End User Programming

Christian Brockly
RWTH Aachen
Templergraben 55
52056 Aachen, Germany
Christian.Brockly@rwth-aachen.de

Klaus Meyer
RWTH Aachen
Templergraben 55
52056 Aachen, Germany
Klaus.Meyer@rwth-aachen.de

ABSTRACT
There has always been and there will always be a wide gulf between creators and users of computer programs. While only a few users actually know how to program in usual programming languages, almost all users want to modify existing applications by adding additional features or specifying their behavior, and even inventing new ones becomes important when looking at technology-enriched homes. There would be a greater benefit of the added technology when users were able to decide how their devices should work and react instead of just using the pre-defined actions.

That is the point where *End User Programming* (EUP) comes into play. This paper shows the importance of providing more natural and intuitive programming tools for end users, gives an overview on different styles of End User Programming, presents some example applications that are already implemented and gives an outlook on possible future developments.

1. INTRODUCTION
End User Programming, or in general End User Development, is a research topic within the field of human computer interaction, concerning techniques to enable end users to create and modify (extend as well as adapt) applications for their needs.

Usually, a software developer has a degree in computer science or taken courses in software engineering. His primary job function is to write and maintain software [11]. Therefore he knows how to write a program in a programming language, use *for*- and *while*-constructs and create class hierarchies, while, in general, he is not an expert in the field that his software serves for. The experts are on the user side. Constabile et al. [4] call them *domain-expert users* as "*they are experts in a specific domain, not necessarily experts in computer science, who use computer environments to perform their daily tasks*." Therefore it should be their task, or at least they should be enabled, to decide how a computer program acts. According to Myers et al. [11], they then would be "End User Programmers, people who write programs, but not as their primary job function."

End User Programming is not limited to programming tasks as we know them from usual programming languages. It is more a principle that is used to enable users to include their knowledge and ideas in their computer software. Often natural metaphors are used to provide easily understandable interfaces. For example, modern office suites and operating systems provide so-called *macro recorders* that simply record the actions that the user performs and create a program out of them, sometimes even in an imperative language. *Spreadsheets* are somehow paper-style tables extended by features that let end users define automatic calculations or diagrams. *Visual programming* interfaces like Max/MSP or LabView let users connect visual components by a plug and cable metaphor to compose their programs. Recent developments like *wikis* and *blogs*, where common internet users are enabled to create complex web pages and huge databases, follow the idea of end user development as well—very successfully. And all of them have one in common, they require no coding but just visual-based development.

End User Programming becomes important when it comes to ubiquitous computing in home environments. In technology-enriched homes it is not obvious how the technology is controlled. Instead of supplying many separated task specific applications to the end users, it is more effective to let the users specify themselves how to use the available ubiquitous technologies.

1.1 Threshold and Ceiling
Threshold and ceiling are important aspects of user interface tools: The *threshold* is how difficult it is to learn using a system, and the *ceiling* is how much can be done with it [10]. Of course the goal is a system with low threshold and high ceiling. We will discuss threshold and ceiling of each example system in the following sections.

2. APPROACHES TO EUP
This section gives an overview over the different approaches to End User Programming. The research focuses in this area can be grouped in these four categories:

- Simplified textual languages
- Visual Programming
- Programming by Demonstration
- Better support by the development environment

2.1 Simplified Textual Languages
The most naive approach to enable end users to write programs is to create programming languages which are easier to learn and to use. The probably first language which tries to achieve this goal was BASIC, developed in the 1960s,

which stands for *Beginner's All-purpose Symbolic Instruction Code*. Many other programming and scripting languages followed, e.g. HyperTalk for Apple's HyperCard, or ActionScript for Adobe Flash. However, all those languages are still not really suitable for end users for the same reasons: Users still have learn the syntax and have to deal with unfamiliar programming concepts like abstraction, variables, and loops.

2.1.1 Chickenfoot

A step in the right direction is *Chickenfoot*[1], an automation tool for the Firefox browser by Bolin [2, 3]. Using Chickenfoot, end users can manipulate the appearance of web pages without knowing their source code. It serves as an interface to JavaScript, but instead of having a fixed syntax it uses keyword pattern matching: By recognizing keywords in the command, Chickenfoot determines the action. All parameters refer to the web page appearance, not to the source code. For example, the keywords `click`, `press`, and `push` denote clicking a link or a button on the web page, so on the Google home page the command

```
click feeling lucky
```

is sufficient to invoke a click on the button "I'm feeling lucky." To avoid ambiguity, such commands can also be written like JavaScript function calls:

```
click("feeling lucky")
```

Due to the flexibility and the lack of a fixed syntax Chickenfoot is very easy to learn and almost intuitive to use. For simple programming web automation tasks, the threshold is relatively low. However, for more complex scripts the programmer needs a deeper understanding of JavaScript's concepts, so there is a second threshold, a *wall*, which is still lower than the JavaScript threshold but may be too high for end users. The ceiling of Chickenfoot is the same as the high JavaScript ceiling.

2.1.2 HANDS

Pane [14] conducted user studies to investigate how people, especially children, solve problems and think about programming tasks. The goal is to develop programming systems and tools which enable users to program in a more "natural" way, thus Myers et al. call this approach *Natural Programming* [12].

Some results of the studies were:

- **Event-based structure:** The program description developed by an end user is usually not flow-based (with loops and branches) but rather event-based or rule-based, that is, events are described and how the program should react to them.

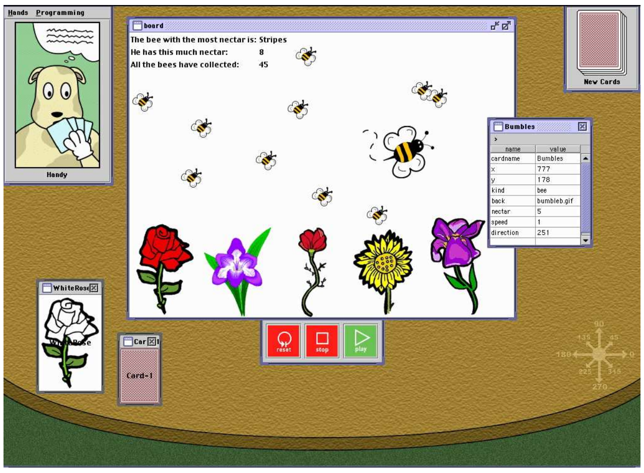
- **Aggregate operations:** A typical pattern was the use of aggregate operators, which operate on a set of objects instead of on just a single object. Such operations were preferred to iterations through the set. An example for an aggregate operation is "Set a, b, and c to 0" instead of "Set a to 0, b to 0, and c to 0."

- **Content-based queries:** The use of control structures like lists or sets is not stated explicitly but rather implied. Also the construction of subsets of existing sets was only implied using queries, e.g. in a PacMan game, a set was referenced with "all of the blue monsters." So again, iterating a set was untypical for end users.

- **No boolean expressions:** The participants did not use Boolean expressions very often, and it turned out that they have a different interpretation from usual programming languages. In 2.2.4 we show how to overcome this problem.

- **Visual vs. Textual:** Although the participants used textual descriptions for actions and behaviors, they often used a visual representation of the overall program layout.

[1]http://groups.csail.mit.edu/uid/chickenfoot/



The bee with the most nectar is: Stripes
He has this much nectar: 8
All the bees have collected: 45

name	value
cardname	Bumbles
x	777
y	178
kind	bee
back	bumbleb.gif
nectar	5
speed	1
direction	251

Figure 1. The GUI of the HANDS programming system. The dog Handy in the upper left corner visualizes the execution of instructions by manipulating cards.

The programming system *Human-centered Advances for the Novice Development of Software* (HANDS) was designed based on these results. Some aspects are represented visually instead of textually, the language is event-based, and it features aggregate operations and content-based queries.

All objects in HANDS are represented by *cards*, which have user-defined properties. The program execution, that is, the manipulation of cards, is represented by an Agent called Handy (see Figure 1).

The keyword `all` denotes a query. If it is followed by only one word, it matches all cards which have this value in any property, e.g. if all cards representing flowers have the property `group` instantiated with `flower`, they are matched by the query

```
all flower
```

It is even allowed to append a plural-s, so `flowers` works as well. Conditions are also possible, e.g.:

```
all (nectar < 100)
```

Since queries return lists of cards, queries and aggregate operations can be combined. In a limited version of HANDS which does not support aggregate operators, it is necessary to iterate through a list:

```
with flowerList calling each flower
    set the nectar of the flower to 0
end with
```

In the full version of HANDS, an aggregate operation simplifies this a lot:

```
set the nectar of all flowers to 0
```

A study with children showed that the participants were more successful in solving programming tasks using HANDS than using a limited version of HANDS, which did not feature queries and aggregate operations and had reduced data visibility. Overall, the participants enjoyed the full version more and found it less difficult than the limited version. Thus, HANDS has a very low threshold and is suitable as programming system for beginners, especially for children. Its ceiling is moderate: One the one hand, HANDS allows for programming reasonably complex games and general programs, e.g. a prime number calculation, but it lacks modularity and abstraction. It is not possible to create subroutines, which are essential for higher complexity.

2.1.3 CAMP: Capture and Access Magnetic Poetry

CAMP [18] is a programming system which enables end users to build ubicomp capture-and-access applications for their home. The actions supported by CAMP are *capture*, *access*, and *delete*, and the possible capture data types are *still-pictures*, *audio*, and *video*. CAMP requires that the used technology is context-aware, i.e. it can recognize the contexts of people and objects, such as locations and activities.

The user interface is based on the magnetic poetry metaphor: Magnetic poetry sets consist of small magnetic tiles which are typically put on a refrigerator door. Each tile has a word or a word fragment printed on it, so the tiles can be combined to form phrases or "poems." The advantage of magnetic poetry is that end users can use a natural language to create flexible applications but they are also restricted by the vocabulary, which allows for evaluation of the formed phrases by the system.

User Study

CAMP was developed based on a user study to find out how end users perceive and describe home applications for capture and access. The authors collected ideas and descriptions for such applications and found the following categories:

- **Provide Peace of Mind:** Applications that make users feel secure, e.g. by monitoring their children.
- **Collect Records of Everyday Tasks/Objects:** Applications that record events for convenience, e.g. to keep track of the car key.
- **Preserve Memories of Experiences:** Applications that record events in advance so that no special memorable events are lost, e.g. if there is no handheld camera nearby.

Many participants perceived the system as an *effector*, which executes user commands. However, some participants perceived it as an *assistant*, which helps the user performing a task, and some perceived it as a hybrid of effector and assistant. Common to all models is the use of the "W dimensions", i.e. *who*, *what*, *where*, and *when*, to describe capture situations.

CAMP Design

Initially the vocabulary and the layout of the magnetic poetry GUI were designed based on this study, in particular all words are grouped in the categories *who*, *what*, *where*, *when*, and *general*. Users can drag words from the vocabulary area to the authoring area and arrange them to phrases (see Figure 2). They can also add new words to the vocabulary and define their meaning, e.g. "*dinner happens between 7 and 9 PM in dining room*" to define "dinner."

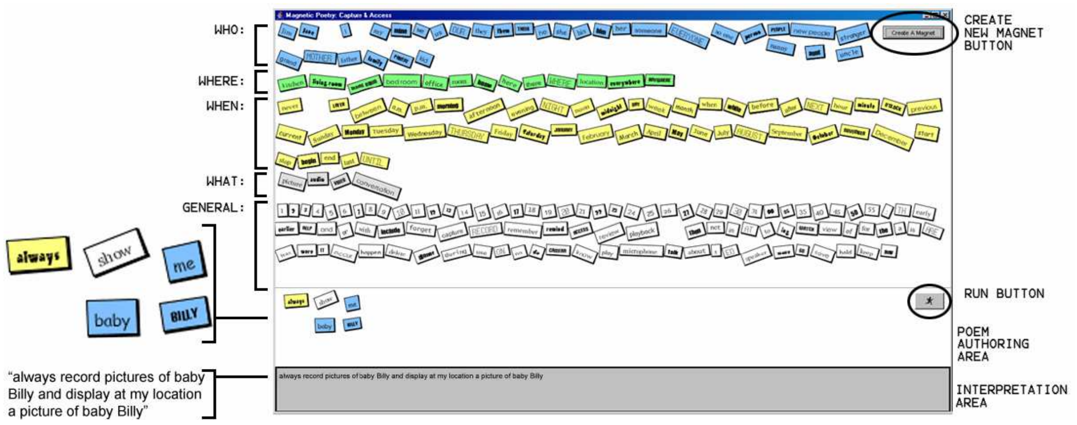
When the "run" button is pressed, CAMP processes the phrase and gives feedback by displaying an interpretation in the interface. First the phrase is simplified and normalized by replacing synonymous expressions, e.g. "*starting at 7 PM capture kitchen for 2 hours*" is rephrased to "*between 7:00 PM and 9:00 PM record kitchen*." Then the phrase is decomposed into sub-clauses, especially when the Boolean connectives "or" or "and" are involved. To resolve conflicts and ambiguity, CAMP assigns priorities to the used statements and assumptions, so that explicitly stated values are more important than implicit ones. For example, in "*capture dinner everywhere*" the word "everywhere" overrides the fact that dinner happens in the dining room. If a dimension is missing, e.g. the time is not specified, it is marked so that the user can refine the phrase.

A common pattern in the study was that the participants made implicit assumptions about the type of recording, e.g. they used statements like "record conversations" without specifying "audio" recording. CAMP takes that into account. If no recording type is specified, recording of still-pictures is assumed.

Because of the magnetic poetry metaphor, CAMP works similar to the keyword pattern matching of Chickenfoot, so it mainly uses the approach of "better textual languages." It also relies on extensive support by the development environment, since this controls the availability, visibility, and arrangement of the tiles.

Preliminary Evaluation

The authors of CAMP conducted a preliminary evaluation to verify the assumptions on which the CAMP interface is based. The results were similar to the results of the initial study: The applications invented by the participants fall



"always record pictures of baby Billy and display at my location a picture of baby Billy"

Figure 2. The CAMP interface consists of a vocabulary area, a poem authoring area, and an interpretation area. To form poems, the user can drag magnets from the vocabulary to the authoring area. When pressing the "run" button, CAMP displays an interpretation.

into the three categories mentioned above, and all participants were able to describe their applications easily using the CAMP interface, so the vocabulary seems to be sufficient for most capture-and-access applications. Not all applications were correct the first time but it was no problem to correct them. Overall, the participants found the CAMP interface easy to learn and intuitive to use. This shows that the CAMP system has a low threshold. The ceiling is just high enough so that the system serves its purpose very well: The development of ubicomp capture-and-access applications.

2.2 Visual Programming

There exists a variety of visual programming environments which generally try to visualize code by some kind of flow diagram. Visual programming languages can be further classified into diagram-based, icon-based and form-based languages. We will first take a look at two popular implementations of diagram-based languages, LabVIEW and Max/MSP, then discuss the principle of form-based applications. At the end we will show approaches for enabling users to visually construct boolean expressions and define ubicomp applications, which do not fit exactly into the three mentioned categories.

2.2.1 *Diagram-based languages*

Diagram-based languages usually use a data flow metaphor, e.g. the data is sent to and received from visual components via wires plugged into slots, usually on the upper (input) and lower (output) side. In general, an input serves for connecting one wire while output slots support connecting multiple ones.

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is a system designed to enable primarily scientist to perform and control tests, measure their outcomes and evaluate or even visualize them. So-called VIs, *virtual instruments*, have the same meaning as methods in imperative languages. They can be combined to build up new virtual instruments and further consist of two sides. One the one side the developer can specify the actions by arranging and connecting VIs, on the other side a graphical interface can be defined and shown. Each VI has inputs and outputs which can be connected by wires. Not only virtual components are available but real world control and measure instruments can be controlled and used to deliver data.

With newer version even object-oriented programming has become possible, including inheritance and polymorphism. Even multithreading, here even *automatic parallelization*, can be used by simply building up two independent data flows.

The other environment, *Max/MSP*, is a similar programming environment for multimedia applications. It offers a variety of components that are to be plugged together by wires to build up a data flow. Again, these components provide input and output plugs as a visualization for giving data to a function and reading the computation results. Max/MSP enables even hobby musicians to develop real time audio and video processing interfaces.

Both LabVIEW and Max/MSP have full expressive power in boolean logic and come with a huge set of predefined components. Actually, they provide a full higher programming language and although LabVIEW and Max/MSP thus have got a very high ceiling factor, simple programs can be build up easily with low threshold. More complex programs using all functionality (or only half of it), as in every other higher programming language, need to be planned carefully to keep the structures clean and understandable. One major problem in handling complex programs of diagram-based visual programming languages is obviously the lack of flexibility when refining code. If new actions are to be included the existing components and wires need to be rearranged or enclosed in subroutines to provide enough space to insert the new ones.

2.2.2 *Form-based languages*

Form-based visual programming environments provide an interface for the creation and manipulation of cells on forms. The cells' values can be either specific values, or formulas that reference values contained in other cells. The underlying evaluation engine then recalculates all influenced values whenever a cell's value is changed or formula is redefined.

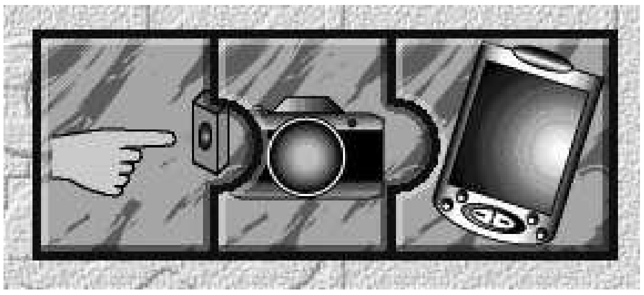


Figure 3. Jigsaw pieces used to combine the doorbell with a webcam and a portable display

The most prominent examples of such languages are spreadsheet applications.

Research [16] has shown that the main problem of form-based visual programming languages is that errors are frequently commited and that users have low confidence in the reliability of their programs. This tells us that for the efficient and correct use of such environments much effort has to be done. Depending on the complexity of the particular form-based language the user encounters both a high ceiling and a high threshold.

2.2.3 Icon-based languages

As an example of icon-based visual programming languages we will discuss the model of *jigsaw pieces* [7] for configuring applications for technology-enriched home environments.

As user studies showed that users are more interested in knowing how their devices are interconnected than in actually programming them, the jigsaw editor was constructed to let them easily define connections and simple action sequences. In the jigsaw editor users can connect jigsaw pieces which act as *digital/physical transformers*, components taking physical or digital effects and each transforming them into the other type. For deeper semantic actions *digital transformers* are used. This component class can perform calculations on digital information. One piece can consist of one or more transformers. Coupling of pieces is done in a left to right fashion, providing the image of an information flow.

The editor is composed of two panels. On the upper one the user can select all available pieces which are automatically collected in a distrubuted dataspace to which the installed ubicomp devices export their properties. The lower one is the actual workspace serving for the assembly of pieces. To avoid assembling pieces that would not result in plausible actions those pieces are shadowed and disabled when the user tries to establish connections. When two or more pieces are put together the underlying engine connects the corresponding input and output properties (if there are multiple corresponding properties a dialog will ask the user to choose the desired one). As an example, Figure 3 shows how a webcam can be configured to take a photo and send it to a portable display device when the doorbell button is pushed.

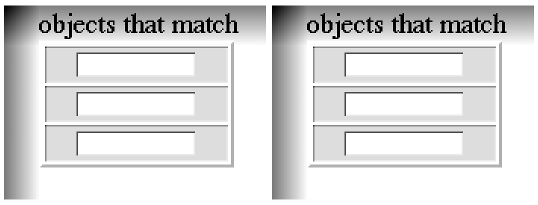


Figure 4. Two conjunctions combined to one disjunction using match forms.

As said before the the jigsaw editor does not aim at programming devices but only at reconfiguring them. Therefore the expressive power is quite restricted. iCAP and CAMP provide the user with much more freedom in defining complex action sequences. On the other hand, even very unexperienced users can rapidly use the jigsaw editor. Thus, we have got low ceiling and low threshold.

2.2.4 Match forms for defining boolean expressions

A user study conducted by Myers and Pane [13] revealed that users have certain problems with accurately defining boolean expressions using textual methods whereas the use of so-called match forms has a significant impact on improving the performance. The main problems were the following ones.

1. Users interpret the AND operator depending on the context. In *select the objects that match blue and circle* AND is generally interpreted correctly, while in *select the objects that match blue and the objects that match circle* 55% of the users read it as a disjunction.

2. Users assign to the NOT operator a higher priority in OR clauses while AND clauses are in general interpreted correctly (*select the objects that match not red and square* and *select the objects that match not triangle or green*).

3. Users misinterpret parentheses as in *select the objects that match (not circle) or blue*.

Figure 4 shows match forms that let users define attribute-value pairs on cards which themselves can be put onto other cards. Statements on the same form are then interpreted as a conjunction while distinct forms adjacent to each other are combined to disjunctions. In other words, match forms interpret horizontal arrangement as disjunction and vertical arrangement as conjunction. A preceding *not* negates the corresponding match form. The study then showed that users perform significantly better in generating queries using match forms than using text.

Both ceiling and threshold level of match forms are very low, but this model helps to understand how to enable users to specify boolean expresions easily, a problem which is often underestimated. We will look at iCAP now, an application that enables users to configure context-aware devices in technology enriched homes, which uses Myers' match forms to formulate *if-then-rules*.

2.2.5 iCAP, an icon-based programming environment

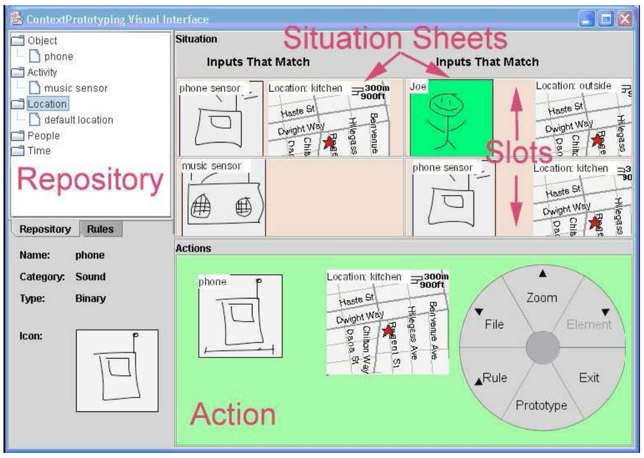


Figure 5. The iCAP interface with its repository, situation sheets with different slots and a sample action to be performed when the situation sheets are matched.

iCAP [6], an interactive Context-Aware Protopyer, follows roughly the same idea as CAMP. It is a tool to build context-aware ubicomp applications and tries to bridge the gulf between low-level toolkits for capture devices and fast prototyping results. Without iCAP, designers prototyping context-aware applications have to write a lot of code to get useful results as low-level toolkits rather support for acquiring the context than for creating applications. Writing software from scratch when no toolkits are available requires even more effort. The iCAP designers find that this fact inhibits the design of new applications as end users are not able to express their ideas freely and have only little control on the behavior of context-aware applications.

Initial user study

The iCAP design was inspired by the results of a user study that focused on retrieving information on *how* non-programmers want to build *which* context-aware applications. Thus, participants were asked to think of possible applications. The study showed the following:

- **Rules** *Every* participant used *if-then rules* to describe his application in contrast to only 5% that used declarative statements. Almost 80% of all rules used simple *if-then rules* while the rest was based on temporal, spatial and personal relationship rules (approx. 7% each). Only 1% focused on environmental personalization.
- **Categories** The rules had a low level of complexity and could be split up into six categories: *activity*, *object*, *location*, *time*, *person* and *state*. The average rule contained about 2.5 different categories.
- **Involved items** About 56% of the rules involved objects. Other important items were activities, location, time and people other then the subject.
- **Situations, actions, preferences** About 70% of the rules concerned the state of the subject (e.g. *if I'm doing...*), only 30% the state of the house (e.g. *when the water starts to boil...*). Almost all rules interpreted the house as an *effector*, only 7 of 371 rules as an *assistant*. Furthermore, specific behavior definitions were more common than descriptions of preferred actions (86% vs. 14%).

Based on this, iCAP was designed to enable users to "*describe an object's situation and command the house to act on that state, describing specific behaviors rather than preferences* [6]."

iCAP design and interaction

iCAP consists of two main components. The first one is a visual interface for the creation of rules, the second one the underlying engine for storage and evaluation. The visual interface itself consists of an element repository to the left and a rule creation area to the right. The upper part of the rule creation area is reserved for so-called *situation sheets*, the lower part for the desired actions that have to be taken if the context matches the defined conditions (see Figure 5).

The interaction with iCAP can be divided into three different steps—*creating elements, constructing rules* and *ambiguity and conflict resolution*.

Creating elements Based on the user study, iCAP distinguishes five categories of elements: objects, activities, locations, people and time. All created elements have a user-sketched graphical icon and are stored in the repository, which is shown on the left side of the interface window. Furthermore, they can be connected to real or simulated capture hardware (or to a combination of both).

Both *activities* and *objects* are treated in the same way. They have either a binary (e.g. *on/off*) or a gradient state (e.g. *volume*). Additionally, objects can output a string to simulate a certain behavior, e.g. an mp3 player playing "La bamba." As activity objects capture home or user activities they can only be placed on the situation side.

Location elements are used to specify a binding of the rule to a specific location. They can be related to make use of spatial adjancencies and convey environmental personalization features that are tried to be satisfied. *People* elements are quite similar. iCAP supports the creation of groups (*family member, roommate*). By default, an "I" people object is created. *Time* elements can be created to restrict rules to a specific moment.

Constructing Rules As described before, rules can be composed by adding elements to condition sheets. Different elements on one sheet are evaluated as a conjunction, distinct sheets combined to a disjunction. If needed, sheets can be subdivided to create more complex structures. We have seen, that using this match form scheme [13] even non-mathematical users can easily create rules without using usual boolean logic paradigms.

The action to be taken is specified below the situation sheets. This rule concept provides a high ceiling.

Ambiguity and conflict resolution Initially, rules can be defined using ambiguous locations, people etc. represented by wildcards (*anyone, anywhere*). If there are conflicting wildcards in a rule iCAP asks the user to disambiguate them. For example, ambiguity in *"when I am in an undefined location AND my friend is in an undefined location. . . "* could be resolved by specifying whether the two locations are equal or distinct.

Furthermore, iCAP checks whether there are conflicts among distinct rules. When saving, such rules are automatically highlighted by iCAP. If conflicts appear at runtime the last updated rule is chosen.

Scenarios can be evaluated using a Wizard-of-Oz interface, real world sensors or a combination of both. An interface to the Context Toolkit (CTK) [5] can automatically fill the repository, pass events and execute actions on real devices. This allows for rapid prototyping of new context-aware applications.

Evaluation user study

In an evaluation study participants were asked to think of own applications and given a set of actions by the user study conductors. Then they had to express both types of rules as precise as possible using iCAP. The study showed that users were successful in building a rich variety of rules—spatial, temporal and personal relationship were used, as well as simple if-then rules and personalization. Furthermore, iCAP supported almost all context-aware behaviors that users could imagine, only high ambiguity, actions based on time intervals and rules using complex concepts like *"it would be good if music was playing that was based on what I was cooking"* were not expressable.

Users did perform well on defining their rules without extraordinary training or skills and the expressive power of iCAP is sufficient to define almost all thinkable applications. Thus, iCAP has a high ceiling with low threshold.

2.3 Programming by Demonstration

In systems using the Programming by Demonstration (PBD) technique, the user interacts with the system to demonstrate how the desired program should work. An application is formed by generalizing the user interaction to "before-after" rules [11].

An example of PBD is Stagecast Creator, formerly known as KidSim or Cocoa [17]. Stagecast uses the spreadsheet paradigm by providing a game board with discrete cells. Some cells are occupied by objects, so-called *agents*, which can act according to given rules in each time step. The *graphical rewrite rules* consist of a "before" part and an "after" part and are created by demonstration. For example, by moving an agent in a rule window into an empty cell on the right hand side, this allows the agent to move to the right in each time step if the right cell is empty.

A common problem of PBD systems is the kind of representation of the final program, which allows end users to modify it. Usually PBD systems use textual or visual programming languages for such representations, e.g. Stagecast has a visual representation of existing rewrite rules.

PBD has the great advantage that end users can create programs without any conventional knowledge about programming. But on the other hand, such systems do not scale well. For relatively complex programs, e.g. sorting algorithms, a huge number of similar rules are necessary because of the lack of abstraction. Using the underlying programming language could be easier than using the PBD technique. Especially graphical rewrite rules have very strong affordances, which lead the end user programmer on a wrong way [15]. So in general PBD systems have low threshold and low ceiling.

2.4 Better Support by Development Environment

The most important aspect of EUP is increasing the support by the programming environment, both for creating and for debugging programs. Two notable collaborations in this research area are the EUSES Consortium (End Users Shaping Effective Software)[2] and the Network of Excellence on End-User Development[3]. Although they focus on end user software engineering, i.e. increasing the effectiveness and dependability of end user programs, this is mainly achieved by user-friendly and supportive programming environments and debugging tools.

Not only end users can benefit from improvements in this area, but also professional programmers are concerned. Integrated Development Environments (IDEs) assist them in the development. Most of the usual techniques for programming and debugging, such as syntax highlighting, breakpoints, and step-by-step execution, are already many years old, some have not even changed in the past 30 years [8]. New techniques that improve the quality of end users' programs are probably also suitable for professional IDEs.

2.4.1 Syntax-Directed Editors

Some environments, for example Alice, remove or reduce the possibility of syntax errors and resulting bugs by "syntax-directed editors" [1]. In Alice, an educational programming system, the language and the environment are not separated. Instead of typing the program, the programmer can choose

[2]http://eusesconsortium.org/

[3]http://giove.cnuce.cnr.it/eud-net.htm

commands, arguments, variables, etc. via drag-and-drop or pop-up menus. This both lowers threshold and ceiling compared to usual editors, since there are no syntax errors but it makes the development more tedious. However, there are also syntax directed editors which allow direct typing, or both.

Syntax-directed programming environments are already common in professional programming. Microsoft Visual Studio's IntelliSense parses the source code and provides useful information when needed. The Java IDE of Eclipse even supports incremental compilation so that errors and warnings can be displayed immediately in the editor.

2.4.2 Whyline

The Whyline [8] is a debugging tool for the Alice programming environment which uses a new debugging paradigm called *Interrogative Debugging*. Whyline is the first tool which supports *hypothesizing* activities, that is, making assumptions about what runtime actions caused failure. Especially when programmers make false assumptions due to a weak hypothesis, debugging will take much longer and even more errors are introduced. In conventional systems programmers can only verify their assumptions indirectly by mapping debugging strategies to the available tools, but using Whyline they can do this explicitly by asking questions of the form *Why did* or *Why didn't*.

User studies showed that questions of the forms "Why didn't" and "Why did" cover almost all questions asked by Alice programmers during debugging. To ask such a question, the user can click on the "Why" button in the Alice interface. From there, he can compose the question via a series of pop-up menus. For example, when testing a Pac-Man simulation, the user assumes that the method *resize* of the object Pac was not called and asks "Why didn't Pac resize .5?". If the resize method was actually called, Whyline informs the user about it, because he made a false assumption. But if it was not called, the Whyline (which is actually a timeline at the bottom of the interface, see Figure 6) shows the sequences of runtime actions which prevented the method from being called. The user can then browse in the timeline to highlight the according code.

In user studies to test the usability of the Whyline, it turned out to be very successful. It reduced debugging time dramatically by almost a factor of 8, and enabled the participants to complete 40% more tasks, compared to an Alice version without the Whyline. So it raises the ceiling a lot.

The Whyline is also a concept which could be extremely useful even for professional programmers. It is very likely that "Why" questions are very common in all debugging environments. However, it is not trivial to provide answers to such questions. Showing the current run-time state of a program is not a problem, but "Why" questions usually refer to events that happened in the past. For the Whyline and similar systems it is necessary to keep track of all important events that happen during run-time. The first tool which achieves this efficiently in a conventional programming language is the *Omniscient Debugger* [9] for Java. Although it does not allow for asking "Why" questions explicitly, it provides the necessary functionality to find the answers.

System	RS	SD	VP	Support
Chickenfoot	Yes	no	no	low
HANDS	Yes	no	no	low
CAMP	Yes	Yes	no	medium
LabVIEW	no	no	Yes	medium
Max/MSP	no	no	Yes	medium
Jigsaw Editor	no	no	Yes	medium
iCAP	no	no	Yes	medium
Stagecast	no	no	Yes	medium
Alice + Whyline	no	Yes	no	high

Table 1. Comparison of programming systems.
RS = Relaxed syntax, SD = Syntax-directed, VP = Visual Programming

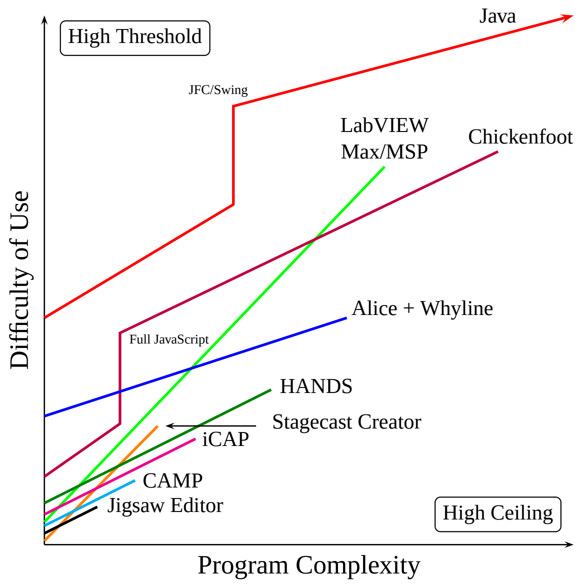
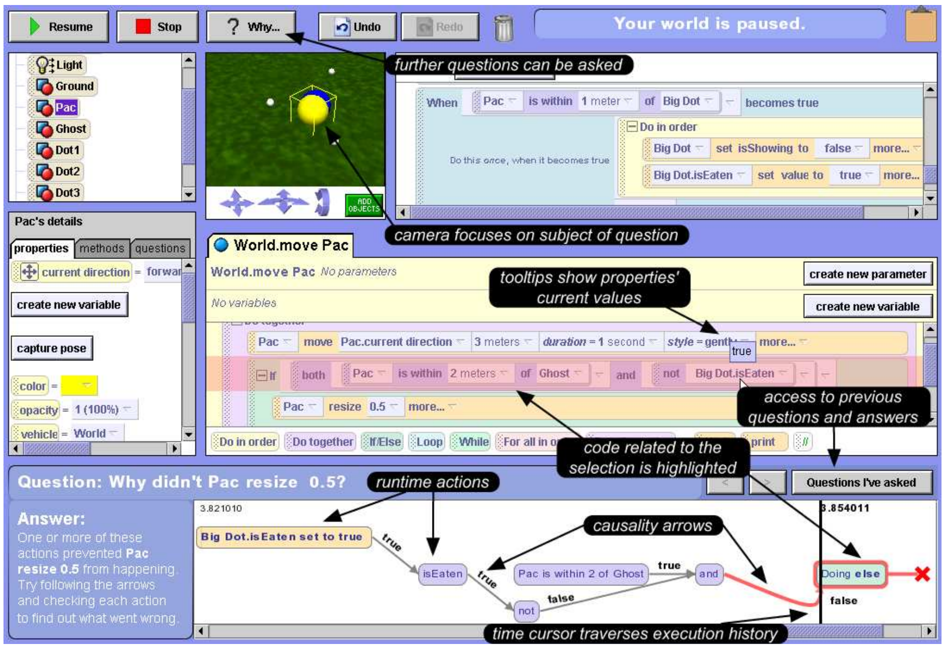


Figure 7. This diagram compares the mentioned programming systems roughly with respect to their difficulty of use, dependent on the complexity and sophistication of programs created with them. The thresholds of the systems are the corresponding y-intercepts, while the ceilings are the points at which the lines stop. Some systems have vertical walls, where the programmer "needs to stop and learn something entirely new" [10].

Overall, with good support by the development environment, it is possible to lower the threshold due to better usability, as well as to raise the ceiling, because it becomes easier to get to a higher level of complexity, which has been out of reach before due to insatisfactory tools.

3. CONCLUSION

We have seen a lot of already implemented but not necessarily perfect interfaces enabling users to create and modify applications for their special needs without any programming knowledge. Many of the shown concepts are already widely accepted and used in today's offices and homes—the spreadsheet calculator might be a good example for that.


further questions can be asked
camera focuses on subject of question
tooltips show properties' current values
access to previous questions and answers
code related to the selection is highlighted
runtime actions
causality arrows
time cursor traverses execution history
Resume
Stop
Why...
Undo
Redo
Your world is paused.
Questions I've asked

Question: Why didn't Pac resize 0.5?

Answer: One or more of these actions prevented Pac resize 0.5 from happening. Try following the arrows and checking each action to find out what went wrong.

Figure 6. When asking a "Why didn't"-question in Alice, the Whyline at the bottom of the interface visualizes the sequences of runtime actions which prevented an event from happening. The user can move the time cursor to navigate to earlier events; the code is highlighted accordingly.

Now, almost 20 years after Weiser's discussion on ubiquitous computing, industry and research institutes provide the necessary technology for arriving at a level that allows users to create and use computer software, in this our case mainly context-aware applications, in a simple way letting the computer itself proceed another step into the background. Both, CAMP and iCAP, enable end users to specify a rich variety of scenarios, with iCAP as a tool for the development of almost every thinkable application of this kind.

3.1 Comparison

We have seen a lot of sample applications and there are many more available using similar or different metaphors. We use the following criteria to compare the mentioned programming systems. Table 1 shows these aspects for all systems at a glance.

- **Relaxed syntax:** This applies to textual programming systems whose language does not have a strict syntax. This avoids the necessity to learn a strict language, allows for a relatively natural language, and thus yields a low threshold. Relaxed syntax is a major feature of Chickenfoot. This pattern can also be found in CAMP: Although the vocabulary is restricted, there is a great latitude of the arrangement of words. The HANDS syntax is also relaxed by allowing variations in the expression of queries, however, it does not go as far as in Chickenfoot.

- **Syntax-directed:** As described in 2.4.1, Alice has a syntax-directed editor. Since in CAMP the selection of words is restricted by the available vocabulary, and the arrangement is directed by the magnetic poetry metaphor, CAMP is also syntax-directed. The editors of the other textual systems, Chickenfoot and HANDS, do not feature any special support.

- **Visual Programming:** Of course all systems mentioned in 2.2 use the Visual Programming paradigm. Furthermore, the PBD system Stagecast Creator uses visual programming to represent before-after rules. Visual Programming is not to be confused syntax-directed textual programming: Although Alice and CAMP use graphical aspects to support the programmer, e.g. drag&drop techniques, the actual programming is text-based.

- **Support by Development Environment:** Since the syntax-directed aspect was already discussed above, we consider only additional aspects here. Chickenfoot and HANDS have almost no special support for editing and debugging. In CAMP one can at least find magnets easily by pressing their first letter, which speeds up the development. The Whyline extension for Alice is particularly noteworthy; it makes end user programs less error-prone and enables the end user to develop applications much more efficiently.

Figure 7 visualizes the connection between ceiling and threshold in the discussed environments. We can see, that in general low threshold and low ceiling go together. Environments with a relatively high ceiling and low threshold are usually restricted to a certain field, e.g. iCAP to context-aware applications. Complex applications cannot be created without a high threshold or high wall.

3.2 An outlook on future developments

What will the future bring us concerning end user programming languages? Certainly a lot. Users will still need to be enabled to configure their applications, operating systems, home theater equipment and even cell phones while the complexity of those devices is growing day by day. The development of programming environments like those we have discussed in this paper is ongoing. Most probably, there will be mixtures of the different approaches. Along with the research on new ways of human computer interaction like for example tabletop displays and gesture recognition, visual programming languages have a huge potential as they do not depend on keyboard inputs. Simpler debugging environment can then be integrated to provide a higher accuracy when constructing more powerful and complex languages.

We believe that even in office environments end user programming will play an important role in the future. Printers, scanners, copy machines, cameras, other capture devices, tabletop displays, wall-mounted large scale displays, handhelds etc. could be quickly interconnected and reconfigured by every employee to, for example, build up more powerful conference rooms and working environments.

On the ubicomp side, hopefully someday, all devices on the market will have a common digital interface to access their properties easily through a distributed dataspace. Together with end user programming languages, then there will not rest many obstacles on the way to the perfectly configured eHome.

4. REFERENCES

1. Farah Arefi, Charles E. Hughes, and David A. Workman. Automatically generating visual syntax-directed editors. *Commun. ACM*, 33(3):349–360, 1990.

2. Michael Bolin. End-user programming for the web. Master’s thesis, Massachusetts Institute of Technology, 2005.

3. Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. Automation and customization of rendered web pages. In *UIST ’05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 163–172, New York, NY, USA, 2005. ACM Press.

4. M.F. Costabile, D. Fogli, C. Letondal, P. Mussio, and A. Piccinno. Domain-expert users and their needs of software development. In *HCI 2003 End User Development Session Papers*. Network of Excellence on End-User Development, 2003.

5. A.K. Dey, D. Salber, and G.D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware application. *Human-Computer Interaction Journal 16(2-4)*, pages 97–166, 2001.

6. Anind K. Dey, Timothy Sohn, Sara Streng, and Justin Kodama. icap: Interactive prototyping of context-aware applications. *Proceedings of the Fourth International Conference on Pervasive Computing*, pages 974 – 975, 2006.

7. Jan Humble, T. Hemmings, A. Crabtree, B. Koleva, and T. Rodden. ’playing with your bits’: user-composition of ubiquitous domestic environments. In *Proceedings of the 5th Annual Conference on Ubiquitous Computing (UBICOMP 2003)*, Seattle, WA, USA, October 2003. Springer-Verlag.

8. Andrew J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *CHI ’04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158, New York, NY, USA, 2004. ACM Press.

9. Bil Lewis and Mireille Ducasse. Using events to debug java programs backwards in time. In *OOPSLA ’03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 96–97, New York, NY, USA, 2003. ACM Press.

10. Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000.

11. Brad A. Myers, Andrew J. Ko, and Margaret M. Burnett. Invited research overview: end-user programming. In *CHI ’06: CHI ’06 extended abstracts on Human factors in computing systems*, pages 75–80, New York, NY, USA, 2006. ACM Press.

12. Brad A. Myers, John F. Pane, and Andy Ko. Natural programming languages and environments. *Commun. ACM*, 47(9):47–52, 2004.

13. J.F. Pane and B.A. Myers. Tabular and textual methods for selecting objects from a group. In *IEEE International Symposium on Visual Languages*, pages 157–164, 2002.

14. John Francis Pane. *A programming system for children that is designed for usability*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2002. Co-Chair-Brad A. Myers and Co-Chair-David Garlan.

15. Alexander Repenning and Andri Ioannidou. Agentcubes: Raising the ceiling of end-user development in education through incremental 3d. In *VLHCC ’06: Proceedings of the Visual Languages and Human-Centric Computing (VL/HCC’06)*, pages 27–34, Washington, DC, USA, 2006. IEEE Computer Society.

16. G. Rothermel, L. Li, and M. Burnett. Testing strategies for form-based visual programs, 1997.

17. David Canfield Smith, Allen Cypher, and Jim Spohrer. Kidsim: programming agents without a programming language. *Commun. ACM*, 37(7):54–67, 1994.

18. Khai N. Truong, Elaine M. Huang, and Gregory D. Abowd. Camp: A magnetic poetry interface for end-user programming of capture applications for the home. In *UbiComp 2004: Ubiquitous Computing*, volume 3205/2004, pages 143–160. Springer Berlin / Heidelberg, 2004.