Reconsidering End-User Development Definitions

Nikolaos Batalas[1(✉)], Ioanna Lykourentzou[2], Vassilis-Javed Khan[3], and Panos Markopoulos[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
p.markopoulos@tue.nl
[2] Utrecht University, Utrecht, The Netherlands
i.lykourentzou@uu.nl
[3] Sappi Europe, Watermael-Boitsfort, Belgium
Javed.Khan@sappi.com

Abstract. We consider definitions that End-User Development and related fields offer for end-user developers, and identify the persistence of viewing end-user development as antithetical to professional development across the years, even as focus has shifted from the identity and then to the role of the developer, and later to the intent of the development effort. We trace the origins of this antithesis to the days of End-User Computing in organizational settings, and argue that modern software development resides in a different paradigm, where end-user Development is part and parcel of any programming endeavour, in professional or other settings. We propose that current development practice, both for those traditionally regarded as end-user and as professional developers, can be better served by EUD as a field, if the focus is shifted to the nature of the task itself, and how technical it needs to be, by way of the platforms that development takes place on.

Keywords: End-user development · Technical development · Definitions

1 Introduction

End-User Development (EUD) is a field of academic research dedicated to the making of software. Related fields such as End-User Programming (EUP) or End-User Software Engineering (EUSE) also address specific aspects of software creation/maintenance. The terms are distinct; EUP can be considered as being about program creation itself, whereas EUSE addresses concerns around reliability, reuse and maintainability. EUD claims to address the wider range of practices that are involved in software development, including design practices. As such, EUD can be considered to be the more encompassing term, inclusive

D. Fogli et al. (Eds.): IS-EUD 2021, LNCS 12724, pp. 19–35, 2021.
https://doi.org/10.1007/978-3-030-79840-6_2

of EUP and EUSE[1]. In this text, the term end-user development will be the umbrella term for all aspects of practice with regard to the end-user production of software, unless referencing specific cases, and EUD will denote the wider field of research, inclusive of topics researched by EUP and EUSE.

EUD's objectives for software construction differ from the traditional academic disciplines dedicated to the task. Such disciplines as Computer Science (CS), Electrical Engineering (EE) or Software Engineering (SE) emphasize specialized knowledge that relates to the construction of software, on areas that include the following:

- the construction of computing machinery on a physical substrate, and the ways these can be controlled and composed into more complex systems that are able to execute programs (which can generally be seen as sets of instructions that operate on data, or produce signals for adjacent systems).
- the ways in which computational problems can be classified, and the finding of efficient solutions for solving classes of problems, in terms of execution time and memory use, as well as studying the properties of relevant data structures.
- the methods via which the development and maintenance of software can be practiced systematically and with discipline, so that reliable results can be reached with predicable use of resources.

EUD on the other hand, aims to make easier the construction of computer programs, or the modification of existing software to alter or extend its functions, but without demanding that the developers should have to employ the depth of knowledge and expertise that it takes to develop software on the technical level that the traditional disciplines are concerned with, regardless of how familiar they are with them.

This paper raises the issue that consideration needs to be placed towards the ways with which academic communities regard End-User Development and its place within the wider landscape of software production. It is motivated by a contradiction observed when applying prevailing definitions of end-user developers to a specific case of software development in clinical psychology, and which arises from the tendency to define end-user developers by juxtaposing them against professional developers. The paper traces the origins of the term end-user developer, and the evolution of software development practices, to show that on one hand, conceptions of end-user developers are rooted in organizational settings of the past that do not necessarily persist any more, and on the other hand, that an

[1] It is interesting to note here the observation by Barricelli et al. [2], that the choice of field to which authors will ascribe their work tends to be a matter of academic culture. European authors will file their work under EUD because of its namesake European Commission initiative, the European Network of Excellence on End-User Development (EUD-Net) which created a network of researchers and relevant conferences. American authors, on the other hand, will prefer the term EUP, which did originate in the United States, and a small subset of those, the community of US universities that participated in the EUSES Consortium pursue work under EUSE.

important quest of software development has been to render itself into an EUD endeavour, with successes along the way. Finally, it invites researchers to reconsider the scope of EUD, from addressing specific types of expertise or intent, to encompassing the whole range of software development.

2 A Cause to Reconsider Commonly Used Definitions

Over time, various definitions of EUD have been offered with regard to who the end-user developer is or what they do. Although the end-user developer tends to be defined in some way as an opposite of the professional developer, attempts at definitions have varied, mostly in service of illustrating a particular point that the author is making. With regard to the workplace, some authors have seen end-user programmers as those who have non-programming jobs to perform, or do not care about computers, but still have to program [14,41]. In researching more accessible ways to produce software, authors have regarded end-user developers as novices to computer programming or less skilled at it [31,46]. Yet others have argued that skill and expertise is irrelevant [29].

In more detail, considering someone as an end-user developer due to aspects of personal identity (e.g., by being a novice [14] or not caring about computers [41]) excludes potential categories of end-user developers, such as system administrators or research scientists, labelled as professional end-user developers [51], who possess or acquire the technical knowledge to develop software in order to further their professional goals. To overcome such issues, Lieberman et al. [32] cast the end-user developer as *a role*, in which someone *acts as* (rather than *is*) a non-professional, and offer a definition for the field of research, rather than the end-user developer or their activities. They define EUD as "a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artefact". The antithesis to the professional developer, found in earlier definitions, remains, but a definition for the end-user developer is missing.

In a later definition, Ko et al. [29] acknowledge the problems of considering end-user programmer characteristics as a matter of personal identity. They propose that the characterization of one as being end-user vs. professional developer is a matter of intent, and as such, expertise and skill in programming are irrelevant. End-user and professional developer are two endpoints of a spectrum, and one's place on it is determined by the number of users they are building the software for. If they are serving their own needs, they are end-user developers. If they are serving a large population of users of their software, they are a professional developer. The authors here offer a way to discern between end-user development and professional development, since the number of users functions as an indicator of whether the software is intended for personal or wider user, and therefore a measure of how important it is for the software to be reliable, and therefore might require professional software engineering practices. According to this view, end-user development is development for one's own self, irrespectively of how experienced the programmer is, whereas developing for others, is a characteristic of professional development.

Ambulatory Assessment is a research method predominantly used within clinical psychology, with the purpose of capturing bio-psycho-social processes in the context of daily life [12], gathering self-reports and sensor measurements from groups of people, by sampling them repeatedly over time. Researchers who use it, deploy sampling instruments to a population of participants in their natural settings, and use the collected data to discover ways in which the constructs under investigation relate to each other. Oftentimes, they also seek to offer personalized interventions to each participant, based on data collected. Data collection happens increasingly via mobile platforms such as smartphones and wearables, and considerable effort goes into the making of tools to enable clinical psychologists to define what data these platforms collect and how [3,22,49]. With these tools, psychologists do not merely produce parameters for the configuration of prebuilt software, but effectively write programs consisting of function calls, which perform such actions as instantiating user-interfaces or invoking the sampling of hardware sensors, which are either executed sequentially or as a result of conditional branching according to the evaluation of logical statements (if-then-else), which involve variables representing user or system states.

Researchers who employ these methods, distribute their programs to potentially hundreds of users, who participate in their studies, to use for recording data. They might also share them with other members of their research community, who wish to reuse them. We would traditionally (in terms of identity or role) reason about these psychologists as end-user programmers, given the fact that their formal training or work practices do not usually include elements of CS, EE or SE that would justify viewing them as professional programmers, and the fact that the tools that allow them to create these programs are tailored to their own professional domain. Yet in producing and distributing their programs, they do not singly intend them for themselves or for others, but for both. They aim both to collect longitudinal data for their own research purposes, but also to produce programs that their participants can use to supply this data. Depending on what view of their intent we espouse, we could use the criteria proposed by Ko et al. [29] to classify the same activity as either end-user programming, or professional programming, but definitions of end-user programming or development consider these notions to be opposites. This contradiction motivates us to wonder about the discriminatory power these definitions, which define end-user development as opposite to professional development, have across various modern development configurations.

For this reason we will try to better understand the two aspects that these definitions address, that of the end-user and that of the professional developer. In the rest of this paper we will discuss these concepts in more detail, and argue that modern software-development practices in professional settings are rife with EUD success stories. Given our motivating contradiction, we will suggest that it is less fruitful to regard EUD-research as applicable only to non-professional instances and it is worth to pursue a definition of end-user development driven by platform and outcome instead.

3 End-User Development Is an Evolving Concept

The term *end user* is arguably an invention of IBM in the 1950s [34]. It was used to point to people, such as corporate executives, who would be the budget holders responsible for commissioning the purchase of computing technology [42]. They were considered to be separate from *intermediate users*, usually experts who would operate the machines [45], working in units known as data processing departments, and tasked with performing computations in answer to questions posed to them by management.

In the 1960s and 1970s, solid state transistors and the microchip brought speed and power to mainframes, and gave rise to minicomputers. Computing became dramatically cheaper and thus more accessible to members of organizations working outside of data processing centers. The end users were now people with access to machines, and computing departments started dealing with the strategies for organizations to provide their members with access to applications of the enterprise, as well as manage their workload.

As the trend continued in the late 1970s and 1980s, employees were able to procure their own personal microcomputers [4] independently of a dedicated department. End users were now the users of application software which they did their own data processing with. The field of End-User Computing (EUC) came more prominently into being, specifically concerned with enabling and supporting computing performed by employees within organizational settings. Growing demand for software solutions led EUC research to consider how to enable these end users to become developers of their own programs [36], and practice End-User Programming (EUP).

For the purpose of studying the use of computers in organizational settings, several taxonomies of users were proposed [11,13,36,48], examining what sort of use was made of Information Systems and for what purposes. Classifications of this sort make sure to set apart data-processing professionals, who are employed to write code for others. Workers of the organization, who are trained in other domains but write code, are classified as amateurs [36] and are considered to only write code for themselves [56], evidently so because they are not employed to write code for others in the first place. It should be noted that data-processing professionals are designated thus by decree of the organization. Demand for such workers was too great to fulfill by sourcing them from specific educational backgrounds or certifications, as is the case with members of a traditional professional class, and in order for employees to qualify for the computing-related departments, organizations would oftentimes have to provide specialized training [50].

Therefore, each role of programmer is essentially fixed to the department's mission, with employees of the data processing department considered to be the professional ones. As a result, EUP is not concerned with programming professionals because it is not concerned with the data-processing department, and not necessarily because the professionals do not carry out similar tasks or do not need to be supported in similar ways, e.g. by making the understanding or modification of code more accessible for those just-starting professionals who do not yet have vast experience. This organizational distinction is perhaps the

reason why end-user developers are juxtaposed against professional developers, to this day.

During this time and into the 1990s, the Graphical User Interface (GUI) was popularized, and desktops became more user-friendly, and were adopted even more widely. As one of the main reasons to purchase desktop computers for offices, the electronic spreadsheet became one of the most popular applications for data processing by end users. It offered an easy to understand and visually manipulate data, and allowed users to perform bulk operations on it in ways more intuitive than the type definitions, loops and memory management of typical programming. It became one of the most prominent success stories in literature for EUP [8].

Gradually, EUC in organizations became less concerned with application software and EUP. Data processing departments evolved into Information Technology (IT) departments, managing the technology infrastructure which allowed an organization to run, i.e. hardware, networks, software licences and data storage, and supporting end-users in accessing it. Availability, scalability and security became the more central issues.

EUP moved into the domain of Human Computer Interaction (HCI) [40], where research was invested in understanding and supporting programming tasks, both as a general issue, and also within specific application domains. As computing became a staple of daily life in various forms, discussions on EUP also became disentangled from organizational settings. More recently, in the 2000s, research programs in the European Union and in the United States brought about EUD as defined by Lieberman et al. [32] and EUSE as discussed by Ko et al. [29], and which we discussed in Sect. 2. In both research programs, the contrast of end-users against professional programmers from the EUC days of large organizational settings has been carried over into the modern wider landscape of software development.

Regardless, it is easy to identify parallels between concerns that EUD pursues, and advances in software development practice. In Table 1 we list such counterparts to EUD pursuits surveyed by Patterno [44]. In the following section we will argue that many of the advances that are considered part of modern professional programming, are essentially EUD advances.

4 Software Development Is an Evolving End-User Development Practice

It can be argued that programmers have always tried to build platforms that would allow them to function as end-user developers on. That is, as people who want to get their work done and should not have to care about (some aspects of) the computer, as in Nardi [41]. We maintain that the evolution of computer programming and its related tools is very much an EUD success story.

Historically, many advances in computer programming have come in the form of layers of abstraction, whereby two things are achieved; the creator of the abstraction is able to suppress details of the underlying layers, which are irrelevant

Table 1. EUD pursuits and corresponding advances in software development practice

End-User Development (EUD) pursuits	EUD examples from development practice
A main goal in EUD is to reduce the learning effort that might be required to produce non-simple/complex functionality	Programming languages share similar goals. Features such as garbage collection and dynamic typing make for simpler languages, and less bureaucratic code that is easier to modify. IDE[a]-features like predictive text are helpful too
Some EUD approaches aim to enable users to compose and customize sets of available basic elements, which other programmers have developed	A lot of software libraries are produced in service of such goals, e.g., by packaging complex processes into simple purpose-specific function calls with accessible names and simpler argument lists[b]. Many examples can be found across systems of libraries specializing in GUI, audio, graphics, I/O, numerical methods, etc.
EUD investigates collaboration processes and environments, bur the diversity of the backgrounds of people involved might raise special concerns	Yet many software projects employ very different roles, requiring not only software design and coding, but also the facilitation of design processes, the production of sketching and prototyping materials, or documentation. Online collaboration tools support many of these tasks, using intuitive interfaces when GUIs are involved, or accessible syntax of text[c], for producing and sharing materials
A central issue of EUD is the discovery and utilization of intuitions, metaphors, and concepts familiar to the domain of interest so that designs can be explored and specifications for software can be produced	UX design methods, requirements elicitation, and agile practices aim to understand the domain in which a piece of software is meant to function, and derive the specifications, according to which it can be built so that it is effective and intuitively usable

[a] Integrated Development Environments (IDEs) are classes of applications that automate much of the work that goes into software development, e.g., maintaining code libraries and versioning, compilation, linking. They often include helper applications for writing code visually, e.g., UI editors where drag and dropping interface elements on a canvas generates blocks of code.

[b] An exemplar of this is jQuery, a JavaScript library for Web-browsers, mainly for manipulation of the Document Object Model (DOM), which is the data structure web browsers use to represent an HTML document programmatically. jQuery was ubiquitous in web development in the late 2000s and early 2010s. It provides a uniform Application Programming Interface (API) across all browsers, which at the time still had significant differences in their implementation of the DOM, offered higher level helper functions (e.g., click() on top of addEventListener()), which could also operate on aggregate objects and which are chainable, making for terse and easy to read programs.

[c] Markdown is an example of this, which is a now ubiquitous plain-text formatting language that keeps the original text readable, but also contains formatting instructions for parsers to produce rich documents e.g., in HTML.

to the programming task, and also to invent and express the model of a machine which is more relevant to the task, and perhaps even already familiar to the user of that abstraction [33].

In more detail, after initial innovations in performing binary operations with relays and switches [52] and the first electronic computers in the 40s, the 50s saw the rise of the stored program and the programmable computer, where the hardware does not need to be re-wired per program. Adams [1] discusses how subroutines, accessible as symbols of abbreviated words make it possible for the increasing number of computer users to produce usable programs of numerical analysis. His focus lies on allowing entry level programmers to achieve results, and envisions that a verbal statement of the problem will be sufficient for the computer of the future.

In subsequent years, a host of programming languages and compilers were invented by people who wished to program computers in terms closer to their level of expertise or to their application domain. Many of the innovations we regard today as arcane programming tools, were driven by the personal needs of their inventors to get their job done. For example, FORTRAN, offering a way to define algebraic expressions, was heralded as a "revolution", one that would "have engineers, scientists, and other people actually programming their own problems without the intermediary of a professional programmer" [18]. UNIX came into being because of the desire of its makers to have their own time-sharing system [47]. Programming languages at levels higher than Assembly, such as C, offer programmers the model of an abstracted computer, and allow them to (largely) not care about the particulars of the hardware itself. Fischer [20] acknowledges the promise of these innovations for making systems more "convivial" [26] a term which designates technologies that foster creative connections amongst people and their environments, combative to the alienation suffered on account of industrialization [10], being treated as mere consumers.

Innovations with regard to making code reusable, rendered so by programming-language constructs such as classes, objects, encapsulation, and distributing as code libraries, is a way of making these software artifacts end-user programmable. Notably, programmers in their daily practice set intermediate personal goals to structure their code in such ways as to build abstractions and interfaces and hide its complexity, so as to later render themselves end-users of it, and make it easier for themselves to get their job done by using it as a functional unit.

Furthermore, software development is not a single domain, and does not imply a uniform technical profile of a practitioner [15]. Different developers hone their craft on vastly different technical or creative problems, have domain knowledge on different levels of abstraction within the software-hardware stack, many of which have their own elaborate theoretical backgrounds (e.g. graphics programming vs database programming) and are end-users of various tools and platforms in order to carry out their work. Illustrative of this are job listings seeking programmers, which advertise not only for a particular application domain (e.g. front-end development) or a specific programming language (e.g. JavaScript) but for familiarity with specific code libraries and APIs (e.g. Angular vs React). It can very well be the case that the pro in one field is naive in another.

There's also a large selection of software tools in support of communities. Those involved in a project or making use of it can share their issues and seek support on how to solve problems in knowledge markets like StackOverflow. They can report bugs and propose desired features in issue trackers. They pull ready to use components from package managers which manage their updates automatically. They can put up for discussion and run programs in code sandboxes so that others do not have to replicate their development environment in order to view them. Such tools provide rich avenues for facilitating cultures of participation, which is also a vision for EUD [19].

5 Professional Software Development

In the previous sections we have seen that so-called professional computer-programming and software-development domains are regarded as separate from end-user development, to a large extent due to legacy organizational points of view for each practice. We have also pointed out that in many respects, as evidenced by directions in which computer programming practice has evolved, professionals pursue methods and tools that lessen the effort of their practice and increase the reliability of their outcomes, in directions parallel to those that EUD does. In this section we will examine various implications of the term *professional* as is applied to software development, and why it is not the best way to, by negation, define end-user development.

5.1 Connotations of Professionalism

The definition of the professional is a complex subject of sociology, and several approaches exist in establishing criteria by which to identify professionals, and the processes through which occupations become professions. Visiting the main trends through which sociology discusses professions and professionals, will give us some indication of the rich and complex landscape against which these discussions take place.

Trait-based views [23] derive sets of characteristics that distinguish a profession from an occupation, such as having an organized body of knowledge from which the provision of services flows, and having autonomy from employing organizations and authority over the recipients of their services (i.e., having clients, not customers). This authority is sanctioned by the community at large, imparted through accreditation by controlled training centers, and regulated through a formally established code of ethics. The distinction that such traits provide is considered to be quantitative, not qualitative [23], therefore occupations that are not professional will be found to also possess them, but to a lesser degree. This places any given occupation on a spectrum of professionalization, where at one end the traditional professions can be found (e.g., physician, attorney, scientist) and at the other those that completely lack these traits.

Various views examine how occupations become professions. The functionalist view regards the ways in which professions function for the benefit of the

larger societal context, and examines interactions between professions, society and structures of authority such as the state or military. Professions provide services based on knowledge that is that is both of great importance, and that could be harmful if abused. They ideally support social responsibility and contribute to the avoidance of authoritarianism and anarchy [16, page 17].

On the other hand, the conflict approach examines the process of professionalization, as motivated by the endemic self-interests of the professionals. Traits such as certification and licensing, regulated by professional associations, are seen as devices for occupational control, restricting the supply of labour and enhancing the status and earnings of the professional. To sustain such control, professional associations must also attend to the quality of their services [16, page 18].

Evidently, professions and the professionals are created through complex dynamic societal processes, and can have different expressions in different locales, e.g., they are often purposefully shaped by state policies. Therefore, professionalism cannot be reduced to mere technical expertise (as is the case with EUC), which is the focal point in organizational settings [17, page 100]. To do so, would be to exclude from consideration influential demands on the shape of both the professionals' modes of performing work, but also the realms of their responsibility. Many skilled labourers call themselves professionals, but they do so, as many others who perform skilled labour, within and in reference to the complexities and their manner of developing their knowledge, practicing their vocation and offering their services along their career path. Used in daily life, the term *professionalism* conveys the colloquial sense, and can be considered the opposite of amateur, associated with performing the work for payment, or not botching the job, concepts which aren't necessarily mutually exclusive.

5.2 Software Development Does Not Have a Singular Model of Labour

Developers of software in particular take up the occupation through a variety of paths, not all of which originate from academic education [54]. Indicatively, McConnell [35], summarising published demographics [24,53,55], notes how in the USA, there are 50,000 new developers each year, but only 35,000 software-related degrees are awarded each year. Muffatto [38, page 50] presents the findings of several studies on the demographics of open source developers, according to which, in terms of education 20% have just a high school degree, and in terms of professional background, 20% are students in academia. Paterno [44] states that "more and more applications are being written not by professional developers, but people with expertise in other domains". Developers can arrive from academic education, to training on the job, to self-study. The roles that participate in the making of software have expanded and diversified as computers acquire more capabilities and form factors. For example, whereas before the era of multimedia, in the 1980s, it was enough to have the skills of a computer programmer, in the era of the World Wide Web it became crucial to employ the

skills of a graphic designer. Nowadays the development of many types of software is increasingly an interdisciplinary effort, populated both by professionals in the more traditional sense, with education and certification in their own fields, and knowledge workers of more recent fields of expertise that take part in software development. For example, user-facing pieces of software have contributions from User Experience (UX) designers or psychologists (such as workplace psychologists) who pay attention to requirements, or projects with frequent update cycles employ experts in tools for Continuous Integration/Continuous Deployment (CI/CD).

As software permeates ever more aspects of daily life, its misfunctions, which happen by unintended design [21], can effect loss of income[2], amplify inequality[3] or even cost lives[4]. There is therefore very active discussion within software development communities but also in legal, governmental circles, and society at large, with regard to the greater responsibility that needs to be taken up by software developers, providers, and the regulation of their services. However, software development as such, still lacks the training, qualifications, and modes of regulation that are associated with the traditional professions [39]. There is yet no standard of care that software development professionals can be expected to uphold when cases are tried in legal courtrooms [9]. Potentially, a trajectory could be charted where future legislators will require certain types of software development to be carried out in the more traditionally professional sense, while others not.

To illustrate the different modes in which software can be produced, the diversity of roles that may take up its production, as well as how ubiquitous it is becoming in modern societies, it may be useful to draw analogies to the production, preparation and use/distribution/consumption of food in various settings, and the diversity of configurations in which this activity can be encountered[5]. Food is produced, processed, prepared, and consumed at various levels of preparation, and engages a wide range of workers with equally varying expertise, from highly trained, expert chefs who explore novel gastronomic horizons, to teenagers assembling hamburgers out of industrially manufactured compo-

[2] The cost of poor quality software in the US in 2018 was estimated to be approximately $2.84 trillion dollars, the largest component of which (37.46%) were losses from software failures, at $1.064 trillion [30].

[3] opaque algorithms assessing the risk of an offender repeating a crime, heavily used in the judicial system of the USA, have been suspect of encoding societal and racial biases [25], resulting in harsher punishments [43].

[4] The two fatal accidents of the Boeing 737 MAX aircraft in 2018 and 2019 have been attributed to Boeing's introduction of a software component, called the Maneuvering Characteristics Augmentation System (MCAS), which was working against the pilot's maneuvers. MCAS was unique to that aircraft, and its existence had largely been kept quiet [27].

[5] The choice of analogy is not unfamiliar. Algorithms are often compared to food recipes [28], and there is an abundance of programming "Cookbooks" for various frameworks and Software Development Kits (SDKs) or problem domains. One important exception is that to a certain extent, food production is more regulated than software.

nents, and of course also in non-professional environments, e.g., at home, for own consumption. They ways in which all handlers of food oversee the supply of materials, create recipes, and execute them, do have aspects that are specific to the person's particular position (e.g., freedom for initiative, or supply of special materials and tools), but also many other aspects (tools, methods and raw or processed materials) are shared widely across all configurations, irrespectively of whether they function as professionals or not.

Such is the diversity of configurations that can found in the production of software as well, e.g., having highly tailored solutions created for unique clients by highly specialized experts, or maintaining legacy systems from past eras of computing, or customizing the same blueprint for different customers. There is of course art, craft, and science in many stages of software development, and a growing, detailed body of knowledge with regard to good practices for producing software [7]. However, not all the components that make up a software product are developed in the same way. For example, even while inventing novel solutions for a particular problem, a developer will be the end user of packaged components that encapsulate often complex functionality. There is therefore no way to exclude particular methods or tools from the arsenal of anything that might for any number of reasons be considered professional practice.

If EUD's goal to facilitate design exploration, specification and implementation of software artifacts, and produce methods and systems that make problem solving through programming more accessible for people who would not be expected to develop software without its interventions, then it can provide similar benefits, such as easier implementation of complex functionality and greater accessibility to unfamiliar systems and to people who do develop software. Moreover, the methods and tools that EUD produces can certainly constitute means, with which software services that bear professional characteristics can be built.

As history has shown, (professional) developers will take up this challenge to empower themselves anyway. Taking these into account, given how software development has changed since the first investigations of EUP, and the multiple connotations of professionalism, it might now be opportune to explore other directions for defining end-user development, than continuing to use the professional/non-professional dichotomy. Table 2 shows how end-user development can be a concern orthogonal to professionalism, not opposite to it.

Table 2. Disentangling end-user development from professionalism allows more nuance in classifying development activities. Here, four broad-spectrum examples of domain and platform are mentioned, but further nuance can be afforded as one looks at different sub-problems and how they are solved.

	End-user development	Technical development
Amateur	Excel macros with Visual Basic For Applications	Home automation with Raspberry Pi
Professional	Interactive UI prototyping with inVision	Game development with the Unreal Engine

6 Replacing Professional Development with Technical Development

If professionalism is a concern independent from end-user development, then we propose that the term *technical development* take its place as the notion that is opposite to end-user development. In his Theoretical Introduction to Programming, Mills [37] devotes a section to the notion of Technical Programming:

> Technical programming is about defining a specific problem as clearly as possible, and obtaining a clear solution.[...] It has much in common with the technical (rather than bureaucratic) aspects of all engineering disciplines.[...] Precise sub-problems are identified.[...] What is or is not technical, depends on the techniques available.

The term *technical* translates only to the characteristics of the development task itself, not the person performing it, and denotes the kind of engagement with problem-solving that demands good grounding in methodology, and the ability to identify sub-problems and to give structure to the problem domain [37]. For example, where end-user programming would consist of, e.g., using function calls on a platform/abstraction, Technical Programming would be building the platform/abstraction in the first place, and exposing the functions that subsequently end-user programmers can call.

Software is developed on some sort of platform, or if viewed in greater detail, various components of a larger piece of software are developed on several complementary platforms. A Platform is a framework (be it hardware or software) that supports other programs. Platform studies [5] offer the theoretical framework both for conducting a discourse on platforms and when and whether it is useful to view a given system as such [6], not only from a technical, but also from a cultural perspective. We can regard as platform the hardware of a computer, an operating system, an API (e.g. OpenGL), a toolkit (such as Qt), or an application such as the Web-browser. Platforms that enable software development, expose concepts to the user in which certain types of problems and solutions can be directly expressed. For example, a programming language like C enables one to write loops, so when iterating over a set of instructions is the issue, there is a concept available directly related to that. Likewise, Matlab offers a function for computing the Discrete Fourier Transform (DFT) of a signal, so a solution that can be expressed in terms of a DFT can be supported by that platform.

A platform then enables end-user development for a given task, to the extent that it offers readily accessible functionality, that allows the task to be accomplished in more or less straightforward ways (e.g., a certain function call), rather than requiring the implementation of deeper-layered functionality in order to later enable it. On the other hand, technical development occurs to the extent that the concepts that are related to a particular solution also need to be developed, in order then to be used.

7 Resolution of the Contradiction

We propose that a platform-driven lens can be developed to help determine the nature of one's software development task at a specific point in time, as being end-user development or technical development, and to what degree. In adopting a platform-driven view, one would have to acknowledge that end-user development is part and parcel of any creative programming endeavour, and practiced routinely alongside technical development in professional or other settings, since making use of the abstractions a platform offers, is to perform end-user development on it. As these abstractions are used in the service of solving more technical problems, so does the development task become of a technical nature, possibly leading to a new layer of abstraction, and the cycle repeats.

A platform-driven view can help avoid the contradiction that comes up in our motivating case of clinical psychologists writing Ambulatory Assessment (AA) programs, where the twofold intent these researchers pursue in writing and distributing their data-collection programs can have their work classified as either opposite, i.e., end-user development or professional development. In a platform-drive view, when the encode tasks of their own problem domain as programs, expressed in familiar terms by way of purposely-built tools, or software components, the perform end-user development, but in cases when deeper layers of system functionality needs to be accessed to implement constructs of the higher-level domain, development becomes more technical in nature.

For example, Batalas et al. [3] offer a set of components written in HTML5, which render user interfaces for data input when invoked. These components constitute a layer built on top of the syntactic structural elements of a web-page as defined by the W3C standard (e.g., div, span, p), and provide a way to write a web-application for data collection using terminology of the researcher's domain instead. By using these components, the researchers perform end-user programming, but they would have to do more technical work if they wanted to produce new interfaces at the same semantic level, since they would have to have knowledge of the underlying layer, which is the web-browser with its Document Object Model, Document Flow, Cascading Style Sheets and JavaScript APIs.

8 Conclusion

The roles of the end-user and the developer are largely inventions of the platform being used each time. In other words, it is not only the case that users, who put requirements forward, and developers, who design and write the code, shape software platforms. It also happens that through the conventions they employ, abstractions that they put forward and types of work they allow, software platforms also in effect bring into being the substance of what their end-users or those who develop on them do. As the platforms evolve through history, so does our understanding of who the end-user developers are and what they do, and so do the definitions of end-user development that researchers consider representative.

In this work however, we have avoided stating any particular wording for another definition of end-user development, and we regard this to be out of the

scope of the discussion presented here. Rather, we consider it more productive to submit the points made here to the consideration of the researchers in the field and hopefully enrich relevant discourse. These points include the legacy origins of viewing end-user development as opposite to professional development, the extent to which this view is representative of modern software-development configurations, and the possibility to instead account not for the identity or role of the developer, nor for their intent in developing a piece of software, but for the nature of the task (i.e. how technical it is) when performed on a given platform.

Increasingly, software development takes place on such multiple layers of abstraction, with platforms and tools for the construction of software already delivered to the developers, that an end-user development aspect is always involved. For this reason, we propose that a platform-driven view of end-user development, inclusive of all types of developers as beneficiaries of EUD's findings, could better reflect this state of things and, in this manner, better anticipate the future.

References

1. Adams, C.W.: Small problems on large computers. In: Proceedings of the 1952 ACM National Meeting (Pittsburgh), pp. 99–102 (1952)
2. Barricelli, B.R., Cassano, F., Fogli, D., Piccinno, A.: End-user development, end-user programming and end-user software engineering: a systematic mapping study. J. Syst. Softw. **149**, 101–137 (2019)
3. Batalas, N., Khan, V.J., Franzen, M., Markopoulos, P., aan het Rot, M.: Formal representation of ambulatory assessment protocols in html5 for human readability and computer execution. Behav. Res. Methods **51**(6), 2761–2776 (2019). https://doi.org/10.3758/s13428-018-1148-y
4. Benson, D.H.: A field study of end user computing: findings and issues. Mis Quarterly, pp. 35–45 (1983)
5. Bogost, I., Montfort, N.: New media as material constraint: an introduction to platform studies. In: Electronic Techtonics: Thinking at the Interface. Proceedings of the First International HASTAC Conference, pp. 176–193 (2007)
6. Bogost, I., Montfort, N.: Platform studies: frequently questioned answers. Digital Arts Culture **2009** (2009)
7. Bourque, P., Fairley, R.E. (eds.): SWEBOK: Guide to the Software Engineering Body of Knowledge. IEEE Computer Society, Los Alamitos, CA, version 3.0 edn. (2014). http://www.swebok.org/
8. Burnett, M., Cook, C., Rothermel, G.: End-user software engineering. Commun. ACM **47**(9), 53–58 (2004). https://doi.org/10.1145/1015864.1015889
9. Choi, B.H.: Software as a profession. Harvard J. Law Technol. **33** (2020)
10. Clearver, H.: Industrialism or capitalism? conviviality or self-valorization? (1987). https://la.utexas.edu/users/hcleaver/hmconillich.html
11. Committee, C.E.U.F., et al.: Codasyl end user facilities committee status report (1979)
12. Conner, T.S., Mehl, M.R.: Ambulatory assessment: Methods for studying everyday life. Emerging Trends in the Social and Behavioral Sciences: An Interdisciplinary, Searchable, and Linkable Resource (2015)

13. Cotterman, W.W., Kumar, K.: User cube: a taxonomy of end users. Commun. ACM **32**(11), 1313–1320 (1989)
14. Cypher, A., Halbert, D.C.: Watch What I Do: Programming by Demonstration. MIT Press, Cambridge (1993)
15. Denning, P.J.: Computing the profession. In: Computer Science Education in the 21st Century, pp. 27–46. Springer (2000)
16. Dent, M., Bourgeault, I.L., Denis, J.L., Kuhlmann, E.: The Routledge Companion to the Professions and Professionalism. Routledge (2016)
17. Elliott, P.R.C.: The sociology of the professions. Macmillan International Higher Education (1972)
18. Ensmenger, N.L.: The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise. MIT Press, Cambridge (2012)
19. Fischer, G.: End-user development and meta-design: foundations for cultures of participation. In: Pipek, V., Rosson, M.B., de Ruyter, B., Wulf, V. (eds.) IS-EUD 2009. LNCS, vol. 5435, pp. 3–14. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00427-8_1
20. Fischer, G., Girgensohn, A.: End-user modifiability in design environments. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 183–192 (1990)
21. Floridi, L., Fresco, N., Primiero, G.: On malfunctioning software. Synthese **192**(4), 1199–1220 (2015)
22. Froehlich, J., Chen, M.Y., Consolvo, S., Harrison, B., Landay, J.A.: MyExperience: a system for in situ tracing and capturing of user feedback on mobile phones. In: MobiSys'07: Proceedings of the 5th International Conference on Mobile Systems, Applications and Services, pp. 57–70. ACM (2007). https://doi.org/10.1145/1247660.1247670
23. Greenwood, E.: Attributes of a profession. Social work, pp. 45–55 (1957)
24. Hecker, D.E.: Occupational employment projections to 2012. Monthly Lab. Rev. **127**, 80 (2004)
25. Huq, A.Z.: Racial equity in algorithmic criminal justice. Duke LJ **68**, 1043 (2018)
26. Illich, I., Lang, A.: Tools for conviviality (1973)
27. Johnston, P., Harris, R.: The boeing 737 max saga: lessons for software organizations. Softw. Quality Prof. **21**(3), 4–12 (2019)
28. Knuth, D.E.: The Art of Computer Programming, vol. 1. Addison-Wesley, Massachusetts (1973)
29. Ko, A.J., et al.: The state of the art in end-user software engineering. ACM Comput. Surv. **43**(3), 1–44 (2011). https://doi.org/10.1145/1922649.1922658. http://portal.acm.org/citation.cfm?doid=1922649.1922658
30. Krasner, H.: The cost of poor quality software in the us: A 2018 report. Consortium for IT Software Quality, Tech. Rep 10 (2018)
31. Lieberman, H.: Your Wish is my Command: Programming by Example. Morgan Kaufmann, San Francisco (2001)
32. Lieberman, H., Paternò, F., Klann, M., Wulf, V.: End-user development: an emerging paradigm. In: Lieberman, H., et al. (eds.) End User Development, pp. 1–8. Springer, Dordrecht (2006). https://doi.org/10.1007/1-4020-5386-X_1
33. Liskov, B., Zilles, S.: Programming with abstract data types. ACM Sigplan Notices **9**(4), 50–59 (1974)
34. Mackay, W.E.: Users and customizable software: a co-adaptive phenomenon. Ph.D. thesis, Citeseer (1990)
35. McConnell, S.: Code Complete, 2nd edn. Microsoft Press (2004). http://portal.acm.org/citation.cfm?id=1096143

36. McLean, E.R.: End users as application developers. MIS quarterly, pp. 37–46 (1979)
37. Mills, B.I.: Theoretical Introduction to Programming. Springer Science & Business Media, New York (2005)
38. Muffatto, M.: Open source: a multidisciplinary approach, vol. 10. World Scientific (2006)
39. Muzio, D., Ackroyd, S., Chanlat, J.-F.: Introduction: lawyers, doctors and business consultants. In: Muzio, D., Ackroyd, S., Chanlat, J.-F. (eds.) Redirections in the Study of Expert Labour, pp. 1–28. Palgrave Macmillan UK, London (2008). https://doi.org/10.1057/9780230592827_1
40. Myers, B.A., Ko, A.J., Burnett, M.M.: Invited research overview: end-user programming. In: CHI'06 Extended Abstracts on Human Factors in Computing Systems, pp. 75–80 (2006)
41. Nardi, B.A.: A Small Matter of Programming: Perspectives on End User Programming. The MIT Press, Cambridge (1993)
42. Noyes, J., Baber, C.: User-Centred Design of Systems. Springer Science & Business Media, New York (1999)
43. Pasquale, F.: Secret algorithms threaten the rule of law (2018)
44. Paternò, F.: End user development: survey of an emerging field for empowering people. ISRN Softw. Eng. **2013**, 11 (2013)
45. Plusch, S.P.: The evolution from data processing to information resource management. Technical report, ARMY WAR COLL CARLISLE BARRACKS PA (1984)
46. Repenning, A., Ioannidou, A.: What makes end-user development tick? 13 design guidelines. In: Lieberman H., et al. (eds.) End User Development. Human-Computer Interaction Series, vol. 9, pp. 51–85. Springer, Dordrecht (2006)
47. Ritchie, D.M., Thompson, K.: The unix time-sharing system. Bell Syst. Tech. J. **57**(6), 1905–1929 (1978)
48. Rockart, J.F., Flannery, L.S.: The management of end user computing. Commun. ACM **26**(10), 776–784 (1983)
49. Rough, D., Quigley, A.: Jeeves-a visual programming environment for mobile experience sampling. In: 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 121–129. IEEE (2015)
50. Ruiz Ben, E.: Defining expertise in software development while doing gender. Gender, Work Organ. **14**(4), 312–332 (2007)
51. Segal, J.: Professional end user developers and software development knowledge. Department of Computing, Open University, Milton Keynes, MK7 6AA, UK, Tech. Rep (2004)
52. Shannon, C.E.: A symbolic analysis of relay and switching circuits. Electr. Eng. **57**(12), 713–723 (1938)
53. Snyder, T.D., Tucker, P., Stone, A.: Digest of education statistics. National Center for Education Statistics (2002)
54. Thayer, K., Ko, A.J.: Barriers faced by coding bootcamp students. In: Proceedings of the 2017 ACM Conference on International Computing Education Research, pp. 245–253 (2017)
55. US Bureau of Labor Statistics: Occupational Outlook Handbook 2004–05 edition. Bureau of Labor Statistics (2004)
56. Weinberg, G.M.: The psychology of computer programming; 1971. von Nostrand Reinhold, New York (1998)