Available online at www.elixirpublishers.com (Elixir International Journal)

Computer Science and Engineering

Elixir Comp. Sci. & Engg. 69 (2014) 23057-23061

ISSN: 2229-712X

End User Software Engineering Importance and Related Techniques

Shahnaz Baghbani
ICT Department, ACECR Institute of Higher Education, Isfahan, Iran.

ARTICLE INFO

Article history:
Received: 15 February 2014;
Received in revised form:
29 March 2014;
Accepted: 7 April 2014;

Keywords
End user,
Software,
Debugging.

ABSTRACT

Today there are more end user programmers than professional programmers. since there is ample evidence that programs which end users create have led to huge expenses, software quality is necessary. In this paper End User Software Engineering (EUSE) and related terminology will be defined, then compared with professional software engineering. Finally to aid end user programmers for improving their software quality, some tools will be introduced.

Introduction

End-user programming enables end users to create their own programs. To this aim, Researchers and developers have been working on empowering end users to do this for a number of years, and they have succeeded. As a result, todays, end users create numerous programs.

The "programming environments" used by end users include spreadsheet systems, web authoring tools, and graphical languages for creating educational simulations. Using these systems, end users create programs in some forms such as spreadsheets, dynamic web applications, and educational simulations. Some ways in which end users create these programs include writing and editing formulas, dragging and dropping objects onto a logical workspace, connecting objects in a diagram, or demonstrating intended logic to the system[1].

According to statistics from the U.S. Bureau of Labor and Statistics, by2012 in the United States there will be fewer than 3 million professional programmers, but more than 55 million people using spreadsheets and databases at work, many writing formulas and queries to support their job [2]. There are also millions of designing websites with Javascript, writing simulations in MATLAB [3], prototyping user interfaces in Flash [4], and using countless other platforms to support their work and hobbies. Computer programming, almost as much as computer use, is becoming a widespread, pervasive practice.

What makes these "end-user programmers" different from their professional counterparts is their goals: professionals are paid to ship and maintain software over time; end users, in contrast, write programs to support some goal in their own domains of expertise. End-user programmers might be secretaries, accountants, children [5], teachers [6], interaction designers [Myers et al.2008], scientists [7] or anyone else who finds themselves writing programs to support their work or hobbies. Programming experience is an independent concern. For example, despite their considerable programming skills, many system administrators view programming as only a means to keeping a network and other services online [8]. The same is true of many research scientists [9].

Despite their differences in priorities from professional developers, end-user programmers face many of the same software engineering challenges. For example they must choose which APIs, libraries, and functions to use [18]. Because their programs contain errors [11], they test, verify and debug their programs. They also face critical consequences to failure. For example, a Texas oil firm lost millions of dollars in an acquisition deal through an error in a spreadsheet formula [12]. The consequences are not just financial. Web applications created by small-business owners to promote their businesses do just the opposite if they contain bad links or pages that are displayed incorrectly, resulting in loss of revenue and credibility [13]. Software resources configured by end users to monitor non-critical medical conditions can cause unnecessary pain or discomfort for users who rely on them [14].

Because of these quality issues, researchers have begun to study end-user programming practices and invent new kinds of technologies that collaborate with end users to improve software quality. This research area is called end-user software engineering (EUSE). The topic is distinct from related topics in end-user development in its focus on software quality. It introduces tools that aid end users to improve their software quality.

Definition

One contribution of this article is to identify existing terms in EUSE research. This section, will be start with a basic definition of programming and end up with a definition of end-user software engineering.

Programming and Programs

Programming similarly to modern English dictionaries is defined, as the process of planning or writing a program. This leads to the need for a definition of the termprogram. KO et al. define a program as "a collection of specifications that may take variable inputs, and that can be executed (or interpreted) by a device with computational capabilities."[15] Note that the variability of input values requires that the program has the ability to execute on future values, which is one way it is different from simply doing a computation once manually. This definition captures general purpose languages in wide use, such as Java and C, but also notations as simple as VCR programs, written to record a particular show when the time of day (input) satisfies the specified constraint, and combinations of HTML and CSS, which are interpreted to produce a specific visual rendering of shapes and text. It also captures the

Tele:
E-mail addresses: Baghbani@jdeihe.ac.ir

use of report generators, which take some abstract specification of the desired report and automatically create the finished report.

End-User Programming

To fully define this term it is important firstly to understand the definition of an end-user. To be concise, an end-user is anyone who uses a computer [15]. When we refer to end-user programming it is, in general, programming done by anyone using a computer. This however, is not a very descriptive definition. KO et al. define end user programming as "programming to achieve the result of a program primarily for personal, rather [than] public use" [15].

The important distinction here is that program itself is not primarily intended for use by a large number of users with varying needs. For example, a teacher may write a grades spreadsheet to track students' test scores, a photographer might write a Photoshop script to apply the same filters to a hundred photos, or a caretaker might write a script to help a person with cognitive disabilities be more independent [16]. In these end-user programming situations, the program is a means to an end and only one of potentially many tools that could be used to accomplish a goal. This definition also includes a skilled software developer writing "helper" code to support some primary task. For example, a developer is engaging in end-user programming when writing code to visualize a data structure to help diagnose a bug. Here, the tool and its output are intended to support the developers' particular task, but not a broader group of users or use cases.

In contrast to end-user programming, professional programming has the goal of producing code for others to use. The intent might be to make money, or to write it for fun, or perhaps as a public service (as is the case for many free and open source projects). Therefore, the moment novice web designers move from designing a web page for themselves to designing a web page for someone else the nature of their activity has changed. The same is true if the developer mentioned above decides to share the data structure visualization tool with the rest of his team. The moment this shift in intent occurs, the developer must plan and design for a broader range of possible uses, increasing the importance of design and testing, and the prevalence of potential bugs.

It is also important to clarify two aspects of this "intent"-based definition. First, our definition is not intended to be dichotomous, but continuous. After all, there is no clear distinction between a program intended for use by five people and a programintended for fifty. Instead, the key distinction is that as the number of intended uses of the program increases, a programmer will have to increasingly consider software engineering concerns in order to satisfy increasingly complex and diverse constraints. Second, even if a programmer does not intend for a program to be used by others, circumstances may change: the program may have broader value, and the code which was originally untested, hacked together, and full of unexercised bugs may suddenly require more rigorous software engineering attention.

End-User Software Engineering

With definitions of programming and end-user programming, we now turn to the central topic of this article, end-user software engineering. As we discussed in the previous section, the intent behind programming is what distinguishes end-user programming from other activities. This is because programmers' intents determine to what extent they consider concerns such as reliability, reuse, and maintainability and the extent to which they engage in activities that reinforce these qualities, such as testing, verification, and debugging. Therefore, if one defines software engineering as systematic and disciplined activities that address software quality issues, the key difference between professional software engineering and end-user software engineering is the amount attention given to software quality concerns.

In professional software engineering, the amount of attention is much greater: if a program is intended for use by millions of users, all with varying concerns and unique contexts of use, a programmer must consider quality regularly and rigorously in order to succeed. This is perhaps why definitions of software engineering often imply rigor. For example, IEEE Standard 610.12 defines software engineering as "the application of systematic, disciplined, quantifiable approaches to the development, operation, and maintenance of software." Systematicity, discipline, and quantification all require significant time and attention, so much so that professional software developers spend more time testing and maintaining code than developing it [17] and they often structure their teams, communication, and tools around performing these activities [18].

In contrast, end-user software engineering still involves systematic and disciplined activities that address software quality issues, but these activities are secondary to the goal that the program is helping to achieve. Because of this difference in priorities and because of the opportunistic nature end-user programming [19], people who are engaging in end-user programming rarely have the time or interest in systematic and disciplined software engineering activities.

Given these differences, the challenge of end-user software engineering research is to find ways to incorporate software engineering activities into users' existing workflow, without requiring people to substantially change the nature of their work or their priorities. For example, rather than expecting spreadsheet users to incorporate a testing phase into their programming efforts, tools can simplify the tracking of successful and failing inputs incrementally, providing feedback about software quality as the user edits the spreadsheet program.

Comparison to Professional Software Engineering

As discussed in the previous section the difference between End-User Software Engineering and Professional SE is the intent behind the programming being done. In EUSE the intent of the programming is to achieve a personal goal (e.g. writing macros to automate repetitive tasks). In Professional SE the intent of the programming is to achieve a public goal (non-personal). This difference in intent is the cause of the key difference between professional and end-user software engineering, "the amount of attention given to software quality concerns" [15].

The differences in intent and attention to quality are not the only differences between these two activities. Therefore, to fully understand EUSE it is important to discuss these differences. Examining the reasoning that drives the actions taken during different phases of the software development lifecycle is the best way to highlight the major differences and similarities. The areas that will be examined are: requirements gathering, design and specification, testing and verification, debugging, and maintenance.

Requirements Gathering

If you ask any Professional Software Engineer they will quickly tell you exactly why gathering requirements is crucial to the success of the program. If the requirements are not thorough it is almost impossible to build not only the right program, but to build the program right. Therefore, tons of time is spent in the requirements gathering phase before even beginning any programming, usually. However, for end-user software engineering, having formal requirements, or requirements in general, is often seen as unnecessary. This difference between professional and end-user software engineering is due to the difference in the source of the requirements [15].

For professional software engineers the requirements come from the customers and users of the program, generally not the engineers themselves. This is not true for end-user programmers. End-user programmers generally program for themselves, a friend, or a colleague [15]. It is this difference that leads to the difference in importance placed on requirements gathering. The end-user is the customer/user and therefore is programming to achieve a personal goal, not programming to fulfill someone else's. Therefore, "[f]or end users, the requirements are both more easily understood (because they are their own) and more likely to change" [15].

Design and Specification

The next area of development we will examine is design and design specification. In software engineering, the purpose of design specifications is to specify the systems internal behavior [15]. The design specifications are used to lay out the implementation strategy for ensuring that the system meets all of the requirements. This is done by assigning appropriate priorities to each requirement so that the highest priority requirements are taken care of before the low priority ones.

In end-user programming, it is often a struggle for the end-user to translate their requirements into a working program [15]. This is due to the fact that end-user programmers generally don't have training or experience in design and therefore see little to no benefit to it [15]. However, instead of design specifications, end-user programmers' often come to realize what constraints exist on their programs' implementations through the process of writing their program [20].

Testing and Verification

In professional software engineering testing is an essential part of the software development lifecycle. Testing is the way in which software engineers ensure the proper execution of the program. This is done through the use of many different testing techniques such as JUnit testing, regression testing, embedded test cases, etc. It is by running these tests that software engineers are able to identify bugs and properly debug the code, which will be discussed in the next section.

The primary difference between EUSE and professional SE is that the priorities' of the end-user programmer frequently lead to overconfidence in their programs correctness [15]. It is known that professional programmers are overconfident [24][26][23], but as they gain experience this overconfidence subsides [22]. In comparison, many studies about spreadsheets report that in spite of high error rates in spreadsheets, the developers of the spreadsheets are carelessly confident about its correctness [25][21].

Debugging

In professional software engineering, debugging is an essential activity to ensure that the programs requirements are being met. Debugging is different from testing and verification in that instead of being used for the detection of errors, debugging is the means by which errors are found and removed from the program [15]. This activity is one of the most time consuming activities undergone to ensure that the program meets all of the requirements. This is because debugging requires that the programmer has an excellent understanding of the program, and is able to identify areas that could have caused the problem.

The primary difference between end-user programmers and professional software engineers is that, unlike professional SE, end-user programmers often lack accurate knowledge and understanding about their programs execution [15]. Because of this, it is very hard for many end-user programmers to even conceive what the root-cause could be, and even harder for them to actually be able to remedy the bug. Furthermore, because end users frequently prioritize their external goals over software reliability, they often rely on debugging strategies such as making code changes until it appears to work as expected [15]. This approach often leads to the introduction of additional errors in the code and the original functionality can be lost.

Popular Tools that aid in End-User Software Engineering

There are many tools available whose main purpose is integrating software quality principles used by professional software engineers into end-user software engineering. Because the vast majority of end-user programming is done through the creation of spreadsheets, the majority of the tools below are aimed at debugging in spreadsheets. The tools that have presented in EUSES Consortium (http://eusesconsortium.org) will bediscussed as follow.

- **Topes**

It is a model and supporting system to support validation and reuse of short, human-readable data in end-user programmers' programs. Users create a "tope" to describe rules for recognizing and reformatting a certain kind of data, such as phone numbers, and then associate a tope with spreadsheet cells, web form textfields, web macro variables, or other fields. Values are automatically checked and transformed at runtime. Studies show that end-user programmers can validate data more quickly and accurately than with existing tools.

- **WYSIWYT: TheWhatYou SeeIs WhatYou Test**

It helps users test their spreadsheets while they're creating them. As a user develops a spreadsheet, he or she can also test that spreadsheet incrementally yet systematically. At any point in the process of developing the spreadsheet, the user can validate any value that he or she notices is correct.

- **Whyline**

Whyline is a debugging tool that allows programmers to ask "Why did" and "Why didn't" questions about their program's output. Programmers choose from a set of questions generated automatically via static and dynamic analyses, and the tool provides answers in terms of the runtime events that caused or prevented the desired output. In user studies of the Whyline Alice programming, programmers using the Whyline to debug spent a factor of 8 less time debugging the same bugs than programmers without the Whyline.

- **Gencel**

Observing that all errors in spreadsheets result from updates or changes applied to the spreadsheet - be it during the creation of a new spreadsheet or while adapting an existing one - an obvious alternative to debugging is to prevent errors by making these update operations safe. An extension to Excel, called Gencel, that is based on the concept of a spreadsheet template, which captures the essential structure of a spreadsheet and all of its future evolutions. Such a template ensures that the spreadsheet can be changed only in the anticipated ways, so that spreadsheets evolving from templates will provably never contain any reference, range, or type errors. Gencel can help to reduce maintenance costs while at the same time it dramatically increases the level of correctness and reliability of spreadsheets.

- **Citrus**

Graphical structured editors for code and data have many benefits over editing raw XML, but they can be difficult and time-consuming to build using modern programming languages. Citrus is a new object-oriented, interpreted language that is designed to simplify the creation of such editors, by providing first-class language support for one-way constraints, custom events and event handlers, and value restrictions and validation.

- **Barista**

Barista is a new implementation framework, implemented in Citrus, which enables the creation of a new class of highly visual, highly interactive code editors. Editors built with Barista can offer standard features such as conventional text-editing interaction techniques, immediate feedback about errors and code-completion menus. However, Barista editors can also support drag and drop interaction techniques, new types of embedded tools, and alternative views of code.

- **Crystal**

It is an application framework (written in Java and using the Swing toolkit) that extends the work of the Whyline, enabling the creation of software applications that allow users to ask questions about their data and the application's state.

- **Robofox**

Robofox is a web browser extension that enables the automation of repetitive browsing tasks such as extracting information from a web site, integrating data from different web sites, and customizing the appearance of the collected information.

- **Slate**

Many spreadsheet systems allow users to specify units with their data in order to help users detect errors. Slate allows users to specify the object of measurement , in addition. By intelligently propagating labels representing these objects, Slate helps users identify errors in their spreadsheets that other spreadsheet systems can't.

- **WebAppSleuth**

Web applications are increasingly prominent in society, serving a wide variety of user needs. Engineers seeking to enhance, test, and maintain these applications must be able to understand and characterize their interfaces. Third-party programmers (professional or end user) wishing to incorporate the data provided by such services into their own applications would also benefit from such characterization when the target site does not provide adequate programmatic interfaces.

Conclusion

As the number of end user programmers today is rapidly increasing and many programs are written by them, the quality of this software is important. End User Software Engineering focuses on developing the software without error, in this way there are some tools which end users can use to solve their software quality problems. Because the vast majority of end user programming is done through the creation of spreadsheets, the majority of tools are aimed at debugging in spreadsheets.

References:

1.BURNETT, M. 2009 What Is End User Software Engineering And Why Does It Matter?IS-EUD '09 Proceedings of the 2nd International Symposium on End-User DevelopmentPages 15 - 28

2.SCAFFIDI, C., MYERS, B. A., AND SHAW, M. 2008. Topes: Reusable abstractions for validating data. In Proceedings of the International Conference on Software Engineering.

3.GULLEY, N. 2006. Improving the quality of contributed software on the MATLAB file exchange. In Proceedings of the 2nd Workshop on End-User Software Engineering, in conjunction with the ACM Conference on Human Factors in Computing.

4.MYERS, B., PARK, S., NAKANO, Y., MUELLER, G., AND KO. A. J. 2008. How designers design and program interactive behaviors. In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing.177–184.

5.PETRE, M. AND BLACKWELL, A. F. 2007. Children as unwitting end-user programmers. In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing. 239–242.

6.WIEDENBECK, S. AND ENGEBRETSON, A. 2004. Comprehension strategies of end-user programmers in an event- driven application. In Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing. 207–214.

7.SEGAL, J. 2005. When software engineers met research scientists: A case study. Empir. Softw. Eng. 10, 517–536.

8. BARRETT, R., KANDOGAN, E., MAGLIO, P. P., HABER, E. M., TAKAYAMA, L. A., AND PRABAKER, M. 2004. Field studies of computer system administrators: analysis of system management tools and practices. In Proceedings of the ACM Conference on Computer Supported Cooperative Work. 388–395.

9.CARVER, J., KENDALL, R., SQUIRES, S., AND POST, D. 2007. Software engineering environments for scientific and engineering software: a series of case studies. In Proceedings of the International Conference on Software Engineering. 550–559.

10.KO, A. J. AND MYERS, B. A. 2004. Designing the Whyline: A debugging interface for asking questions about program failures. In Proceedings of the ACM Conference on Human Factors in Computing Systems.151–158.

11.PANKO, R. 1998. What we know about spreadsheet errors. J. End User Comput. 2, 15–21.

12.PANKO, R. 1995. Finding spreadsheet errors: Most spreadsheet models have design flaws that may lead to long-term miscalculation. Information Week, May, 100. Environments. 123–130.

13.ROSSON, M. B., BALLIN, J., AND RODE, J. 2005. Who, what, and how: A survey of informal and professional web developers. In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing.199–206.

14.ORRICK, E. 2006. Electronic medical records–Building encounter forms. In Proceedings of the 2nd Workshop on End-User Software Engineering, in conjunction with the ACM Conference on Human Factors in Computing.

15.Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., et al. (2011). The state of the art in end-user software engineering. ACM Computing Surveys (CSUR), 43(3), 21.

16.CARMIEN, S. P. AND FISCHER, G. 2008. Design, adoption, and assessment of a socio-technical environment supporting independence for persons with cognitive disabilities. In Proceedings of the ACM Conference on Human Factors in Computing Systems. 597–606.

17.KO, A. J. DELINE, R., AND VENOLIA, G. 2007. Information needs in collocated software development teams. In Proceedings of the International Conference on Software Engineering. 344–353.

18.TASSEY, G. 2002. The economic impacts of inadequate infrastructure for software testing. RTI Project Number 7007.011, National Institute of Standards and Technology.

19.BRANDT, J., GUO, P., LEWENSTEIN, J., AND KLEMMER, S. R. 2008. Opportunistic programming: How rapid ideation and prototyping occur in practice. In Proceedings of the Workshop on End-User Software Engineering (WEUSE).

20.Fischer, G., &Giaccardi, E. (2006). Meta-design: A framework for the future of end-user development. End User Development Empowering People to Flexibily Employ Advanced Information and Communication Technology, 427–457.

21.Hendry, D. G., & Green, T. R. G. (1994). Creating, comprehending, and explaining spreadsheets: A cognitive interpretation of what discretionary users think of the spreadsheet model. International Journal of Human Computer Studies, 40(6), 1033–1066.

22.Ko, A. J., DeLine, R., &Venolia, G. (2007). Information needs in collocated software development teams. Proceedings of the International Conference on Software Engineering (pp. 344–353).

23.Lawrence, J., Clarke, S., Burnett, M., &Rothermel, G. (2005). How well do professional developers test with code coverage visualizations? An empirical study. Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (pp. 53–60).

24.Leventhal, L. M., Teasley, B. E., &Rohlman, D. S. (1994). Analyses of factors related to positive test bias in software testing. International Journal of Human-Computer Studies, 41(5), 717–749.

25.Panko, R. (1998). What we know about spreadsheet errors. Journal of Organizational and End User Computing (JOEUC), 10(2), 15–21.

26.Teasley, B., &Leventhal, L. (1994). Why software testing is sometimes ineffective: Two applied studies of positive test strategy. Journal of Applied Psychology, 79(1), 142.