Development of application software without programming

J Hamzabegovic
University of Bihac, Faculty of Technical Engineering Bihac, Irfana Ljubijankica str., 77000 Bihac, Bosnia and Herzegovina

E-mail: jasna.hamzabegovic@unbi.ba

Abstract. End-user programming is an increasingly popular discipline of software engineering. In order to design a development environment that allows programming by non-programmers, it is desirable to understand their mental model. Most languages designed for end-users are based on visual interaction techniques such as programming with graphically presented rules and agents. Visual techniques allow direct manipulation in the environment in such a way that users grasp and pull visual components to match the application, as well as to select specific actions on certain events. Due to its omnipresence in operating systems, desktop and mobile applications, direct manipulation is an interaction technique which non-programmers are familiar with, and therefore highly desirable in application development environments by end-users or non-programmers.

In this paper we present a review of the features and the functionality of the development environment that would allow the end-users logopedists to develop their therapeutic software without writing programming code and without any programming knowledge.

1. Introduction
In the last decade, the interaction between humans and computers has evolved from a system of easy-to-use to a simple development system. For many years, the development of new ones and the modification of existing application programs have been the privilege of educated programmers. However, today professional developers can not meet the needs of all end-users due to the shortage of this profession in the global market.

There were 1.3 million software jobs open last year, according to Trilogy Education Services. Code.org calculates 512,720 open computing jobs now, and the U.S. Bureau of Labor Statistics predicts that in 2020 there will be 1.4 million more software development jobs than applicants who can fill them [1].

2. End-user programming
In order to solve the problem of software developer deficiency, software development by the end-user (EUD) is increasingly encouraged. The EUD is a "set of methods, techniques and tools that enable program system users, acting as unprofessional developers, to create, modify or extend application software for their own needs" [2]. In particular, the EUD allows end-users to design or customize the user interface and functionality of their own software. This is very important because end-users know

---

1 End-User Development

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd.
their own context and needs better than anyone else and have timely information about changes and new developments in their area. Through EUD end users can set up software to suit their requirements better than would be possible without EUD.

End-users do not undergo training in programming languages, development processes, or modeling, and diagnostic notation passed by professionals. They do not have time or motivation to learn traditional techniques, because end users usually want to write code to achieve an ad-hoc short-term goal - to ease the solution of the current problem. Accordingly, the EUD supports and responds to requests by offering appropriate tools, structures and development processes that are useful, and quickly learned and easily integrated into existing practice.

The EUD overlaps two similar concepts: end-user programming (EUP) and end-user software engineering (EUSE). End-user programming allows end users to create their own programs [3]. The difference between the EUSE and the EUP is that EUSE methods, techniques and tools provide the entire life cycle of software development, including modifications and extensions of the software, and not just the "create phase". End-user software engineering is a relatively new sub-domain of the EUD, which is being developed to reduce errors that end users make in developing their own software. As the focus on quality creation, quality modification and expansion of software created by the end user, the subject of software engineering research are methods, techniques and tools that enable this quality.

3. Psychology of programming
Programming is a demanding mental action that involves designing detailed instructions for the computer to solve a problem. Therefore, understanding the intellectual background as a human skill in the field of computer programming is of great importance. This knowledge is of great benefit in designing both programming languages, as well as human-computer interfaces, and other user-friendly software systems.

Cognitive science is defined as a scientific discipline dealing with the study of human mind and intelligence [4]. It is interdisciplinary because it includes: psychology, psychiatry, linguistics, anthropology, computing and biology. The name "cognitive science" was first used by Christopher Longuet-Higgins in his commentary from 1973 [5], and in what is said in his report by Lighthill [6], who was researching the current state of artificial intelligence. The term "cognitive" in the title is "used for any type of mental operation or structure that can be studied in precise terms" [7].

Cognitive science deals with the research of the nature of mental processes such as: perception, memory, attention, locking, language, motor control and problem solving. Its goal is to understand the concepts and processes that take place in the human mind: how they form, how they evolve and how they spend, or understand how the human mind works. It is precisely this knowledge from cognitive science that is very important in computing in the construction of programming languages. The cognitive science branch dealing with psychological aspects of programming is called psychology of programming. The psychology of programming (PoP) is an interdisciplinary field that includes research on the cognitive activities of the programmer (perception, recognition, reasoning and conclusion), about tools and methods for programming, and about teaching programming skills.

It is desirable to achieve programming performance such that the created program meets its specification, it is adaptable to the future and works effectively. If the psychological aspects of computer programming are understood, one may conclude how to achieve greater programming efficiency.

Cognitive psychologists and computer experts are concerned with the study of psychology of programming. The main motivation for computer experts is to improve existing tools and develop new ones, as well as the discovery of general principles that apply to people in the context of program tasks. Psychologists, however, are interested in new theories of human knowledge applied in other areas. For them programming, as a very complex job, provides good opportunities for studying cognitive processes at a high level in a complex environment. It is precisely the dual character of programming psychology that requires a good understanding of psychology, social sciences and program engineering from the PoP researchers.
Methods of research in the field of PoP are different. The most common methods of research in the field of cognitive psychology are controlled experiments performed in laboratories. In the field of social sciences, this is field research using the technique of qualitative analysis. In the field of program engineering, the problems of beginners in programming, as well as the problems in programming teaching are studied. A new source of research materials are various so-called. open source communities that share program code among themselves, and blogs and forums where developers participate, which are publicly available on the Internet.

4. Mental model
A mental model is the thought process of a person about how he / she works, i.e. understanding the surrounding world. The mental model is based on an incompletely established factual state, experience, but also on intuitive knowledge. It helps shape actions and behaviors, influence what people need to pay attention to in complicated situations, and define how people approach problems and how to solve them.

4.1. Mental model of the end-user programmer
For "end users developers", better development tools will be built if they know how they think. If the tool works in the way the user expects and if it is felt by the user, such a tool will be well accepted because it can be easily learned and easily used. Likewise, the tool can be designed to redefine the way the end-user develops the problem. In both cases it is important to know the mental model of the user.

The human mental model is shaped by his education and experience, and he develops as he learns further. The term "natural" or "naturalness" applies to software development technology that is referred to the mental model that the user possesses before learning the tool or programming language. It is very important to minimize the "mental distance" between the user's objectives and the language specification [8], [9].

4.2. Mental model of speech therapist
In the context of this work under the mental model is meant the way logos visualize therapeutic applications, and mental representations of entities and connections that make up the entire system. In order to understand the mental model of speech therapists, the results of two studies were used.

The first study as a starting point uses methods from Pane, Ratanamahatan and Myers (2001), who dealt with the question of naturalness in the context of a programming language for children, studying how children and adults use the natural language to solve program problems [10]. They used the results of these studies to design a programming environment that offers concepts close to the natural mental model of the end-user, the future developer. Following this model, a second study looked at the behavior of the target population of the speech logic in order to identify the control logic and specify the elements and actions they apply in the natural language in everyday practice, and then designed a new language and environment based on these abstractions. After this general approach, it was explored that logos describe the behavior of a future development environment that should enable them to create interactive and multimedia applications, and how these applications should look, i.e. what should they contain and what to enable. Answers to these questions were given by logos as the best connoisseurs of the mental model of children with dyslexia for whom applications would develop. The results of all these studies have been applied to the modeling and design of the desired development environment.

4.3. Researching the language and concepts of speech therapists, future programmers
The first efforts in researching the mental end-user model were focused on researching the language, concepts and ways in which logos create therapeutic exercises for children with dyslexia. They wanted to find out how they would specify and implement therapeutic applications, and what development techniques would be most natural for them. There was little concern about their lack of language skills for specifying the elements of the user interface (such as terms: toolbox, control, textbox, grid,
checkbox, listbox), as well as language and diagrams for specifying the behavior of the application. But these problems are through direct communication with them and a detailed explanation is successfully solved. There was also an interest in how users would describe the properties of created objects, their behavior, and actions as a result of certain events. For the needs of the research, a simple Windows application for the matching of elements, i.e., designing a future therapeutic application.

Participants received a general introduction to the research goals and were then asked to review and designate all the elements of the screen editor design (window elements and toolbars) that can be used to create workout exercises for children with dyslexia. They were presented with a way of adding objects and staging the scene. Labeling by logopeds meant the encircling of interface elements that would also use the precipitation of those elements they would not need. They were also suggested to be able to add new elements that were not foreseen by the existing proposal. This first phase of the research was carried out separately with each logoped in order to explain the functions of the offered tools.

The aim of this research was to obtain relevant information about elements of a future visual language that the target population will use to create therapeutic software interfaces. The results showed the great need for letters and words to be used with sound and/or image due to the visual perception inherent to dyslexics, which, in contrast to the rest of the population, think in the pictures. Participants also insisted that visual components are directly manipulated in a way that they capture and pull visual components when they interface the user interface of the application, and that they can select specific actions on certain events. Because of its omnipresence in mobile applications, direct manipulation is an interaction technique with which logos are introduced and therefore highly desirable in the development environment for the development of applications by the logopedic-programmer.

After this phase, during which they proposed elements that they consider natural and intuitive, logopeds were asked to show how the program should work. They received an empty sheet of paper where they needed to create a simple exercise from therapeutic practice, using elements from the design editor of the proposed development environment. They should have defined the behavior of the application in different possible situations and based on different requests. We wanted to see what end-users consider as a specification of behavior. They wrote down their requests and remarks on a white sheet of paper. Describing the behavior of the application by the participants helped to study and understand their natural mental model.

When describing the behavior of the application, participants are inclined to combine procedural steps and declarative statements. They use declarative statements to determine constraints on behavior (e.g., "all fields must be filled in"). Procedural statements often represent a test and result (for example, "if a devoted letter is not what is expected, this field will leave blank"). Of course, none of the participants in the behavior specifications mentions conventional program constructions such as: variable, conditional statement or loop, because they do not know the terminology and concepts. Where loops are needed (for example, when filling in the field, that is, all the letters of one word or all the words of one sentence), participants indicate an iteration, thinking that it is expected to be applied, i.e. repeat as much as needed. Of course, none of the participants were able to describe what is happening behind the app.

5. Visual programming
In computer science, the visual programming language (hereinafter referred to as "VPL") allows users to create programs by manipulating more graphic elements of the program than text files. VPL allows programming with visual expressions, spatial layout of text and graphic symbols that are used either as elements of syntax or secondary writing. Visual programming languages also make it possible to create a more natural programming environment that will be familiar and familiar to the user. In order to be used by non-programmers, visual programming languages must use intuitive metadata and specific to the area they are intended to, reduce the cognitive load of end-users in their learning and adoption, to enable the writing of a readable and understandable code, and they are simple and fun.
As scientists point out, in this type of programming more than one dimension is used to represent the semantics [11]. Examples of such additional dimensions are reflected in the use of multidimensional objects, the use of spatial relationships, and the time dimension by typing, so-called. "before-after" semantic relations. Any such potentially significant object or relationship is a sign, and the collection of one or more of them is an image expression. Examples of visual expressions used in visual programming include diagrams, idle sketches, icons, or demonstrations of actions using graphic objects. If in some programming language the syntax includes visual expressions, then this programming language is visual. Multidimensionality is the important difference between VPLs and strictly textual languages.

5.1. Visual programming and parametrization
Visual programming and parameterization tools allow the creation of a program logic by stacking and linking visual elements and their spatial layout instead of writing textual commands of the programming language [12]. Semantic and syntax rules, as well as textual programming languages, are still used, but the language elements in this case are represented by graphical symbols. Although these are tools for developing applications without programming, they still need to be familiar with the programming principles. With the "right" visual programming tools there is no need to write program commands or other texts, except in cases where it is necessary to designate program elements (variables, labels, blocks, etc.) for transparency purposes. Research has shown that visual programming and parameterization tools can be classified into four sub-categories: graphical user interface tools, high-level parameterization software programs, tools that use diagrams for describing data flows or process flows, and graphical representation of program logic tools. Typical representatives of the graphical editing tool for the user interface are MS Visual Studio and MS Expression Blend.

The basic advantage of the visual programming language is that it requires little initial knowledge in order to be able to start efficiently using it. This is because all the elements of the programming language, its semantics and syntax are visually represented, and the use of context dependencies dynamically narrows the choice of elements that can be used and in this way the user facilitates the programming process. All tools in this group are, in fact, software tools, and of the classic programming tools they differ only in that programming is not carried out by writing textual commands of the programming language, but by stacking graphic elements.

6. Component software development
It's the way software is developed with the help of ready, reusable components that appeared in the late 90s of the 20th century.

The concept of a component in the hardware industry has been present for a long time. Building a computer does not happen from scratch, but it folds out of the finished components. The use of finished components carries numerous advantages, such as: the ability to use proven correct components, faster computer construction, easy maintenance, low cost, and the inclusion of non-professionals in assembling the computer. That is precisely one of the reasons for the rapid progress of this branch of industry and the large supply of low-cost computers. A similar concept is also conceived for building software and can substantially help alleviate its deficit. Even if the components that were built were intuitive and close to non-programmers, experts from other areas, and if they were offered an easy way to compose them, the application software deficit would certainly be lower. Most people think about software components similar to hardware, thinking that program engineers compile them into software applications in the same way that hardware engineers compile a computer from hardware components. Various researchers attempted to define software components by keeping this attitude, which resulted in several differently descriptive informal models.

One of the most popular component definitions is provided by the ECOOP Working Group (European Conference on Object-Oriented Programming) which states that "... the software component is a unit of composition by contract of a particular interface and explicit contextual
dependence." The software component can be distributed independently and be a subject for a third-party composition. Others, however, require the extension of this definition, taking into account three different aspects. The aspect of the specification that should describe what component works and how it can be used; the aspect of the packaging perspective that views the component as a delivery unit; and the aspect of the integrity perspective that sees it as a kind of encapsulation. It follows that the component is a software package that offers its various services through interfaces. In accordance with the above, the definition of the component given by D'Souza and the authors. According to them: "The component is a coherent software package that can be developed independently and delivered as a unit, which offers interfaces that can be connected to other components and participate in building a larger system." [13]. It is an independent software package that offers some functionality. It's very important to define its interface well. It can be output through which component provides functionality for other components or input through which component gets services from other components.

All these definitions emphasize the tendency of the nature of the components in the form of a "black box" so that the developer can use them to create a larger system without any knowledge of how the components themselves are implemented. Component models allow program engineers to use components from different languages and platforms together. End users also benefit from this technology. Thus, for example, spreadsheets, word processing, drawing, and database applications often use a component model to incorporate edited data from one application into files created and managed by another application. Currently, the most popular commercial component models are: CORBA² from Object Management Group, COM³ from Microsoft, and JavaBeans from Sun Microsystems.

6.1. Features and benefits of component software development

Development based on software components represents the process of software development by simply stacking or installing already finished components. The basis of this development is a component model, which defines the components and mechanism of their composition. The idea is to facilitate software development by selecting and integrating existing software components into a previously defined software architecture [14]. Software components can be developed by different developers and using different languages and development platforms. With this approach to software development, the costs and time necessary for its appearance on the market are significantly reduced, while at the same time improving: maintenance, reliability and overall software quality [15].

As the development of component software becomes increasingly important in the IT world, numerous concepts and conventions for its construction have been introduced. A special discipline within software engineering that develops software reusable components is called component-based software engineering (CBSE). This discipline was due to the failure of the object-oriented paradigm to support the effective reuse of the code. Unlike class objects that are detailed and specific, the components are more abstract. In addition to the benefits that are reflected in the reuse of finished components, CBSE is based on the basic design principles of software engineering applied to the development of components, which imply that: components must be independent to each other, not to hinder one another, their implementation must be hidden and unmanageable the one who uses them, communication with them must take place through a well-defined and simple interface.

The life cycle and model of software in component-based software engineering differ from the life cycle and software models in traditional software engineering. These two items are exactly what focuses on component based software engineering.

7. Integrated Development Environment

The common elements of an application are: screen user interface, program logic, database management system, and reporting system. For the purpose of their realization, there are special

---

² Common Object Request Broker Architecture
³ Component Object Model
software tools for the creation, correction and testing of these elements, and sometimes they are all integrated into the so-called integrated development environment - IDE.

Modern integrated development environments contain a multitude of visual aids that facilitate the creation of a screen user interface, modeling and linking to a database or making complex reports. However, any more demanding program logic that forms the backbone of the application must be written in the selected programming language. In this part there are tools for development of applications without programming that are aimed at maximizing the programming process, and in some cases completely avoiding it.

7.1. Development environment for component software development

The development environment is very important for the development of software based on components. Not only can the application be added through this component environment, but it can be used to correct errors and test applications. In addition, the efficiency and usability of the development environment is a key factor that affects the easier development of this type of software. In such a visual programming environment, the application assembly process is visualized, and only error correction and interactive testing.

7.2. Development environment for logopeds-developers

When building most of the most popular visual development environments, the greatest attention is paid to the editor in order to make it easier to visualize and build a user interface. In this paper, a development environment for speech logoses is proposed, in which the centralized place is occupied by the visual WYSIWYG editor. Components for constructing application programs by logopeds must be intuitive and meaningful units, minimally dependent on each other so that the change of one component in the system minimally affects other components in the system. The component is usable, i.e. after being designed, implemented and tested, it can be used whenever necessary. Its internal work is hidden and isolated from the interface so that logos can build applications using components without any knowledge of how they work. Components should be reliable because they are the prerequisite for the development of high quality and reliable component-based software. They must be interoperable, i.e. support multiple platforms, executed by anyone, without the need for the source code to be available, and able to describe their public interfaces, i.e. flexible properties and workable events.

The development environment whose prototype is made is completely visual because it is intended for non-programmers. All elements are concrete objects presented with icons that will enable simple and effective handling by logopeds, with the aim of simple design and creation of application programs. Visual syntax will allow logopeds to create their own program by simply customizing components in the work window, and by stacking them together and linking them to the scene.

The Development Environment for Speech and Language Therapist (DESLT) contains the main menu through which the logoped-programmer creates a future therapeutic application as a new project.

After that the window of the development environment appears (Figure 1). This window contains a toolbar with components that allow you to visualize objects and their composition in complex designs of future applications using point-and-click and drag-and-drop techniques.

---

4 What You See Is What You Get
Programming is reduced to placing an arbitrary number of objects on the scene. Before that, the logoped-programmer instantiates them in the right-hand part of the screen (WorkSpace) and specifies their properties in the left part (Object) (Figure 1).

When creating an application after it instantiated the components in the right part of the screen (WorkSpace), they set their properties to the left (Object) and arranged them on the scene (Scene), logoped tasks, and certain rules (Figure 2).

While the logoped using drag & drop technique instantiated the component, in the background, this finished component was placed in the list of interface components as predicted by a software engineer. At the end of the installation of the future therapeutic application, this list will contain all the instantiated components.

The rules are given by specifying the relations between the components that are selected from the list of components or directly selected on the scene. Figure 3 shows the rule: When the Text property of the Queue component is equal to the Text property of the WordWithTextImageSound component (WTIS), you rotate the component WTIS.
When defining the rules, the desired action is selected from the list of predefined actions (Rotate, Show, Hide, Run, Speak Off, etc.) previously programmed by the program engineer (Figure 4). These actions are represented by symbols behind which the corresponding C# code is written by a program engineer. The same actions are available to the logopedia and when defining the behavior of the component.

Figure 3. Defining the rules

Figure 4. The list of predefined actions: Rotate, Unhide, Hide, StartSound, Mute,...
8. Conclusions
Visual programming has an ambitious vision in the future: to enable a world in which the non-programmer, i.e. someone who has no experience in writing code can quickly create flexible software solutions that work to strengthen business processes and are able to meet his real business needs. The development of component-based software is a paradigm aimed at building applications using components developed by third parties.

In the future, the rapid and easy development of therapeutic software by the logoped developer is possible using the libraries of pre-built components. By including only a small number of program engineers, the new components could be designed and distributed over the World Wide Web. Logopeds should be able to download and install them in their environments with just one mouse click.

In order for these components to be not too complex, they are expected to be small in size, and that the web transfer will take a short time. The downloaded components will be able to install themselves, so the logopeds will not face the complicated and computer-dependent installation process.

Application programs for dyslexia therapy, which the logopeds will create, will feature the features of modern windows applications. A development environment with a design editor that resembles the most commonly used MS Office program for logopeds is user-friendly. Logopeds will be able to develop therapeutic software applications, as the most competent person, with previously developed training in using a specially created development environment for them. By empowering the logopeds as the ultimate beneficiaries in the development of their own application programs, it will initiate and give support to experts in other fields to become software creators for their own needs. Current software paradigms allow them that.

References
[1] ***2017 Tech: Where the jobs -- and demand -- are, March 23, USA TODAY, https://www.usatoday.com/story/tech/talkingtech/2017/03/22/tech-where-jobs/99496462/
[2] Lieberman H, Paterno F and Wulf V 2006 End User Development, Springer
[3] Ko A J, Abraham R, Beckwith L, Blackwell A, Burnett M M, Erwig M, Scaffidi Ch, Lawrance J, Lieberman H, Myers B A, Rosson M, Rothermel G, Shaw M, Wiedenbeck S 2011 The State of the Art in End-User Software Engineering, ACM Computing Surveys 43(3) 1-444
[4] Luger G 1994 Cognitive science : the science of intelligent systems, San Diego: Academic Press
[5] Gregory R L and Murrell J N 2006 Hugh Christopher Longuet-Higgins. 11 April 1923 - 27 March 2004: Elected FRS 1958, Biographical Memoirs Fell. R. Soc., pp 149-166
[6] Lighthill J 1973 Artificial Intelligence: A General Survey, Artificial Intelligence: a paper symposium, Science Research Council
[7] Lakoff G and Johnson M 1999 Philosophy in the Flesh: The Embodied Mind and its Challenge to Western Thought, New York: Basic Books
[8] Green T R G 1989 Cognitive dimensions of notations, People and Computers IV, Cambridge, Cambridge University Press
[9] Green T R G 1990 Programming languages as information structures, Psychology of Programming, London, Academic Press, pp 117-138
[10] Pane J F, Ratanamahatana C A and Myers B A 2001 Studying the language and structure in non-programmers solutions to programming problems, International Journal of Human-Computer Studies 54 237-264
[11] Burnett M 1999 Visual Programming, Encyclopedia of Electrical and Electronics Engineering, Postscript, pp 275
[12] Wesley M, Johnston J R, Hanna P and Millar R J Advances in Dataflow Programming Languages, University of Ulster, World Wide Web, http://www.cs.ucf.edu/~dcm/Teaching/COT4810-Spring2011/Literature/DataFlowProgrammingLanguages.pdf
[13] D’Souza D F and Wills A C 1997 *Objects, Components, and Frameworks with UML – the Catalysis Approach*, Addison-Wesley Longman Publishing, Boston MA USA

[14] Pour G 1998 *Component Based Software Development Approach: New Opportunities and Challenges*, Proceedings *Technology of Object-Oriented Languages*, Proceedings of Technology of Object-Oriented Languages and Systems TOOLS 26 (Cat. No.98EX176), Santa Barbara, CA, USA

[15] Pour G, Griss M and Favaro J 1999 Making the Transition to Component-Based Enterprise Software Development: Overcoming the Obstacles – Patterns for Success, Proceedings of Technology of Object-Oriented Languages and Systems TOOLS 29 (Cat. No.PR00275), Nancy, France, France