Technical Note PR-TN 2007/00695

Issued: 10/2007

Making end-user programming easy is difficult

L. Holenderski
Philips Research Europe

Unclassified
© Koninklijke Philips Electronics N.V. 2007
Title: Easy end-user programming

Author(s): L. Holenderski

Reviewer(s): Jos van Haaren (j.a.m.m.van.haaren@philips.com)

Technical Note: PR-TN 2007/00695

Additional Numbers:

Subcategory: 

Project: BRAINS

Customer: Philips Research

Keywords: programming, configuration, end-user, home devices, interaction, complex systems

Abstract: We give a high-level, and quite abstract, overview of several aspects of end-user programming, in hope that such an overview can help in positioning further research on end-user programming.

Conclusions: CE home devices are getting multi-purpose, multi-connected and multi-user. This growing complexity makes the devices more difficult to use and control. The Ambient Intelligence vision promises to solve this problem in a clever way, by simply removing the need to control the devices at all. Before this noble goal is achieved, we propose to consider end-user programming as an intermediate, and complementary, solution.

Making end-user programming easy is difficult. Below we list several potential challenges.

Among the many possible approaches to make end-user programming easy, we advocate tangible interfaces as the most promising solution. For example, robotic devices (used in the BRAINS project) or sensor boards (used in the Interactive Toys project). The most important technical challenge is to find the right metaphors to represent conditions, when using tangible objects, since conditions distinguish programming from configuration.

Research on end-user programming needs multi-disciplinary teams. As a minimum, a team must consist of interaction designers and computer
scientists. If tangible interfaces are involved, mechanical and electrical engineers are needed, and support from Industrial Design specialists would be helpful. Forming such diverse teams is costly and poses an organizational challenge. This may be an obstacle in further research on end-user programming.

Since end-user programming should, by definition, be user-centric, further research on end-user programming should be driven by concrete applications. The problem is whether we should have separate research teams inside projects devoted to particular applications or the subject of easy end-user programming should be researched in centralized way. Since tangible interfaces are application specific, a distributed way would probably lead to better (I mean fine-tuned) solutions but would be more costly. To reduce the cost, more generic solutions could be considered, but these would be inherently more difficult to find since tangible interfaces are specific by nature.
Contents

1. Introduction ........................................................................................................7

2. Sources of complexity ........................................................................................8

3. Configuration versus programming ....................................................................9

4. The space of potential solutions ........................................................................10
   4.1. Thoughts .......................................................................................................10
   4.2. Natural language ..........................................................................................10
   4.3. Artificial languages ......................................................................................11
       4.3.1. Text .......................................................................................................11
       4.3.2. Graphics ..............................................................................................11
       4.3.3. Gestures ...............................................................................................11
       4.3.4. Tangible objects ..................................................................................12
       4.3.5. Hybrid approaches .............................................................................12

5. Generic versus application specific solutions ....................................................13

6. End-user programming and Ambient Intelligence .............................................15

7. Conclusions ........................................................................................................16
1. Introduction

There is growing concern that devices we use in our daily life are becoming more and more complex to use. This is of course not new. The challenge was recognized several years ago, around 2003, and various activities were started in Philips to proactively approach the problem. The two most prominent are the Sense and Simplicity campaign and the Ambient Intelligence vision.

The Ambient Intelligence vision proposes to solve the complexity problem by providing a home environment in which various devices will seamlessly cooperate without much user intervention. Another approach to solving the complexity problem is end-user programming where the user is given explicit control over the way her devices should cooperate in order to fulfil her needs.

In this report we give a high-level, and quite abstract, overview of several aspects of end-user programming, in hope that such an overview can help in positioning further research on end-user programming.

We start with the analysis of the sources of complexity in using devices (see Section 2). Then we analyse the essential difference between configuring devices and programming them (see Section 3). The most important conclusion is that configuration is all about performing simple actions while programming is all about conditions, and this explains why making end-user programming easy is so difficult (we simply do not have obvious metaphors for representing conditions).

In Section 4, a quick overview of the space of potential solutions to the problem of easy end-user programming is given. We conclude with the observation that programming with the use of tangible objects seems to be the most promising line of further research in this area.

We also give some thoughts to the problem of generic versus application specific solutions (see Section 5).

We close the report with the observation that, in fact, end-user programming can be seen as complementary to Ambient Intelligence (see Section 6).

Finally, we give some general conclusions about positioning further research on easy end-user programming. These should hopefully follow from the various simple observations we make in this report.
2. Sources of complexity

Using devices in the past was simple. First, devices were single-purpose and we clearly understood both the purpose and the way to use a device to achieve the purpose. So a phone was just a phone. It allowed us to call another person and it was obvious how to do this. We just needed to dial a number.

Second, devices were isolated from other devices, in the sense that they were seldom connected to, and cooperated with, other devices. Even if they did, as in the case of a phone, they usually cooperated with devices of the same kind and the cooperation was so simple to understand that we took it for granted. Today, when I keep in my hand an expensive handset I have recently bought, I can seldom take things for granted. For example, I know that my handset has this wonderful Bluetooth interface, as advertised on the packaging, so it can connect wirelessly to some other devices I have at home, say a PC. So I conclude I could use my handset as a remote controller, to select a channel on my digital TV (that I have also recently bought) via my home PC (which I have just connected to the TV). The problem is that I don't have a clue how to do this without reading several manuals and spending several frustrating hours on proper configuration of all the devices involved. I just give up and make a mental note to buy next time a cheaper handset without all these extras I won't use anyway.

Third, devices were meant to be used by a single user. Not anymore. Soon, the screen of my TV will be controlled not only by me, but also by my friends with whom I'll be remotely watching a sport event. My photo frame will not only show a family picture I have loaded from my USB stick, but will simultaneously serve as a monitor for my security camera.

In summary, from the end-user perspective, there are three main sources of complexity when using CE devices:

- **multi-purpose** (they provide diverse functionality)
- **multi-connected** (they can be connected to, and cooperate with, other devices, not necessarily of the same kind)
- **multi-user** (they can be used by many users, or be shared by many devices)
3. Configuration versus programming

Most current CE devices, but the simplest ones, need to be instructed on what particular function to perform and how to perform it. For example, a TV set can show TV channels or DVD movies, or serve as a monitor to a home PC. So before watching contents from any of the three sources, I have to set my TV to a respective mode of operation. In addition, I can set various parameters (like brightness, contrast and screen size) that instruct on how various video contents should be shown to me.

Two terms, configuration and end-user programming, are commonly used to describe the process of instructing a device on what particular function to perform and how to perform it. Although there is some overlap between the two terms, they relate to quite different activities, as explained below.

The essence of configuration is to choose from a finite set of predefined possibilities. Typically, the predefined possibilities are various modes of operation or values of parameters (like in the examples above).

While configuration boils down to manually performing one particular action, programming is about describing a process that will automatically perform a sequence of actions. The sequence is constructed dynamically, by an execution engine. What actions are performed, and in what order, is typically governed by conditions. For example, my security camera can be connected to several monitors in my home, like a photo frame, TV set and PC. I want the image from the camera to follow me, when I move around home, so I want to instruct the system to show the image on a monitor which is closest to me, and is switched on. Suppose the camera is equipped with a sophisticated image recognition system that can decide when an image represents a dangerous, as far as my security is concerned, situation. Then I may want to instruct the system to show normal (i.e., not dangerous) images in a passive way, on my photo frame, while the dangerous ones in a more active way, either on my TV set or on my PC monitor, depending on which is closest to me at the moment, and which of them is switched on.

What is important in the above explanation is that actions are supposed to be performed conditionally, and this explains why making end-user programming easy is so difficult. We simply do not have obvious metaphors for representing complex conditions.

In short, while configuration is all about what and how, programming is all about when. This essential difference suggests that research on end-user programming should concentrate on easy ways of forming complex conditions, and how to associate such conditions with particular actions. Actions themselves are less of a problem because we can reuse many techniques known from configuration.
4. The space of potential solutions

The problem of easy end-user programming can be solved in many ways. Here I present my own classification of potential solutions and mention some of their drawbacks.

4.1. Thoughts

The ultimate solution to the problem of easy end-user programming would be the read my mind approach. Imagine a device that can sense your brain waves. Assume the device is so intelligent it can analyse the brain waves, deduce your specific needs, and find a way to adapt its functionality to fulfil them. You would program such a device simply by thinking.

One could even consider a more advanced approach I would term anticipate my mind. If the device was fast enough, it could guess your needs before you consciously formulated your thoughts, like a good butler who serves your favourite drink before you even realize that this is exactly what you have just wanted. Whether this particular solution would please most people is questionable, due to a serious invasion of privacy.

Unfortunately, the current technology is far from realizing this kind of ultimate solutions. The state-of-the-art is to wear a heavy helmet, or electrodes glued on your head, all the time and interact with your home devices in terms of binary code (I mean yes/no answers to a limited set of simple questions).

4.2. Natural language

The second best solution would be a natural language, either spoken or written. Again, the current technology is not mature enough to consider this solution feasible. The main unsolved problem is a reliable semantical analysis of what we say or write. This is a really difficult problem and the Artificial Intelligence research community is nowhere close to solving it.

The second unsolved problem is the way we input natural language phrases to a language processing system. In case of a spoken language, recognizing words (by recognizing particular phonemes and grouping them into sequences) still remains problematic. In case of a written language, the problem of recognizing words disappears, but in expense of the keyboard problem. The current ways of entering text are too cumbersome for ordinary users of home devices. A PC like keyboard is just too big, and smaller ones, like in mobile phones, are not convenient for longer texts. Virtual keyboards realized with the help of touch screens are still too expensive. Unless some innovative ways of entering text are found, the keyboard problem will hinder the usefulness of using a written natural language for easy end-user programming.
4.3. Artificial languages

Currently, the only feasible solution is to use an artificial language. Artificial languages have many possible representations, and we analyse some of them below.

4.3.1. Text

Traditional programming languages are textual. Using them for easy end-user programming is out of question since such languages are simply too difficult for non-professional programmers. Even if one found a way to radically simplify them, the approach would suffer from the keyboard problem anyway.

4.3.2. Graphics

There have been various approaches to make traditional programming languages easier by incorporating a graphical, instead of textual, representation of symbols and hiding the complexity of a language behind a fancy GUI. This is what most traditional research on easy programming concentrates on. The graphical languages have turned out to be quite successful in some particular application domains. For example, in robotics. There the most prominent are Philips OPPR (the software system used to program iCat), Lego Mindstorms, MIT Scratch and Microsoft Robotic Studio.

The main drawback of graphical approaches is that they currently relay on using a PC (needed to create/draw programs, and compile them). In general, graphical approaches to programming work quite well in situations where users are semi-professionals (say, hobbyists and researchers) and a PC is an accepted input device. Although these limitations currently exclude graphical approaches as suitable for the application domain of home CE devices, we still consider graphical notations as an interesting approach, worth to be pushed further. We claim that the current limitations could be lifted if some further research was devoted to the problem, as explained at the end of Section 5.

4.3.3. Gestures

Another class of artificial languages are languages based on gestures, as used by deaf people and divers, for example. In order to instruct a device by gestures one would need a camera, or a touch screen, equipped with a sophisticated image processing software that would recognize particular movements of your hands and interpret them accordingly. A nice example of how such a system could work in practice is shown in one of the scenes of Minority Report movie where the main hero controls a sophisticated computer system by simply waving his hands.

Although existing technology makes this solution feasible (several demonstrators can be found on YouTube; just search for "Minority Report"), the cost is prohibitively high. In addition to the high cost of current gesture recognition systems, languages based on gestures suffer from a more fundamental problem: the set of gestures with universally accepted meaning is rather small. As a consequence, users would have to learn new sets of specific gestures for controlling different kind of devices. For example, controlling
your light system would probably need a different set of gestures than controlling your DVD recorder.

4.3.4. Tangible objects

The problem with universally accepted gestures can be alleviated by tangible interfaces. A tangible object can easily convey a meaning, with little effort from a user to guess the meaning. For example, the meaning of the up/down buttons on the traditional remote controller for a TV set is so obvious that we take for granted how easy it is to adjust the volume of a TV set, or select a TV channel, just by pressing the buttons. Some spectacular examples of how easily a sophisticated light system can be configured with some innovative tangible interfaces are demonstrated in Philips ShopLab. This value of tangible interfaces has been recognized for a long time. In fact, we configure most devices with the help of tangible interfaces.

In my opinion, languages based on tangible interfaces are the most promising approaches to make end-user programming easy. Especially since Philips Research has quite a good expertise in this area. There are currently at least three teams working on tangible interfaces: robotic devices (in the group of Jos van Haaren), toys (in the group of Eelco Dijkstra) and ShopLab (in the group of Reinder Haakma). Some activities of these teams could be aligned in the direction of using tangible interfaces for easy end-user programming.

Of course, further research is still needed to find proper (i.e., the most natural for a particular class of users and applications) sets of objects, and ways of manipulating them to encode programs.

4.3.5. Hybrid approaches

One can also consider hybrid solutions that combine several approaches. For example, imagine a 3D monitor equipped with some sensors (say, cameras) that can measure the position of your hand in front of the monitor. Assume the monitor shows a scene (say, generated by a 3D gaming engine) that contains visualizations of some tangible objects. The scene appears to you in front of the screen (that's why we need a 3D monitor for) so you can attempt to grab the objects and reposition them. Of course, the objects are virtual, not tangible, so you are not really repositioning them. Instead, the 3D gaming engine creates a sequence of new scenes in response to the sensors that measure the position of your hand. This is very similar to virtual reality but without the cumbersome gloves, helmets and goggles. Notice that this solution combines graphical, gesture and tangible interfaces. A similar combination can be achieved by using a touch screen, as in the Philips Entertaible, but without the cool 3D effect.
5. **Generic versus application specific solutions**

The most popular languages used in traditional programming are general purpose languages like C, C++, Java or Python. Although many domain specific languages were designed and implemented, they are not commonly used by professional programmers. This may seem strange because they have an obvious advantage over the mainstream programming languages in that they can dramatically increase productivity in developing complex software. (This is due to the fact that domain specific languages are usually much higher level than the mainstream general purpose languages.)

There are two main reasons why professional programmers avoid domain specific languages. First, an additional effort is required to learn new languages and new software development tools. Second, software written in an exotic language is more difficult to develop and maintain since software development/maintenance teams fluctuate, and it would be more difficult to find new team members with similar capabilities to the members that have left the team.

In end-user programming, the situation is completely opposite in that the role of general purpose languages diminishes in favour of domain specific languages, or even more, in favour of languages tailored to particular applications. General purpose programming languages are based on various abstractions so their use requires special skills (most notably, abstraction abilities) that are not common among ordinary users of devices. Thus, easy end-user programming should instead be based on concrete metaphors known from daily life and these are usually specific to particular activities/applications.

The above observation has the most important consequence for end-user programming based on tangible interfaces. In our opinion, tangible programming solutions should be application specific and no research should be devoted to general purpose tangible programming (the chances for a reasonably general solution are rather slim since there is almost no room for abstractions when dealing with tangible objects).

Application specific solutions are costly since they need to be developed from scratch for every new application. As mentioned above, this cost should be treated as inherent in tangible programming and research efforts should concentrate on reducing the cost of each specific tangible interface instead. However, another path of research could be considered which specifically concentrates on reducing this cost, as explained below.

Graphical languages offer some room for abstractions so it could be possible, in principle, to develop graphical metaphors generic enough to program a class of applications, either for specific devices or even for all home devices (on some level of abstraction, all devices are just reactive systems, a concept well known and studied in computer science). As noted before, one of the main drawbacks of using graphical languages for easy end user-programming is that graphical languages suffer from the PC problem (currently, a PC is needed to write/draw a program and compile it). However, this problem will be solved in several years when the Tablet PC becomes cheap enough to be considered a feasible replacement for the traditional remote controller. Think in terms of a device similar to Apple’s iPhone, with a touch screen on which you can manipulate various graphical elements with your finger.
An additional argument for investing in research into graphical languages is that graphical languages offer some additional opportunities. In short, they may be low hanging fruits there. On one hand, the subject has been studied for a long time in academia, so there is strong competition. On the other hand, academic research on easy programming is typically done by computer scientists who are software developers themselves and who usually advocate their favourite, and quite traditional, programming methods disguised in fancy GUI. A typical technology push, without proper user studies. Such an approach misses the whole point of easy end-user programming in that instead of being user-centric it is technology-centric. Philips has a potential advantage of being user-centric so some innovative graphical languages could be found if research on them was done from the perspective of user needs, in multi-disciplinary teams that combine programming language specialists with user interaction specialists.
6. **End-user programming and Ambient Intelligence**

To cope with the problem of growing complexity in using home devices the Ambient Intelligence vision was proposed. According to this vision the devices will be embedded in an ambient network of sensors and some sophisticated control software will use the sensors to arrange the cooperation of the devices in such a way that the needs of users will be fulfilled seamlessly.

However noble the vision is, it suffers from two main problems. First, we are still far away from realizing the vision, so in the meantime some other means of controlling the devices are needed, and we propose end-user programming as one of such means. Second, even if the vision is realized, there will be situations where Ambient Intelligence will wrongly predict the user needs and will have to be corrected. In other words, the user will have to have some means to instruct the system to do things differently, and this is where easy end-user programming can help.

In summary, end-user programming can be viewed as either complementary to the Ambient Intelligence vision or, in a longer term, as part of this vision.
7. Conclusions

CE home devices are getting multi-purpose, multi-connected and multi-user. This growing complexity makes the devices more difficult to use and control. The Ambient Intelligence vision promises to solve this problem in a clever way, by simply removing the need to control the devices at all. Before this noble goal is achieved, we propose to consider end-user programming as an intermediate, and complementary, solution.

Making end-user programming easy is difficult. Below we list several potential challenges.

Among the many possible approaches to make end-user programming easy, we advocate tangible interfaces as the most promising solution. For example, robotic devices (used in the BRAINS project) or sensor boards (used in the Interactive Toys project). The most important technical challenge is to find the right metaphors to represent conditions, when using tangible objects, since conditions distinguish programming from configuration.

Research on end-user programming needs multi-disciplinary teams. As a minimum, a team must consist of interaction designers and computer scientists. If tangible interfaces are involved, mechanical and electrical engineers are needed, and support from Industrial Design specialists would be helpful. Forming such diverse teams is costly and poses an organizational challenge. This may be an obstacle in further research on end-user programming.

Since end-user programming should, by definition, be user-centric, further research on end-user programming should be driven by concrete applications. The problem is whether we should have separate research teams inside projects devoted to particular applications or the subject of easy end-user programming should be researched in centralized way. Since tangible interfaces are application specific, a distributed way would probably lead to better (I mean fine-tuned) solutions but would be more costly. To reduce the cost, more generic solutions could be considered, but these would be inherently more difficult to find since tangible interfaces are specific by nature.