screening in the Study of PII

The study that is discussed both in § 3.3 and in Publication II was concerned with students' thinking of buttons and their effects in three scenarios. The process of screening the survey responses that were received in the study is presented in Figure B1.

During the date interval from the earliest final accepted response to the latest one, 412 students worked with exercises on the online learning material page that contained a link to the survey used for data collection. Of the 191 responses received, four duplicates and the responses of two possibly-minor participants were removed. Consequently, there were 185 accepted responses to be analyzed.

Figure B1. Screening of students' responses in the study of Publication II.
C. Screening in the Study of PIV

The study that is discussed both in § 3.4 and in Publication IV was concerned with students’ understanding of concepts related to event-driven programming. The process of screening the exercise submissions that were received in the study is presented in Figure C1.

![Diagram of screening process]

Figure C1. Screening of students’ responses in the study of Publication IV.

Of the 130 course participants, 85 answered the enrollment survey, but 11 were rejected due to missing research consent or age. Of the 74 accepted respondents, 28 answered at least one of the two exercises. One off-topic answer was removed, which left 24 concept comprehension exercise answers and 26 code comprehension exercise answers for analysis.
D. The Meaning of the Verb “to Listen”

To illustrate the activeness of the activity that the verb *to listen* describes, this appendix quotes its relevant meanings from four English dictionaries. Emphases are added by the author to highlight the essence.

**Cambridge English Dictionary**

- *To give attention* to someone or something in order to hear him, her, or it.

**Collins English Dictionary**

- If you listen to someone who is talking or to a sound, you *give your attention* to them or it.
- If you listen for a sound, you *keep alert and are ready to hear it* if it occurs.

**Macmillan English Dictionary**

- *To pay attention* to a sound, or *to try to hear* a sound.

**Merriam-Webster’s online English dictionary**

- *To pay attention* to someone or something in order to hear what is being said, sung, played, etc.
- To hear something with *thoughtful attention: give consideration*.
- *To be alert* to catch an expected sound.

---

3 [https://www.macmillandictionary.com/dictionary/english/listen](https://www.macmillandictionary.com/dictionary/english/listen), visited on — —.
4 [https://www.merriam-webster.com/dictionary/listen](https://www.merriam-webster.com/dictionary/listen), visited on — —.
Errata

Publication I

Page 2, paragraph 1, line 10:
Computing Science Curricula should be Computer Science Curricula.

Page 22, paragraph 7, line 1:
Breath-first course should be breadth-first course.

References:
The paper Introducing OO Design and Programming with Special Emphasis on Concrete Examples by Erzsébet Angster, Joseph Bergin, and László Bőszörményi from 1999 was listed with an incorrect Digital Object Identifier (DOI). The correct DOI is 10.1007/3-540-46589-8_20.

Publication II

Footnote 2:
European Credit Transfer System should be European Credit Transfer and Accumulation System.

(continues on the next page . . .)
Errata

Publication IV

Page 6, paragraph 5, line 1:

Incorrect:
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).

The 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).
Computer application that receive information about external events and produce responses to them are common in the modern society. Such applications include web server software, many services used through Internet browsers, as well as most graphical user interfaces in devices from mobile phones and home appliances to general-purpose computers and industrial applications. The development of such applications is often called *event-driven programming*. Even though learning it is important for a software developer, not much empirical research has focused specifically on teaching and learning it.

This dissertation takes three approaches to help educators in improving teaching of event-driven programming in introductory programming education. First, it explores how students understand central concepts of event-driven programming. Second, it presents approaches for contextualizing the teaching of event-driven programming to potentially make it more interesting. Third, it discusses a prototype for a visualization tool that can be integrated into online learning materials for illustrating high-level concepts, including those of event-driven programming.