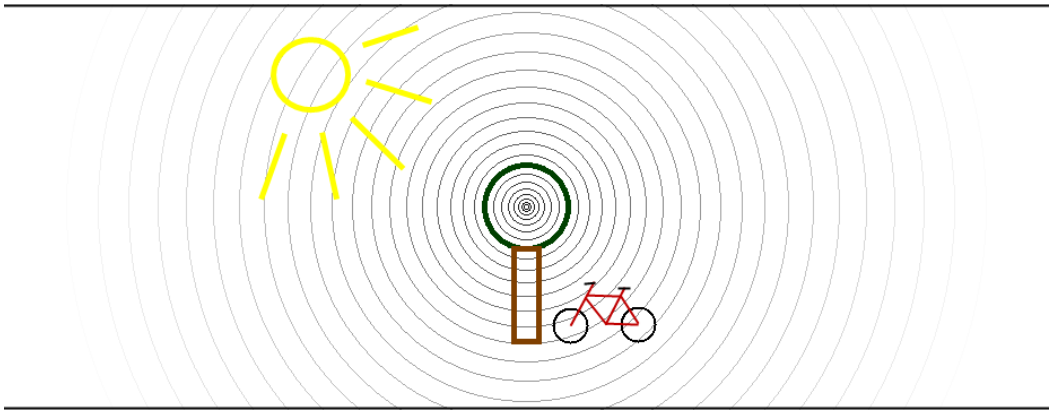
Master Thesis: Programming Technologies

End-user Programming

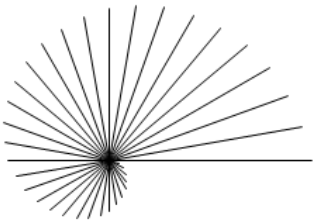


Aalborg University:

Software Engineering, SW10, Spring 2010

Anh Tuan Nguyen Dao & Peter Heino Bøg

June 14, 2010



Department of Computer Science
Software
at Aalborg University
Selma Lagerlöfs Vej 300
9220 Aalborg Ø
Telephone: +45 9940 9940
Fax: +45 9940 9798
E-mail: i16@cs.aau.dk
http://www.cs.aau.dk

Title:

End-User Programming

Theme:

Master Thesis

Programming Technologies

Project Unit:

Software Engineering

10th semester, Spring 2010

Project Group:

d609a

Participants:

Anh Tuan Nguyen Dao

Peter Heino Bøg

Supervisor:

Kurt Nørmark

Copies: 4

Page Count: 100

Appendix: A-G

Finished: June 14, 2010

Synopsis:

The theme of this project is end-user programming, and we focus on how to teach programming to end-users who had never programmed before. We develop a tool (prototype) as an extension to a drawing application along with a domain specific scripting language. The prototype uses the principles of self-disclosing in the concept of Learning by Observation. The goal is to teach end-users to program using the designed scripting language while they use the drawing application. Furthermore, we develop an algorithm to recognize iterative tasks such that a script could be generated to complete the tasks. The prototype and the scripting language are expansions of the work, done in our previous project, SW9.

We end with a usability test which proves part of our hypotheses, such as self-disclosing works as a learning concept. However, we did not prove this entirely for the group of end-users, we originally intended, but more for end-user programmers. Overall, the project and test are a success, and we find several interesting conclusions about the field of end-user programming.

Preface

This thesis documents a project made by the Software Engineering group d609a at the Department of Computer Science at Aalborg University. It is made on the 10th semester, SW10, and the group is part of the Programming Technology department. The theme of this semester is *End-User Programming*. The project period started at February 1st, 2010, and ended June 14th, 2010, and it is a continuation of the previous project, SW9.

The reader is expected to have a basic knowledge in programming and computer science in general. It is, however, not required to have read the previous project report from SW9.

Whenever the words "we" and "our" are used it refers to the authors of this thesis. Throughout the thesis we might use "he" or "his" in reference to a person, such references should be read as "he/she" and "his/her" respectively.

References to literature and sources in this thesis are written in square brackets, such as [nov], and references to figures and tables in the appendix are written with a letter and a number, such as Listing C.3. The bibliography can be found in Appendix A. The code developed for this project can be found on the appertaining CD-ROM, refer to Appendix F.

The order of the content in this thesis does not necessarily reflect the chronological order of our work during the project period.

We would like to thank our testers:

Allan, Bo, Carsten, Daniel, Helena, Joan, Kaj, Kaj, Kasper,

Kim, Lars, Luxshan, Martin, Peter and Simon

And an extra thanks to Allan for lending us his camera.

Group d609a

Anh Tuan Nguyen Dao

Peter Heino Bøg

4

Contents

Chapter 1

Introduction

Nowadays, we find computers everywhere, laptops, desktops, in washing machines, cell phones etc., and they are there to support us in our daily lives and works. Even with their presences all around us, it is far from all users who know what the computers are capable of; These machines are very good at processing information and doing repetitive tasks. Ideally, every computer users know how to utilize the full computer power, but this is far from the reality. The majority of computer users are *end-users* who are computer users with no programming skills. These end-users interact with the computers through computer programs (applications) with graphical user interfaces (GUI) provided by the software designers. It is like *Alice looking at the garden in Wonderland through a key-hole*, as phrased by David Canfield Smith et al. in an article about end-user programming [SCT00a]—a restriction for end-users.

While the restriction is a *wall* between the end-users and their computers, one goal with end-user programming research is to break down this wall by empowering the end-users with the programming skills. However, the general problem is that many end-users simply do not have time or are inclined to learn a programming language; others cannot see the need of programming, because the existing applications are sufficient to their needs. Programming by Demonstration (PBD) is a concept to help end-users obtaining the programming skills.

Another goal in the end-user programming research is to empower those end-users who already are on the other side of the wall, *end-user programmers*, with software engineering principles. End-users programmers are not novice nor professional programmers, they are programmers whose job functions are other than programming. The general problem with end-user programmers is the lack of reliability and security in the software

created by these programmers. A spreadsheet debug agent is a system to help end-user programmers creating more reliable spreadsheet programs.

In this project, we deal with the general problem of empowering end-users with programming skills. We propose a prototype of an existing concept, Learning by Observation (LBO), combined with the PBD principles, and we believe this prototype can teach end-users the fundamental programming principles. The prototype is aimed for end-users who do not have time to learn to program, but can also appeal to some end-user programmers.

1.1 Previous Work

This project is SW10 and is an indirectly continuation of the previous project, SW9. We did some experiments in SW9, and the result is an application which works as the basement for the prototype in this project. Figure 1.1 shows a screenshot of the original DrawTools application before our experiments, and Figure 1.1 shows how the same application looks like after our experiment.

DrawTools is a drawing application created by "Alex Fr" for a tutorial and released on *The Code Project*.[1] It is written in C# using Visual Studio[2], and has been chosen because of its simplicity in both design and interface.

[1] Link to the article on codeproject.com: `http://www.codeproject.com/KB/graphics/drawtools.aspx`.

[2] 2005 edition, but we converted it to 2008

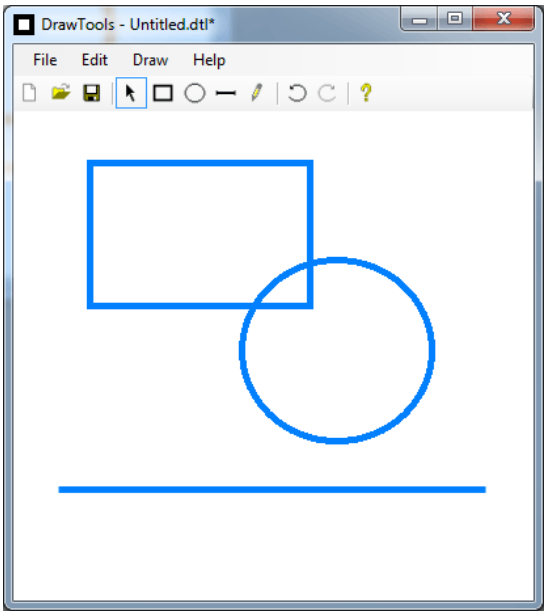


Figure 1.1: A screenshot of the original DrawTools application.

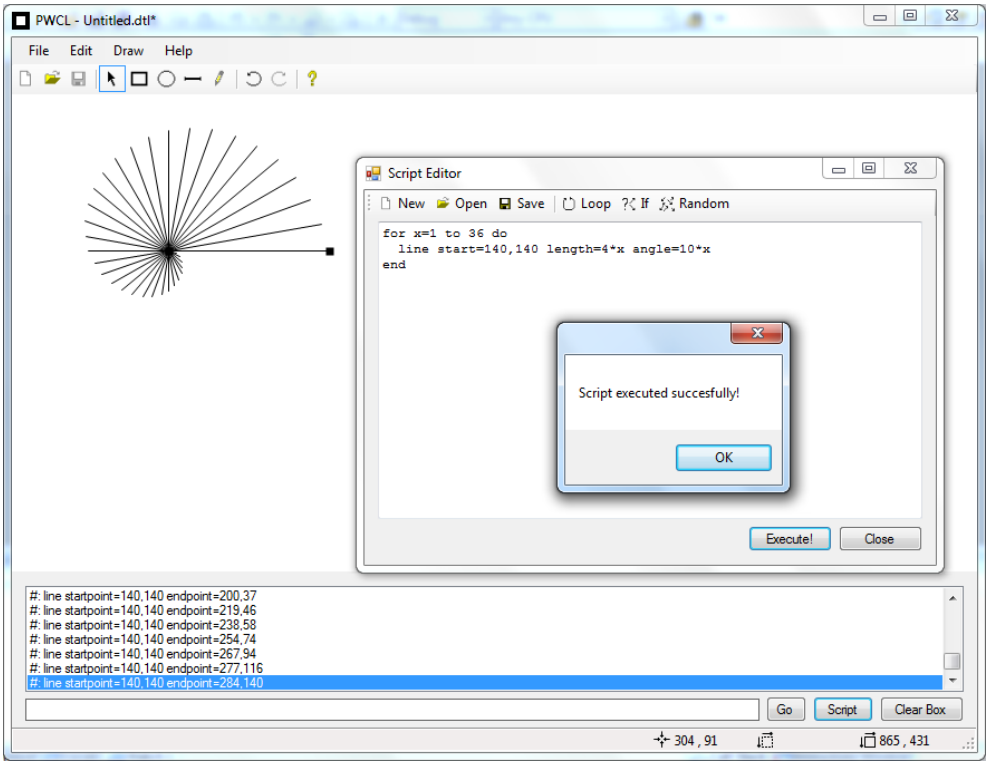


Figure 1.2: A screenshot of how the original DrawTools application looks like after our experiments in SW9.

1.2 Development Strategy

For the analysis, our approach is as following. We first make analysis in the area of end-user programming to get some background knowledge. The result of the analysis give us a problem which we try to find a solution for. For the solution, we make further analysis within the area of the found problem and looking at some related work.

The development process is iterative with incremental implementation of the prototype. We do not use a development method fully, but are still inspired by Scrum, because of its short iteration circuits (sprints). In the same way are we planning the development phase with design, implementation and internal black box tests.

To test our prototype, we use a usability testing method. Like the development phase, the test phase is also iterative with several testing rounds and implementation improvement phases in between.

1.3 Thesis Structure

The following is a list of chapters in this thesis with a description of what each chapter is about.

Chapter 2, Background, this chapter gives an overview of the area of end-user programming.

Chapter 3, Analysis, this chapter gives a more concrete analysis of the area with focus on parts that are related to our work.

Chapter 4, Problem Statement, the consists of hypotheses and delimitations for this project.

Chapter 5, Design, this chapter describes the design of our prototype in details. This includes new features and improvements of existing.

Chapter 6, Test, this chapter documents our preparation and results for the usability test of our prototype.

Chapter 7, Epilogue, in this chapter we conclude our work with this thesis. In addtion, we also make a critical evaluation of the project period, and putting our prototype into perspective regarding to the area of end-user programming.

Finally, we have several appendices which includes the bibliography.

Chapter 2

Background

In this chapter, we describe broadly the area of end-user programming as a result of our analysis in this area. We start with some definitions and usages to clarify some terms used throughout this thesis. Following, is a section about what an end-user programmer is along with different end-user programming approaches. We also describe problems which end-users are facing when learning to program, the so-called learning barriers. Finally, we finish this chapter with a section about two main end-user programming research threads.

Decisions, based on this chapter and related to our project, are reserved and described in the next chapter.

2.1 Definitions and Usages

The following sections we define some of the more important terms used throughout this thesis.

2.1.1 Programming

Before we can discuss about end-user programming, we need a clear definition of what we mean by programming. Different sources on the Internet define programming in many ways, such as *"the writing of a computer program"* (answers.com), *"creating a sequence of instructions to enable the computer to do something"* (thefreedictionary.com), and *"the act, process, or work of writing or developing computer programs"* (yourdictionary.com). All of these are repeated throughout the Internet, but are all too unspecific, e.g. requiring a further definition of what a program is, and finally it is not obvious where these definition

originates from.

Kelleher and Pausch [nov], on the other hand, defines programming as the following:

> "*The act of assembling a set of symbols representing computational actions. Using these symbols, users can express their intentions to the computer and, given a set of symbols, a user who understands the symbols can predict the behavior of the computer.*"

As they also mention in their article, this definition excludes many programming by demonstration systems (like some mentioned by Cypher et al. [CHK+93]). These systems use internal heuristics to generate a program for the user which makes it impossible for the user to accurately predict what program will be produced.

This definition is much more specific, and we feel this describe our own definition of programming the best.

In our case it is defined as *programming* to use the scripting language to do the job, but it is not *programming* to draw and let the application create a script to finish the job as the result of this approach cannot always be predicted accurately.

The set of symbols mentioned in the definition of *programming* can be seen as the *programming language*. However, some authors define a programming language as a machine-readable artificial language that is able to express all algorithms, meaning it is Turing complete. This excludes some of the languages and systems mentioned by Kelleher.

2.1.2 End-User

In the area of computer science, end-users are consumers of software applications. More generally, an end-user is *"the ultimate user for which something is intended"*, according to Webster's Online Dictionary.

If we consider software applications as tools for end-users, then programming languages are also tools and software in some sense. A programmer who uses a programming language created by another programmer must then be an end-user per definition. However, users who know how to do programming are not our primary target group, hence, we reconsider the definition of the term end-users.

Throughout this master thesis, we use the term end-users to cover users who know how to use a computer and applications which are within their job domain with no programming skills. As an example, a secretary needs to write letters and handle accountings, she uses a text editor and a spreadsheet application, but she does not know about other kind of

software applications. She is an end-user. A novice programmer who uses a programming language created by professional programmers is not an end-user.

2.1.3 Repetitive and Iterative Tasks

In general, the term iteration is a synonym for a repetition, but in the context of this thesis we draw a distinction between these two terms. We define an iteration to be a set of actions, and a repetition is two consecutive iterations. Notice that an iteration can consist of one action.

We then draw a distinction between an iterative task and a repetitive task. An iterative task consists of two or more iterations, and a repetitive task consists of two or more iterations with possible pause in-between the iterations. E.g. performing a search and replace of a certain word in a text document two or more times is an iterative task wherein the search and replace actions together forms one iteration. On the other hand, checking email everyday is a repetitive task. Generally, iterative tasks are a subset of repetitive tasks.

In this thesis we consider the problems related to tasks which are iterative. It means that a user can only complete these tasks by repeatedly performing an action or series of actions consecutively.

2.2 End-User Programming

This section gives an introduction of end-user programming. It starts with a definition of what an end-user programmer is followed by different kind of end-user programming approaches which all are different from the traditional semantic programming approach.

2.2.1 End-User Programmer

There are different kinds of programmers: Professional, novice and end-user [MKB06]. The professional programmer is someone whose primary job function is to develop or maintain software. The primary goal for these programmers is to use programming to develop the software program itself. These programmers typically have significant training in programming, e.g. with a bachelor in computer science or higher. The novice programmer is basically a professional programmer in training and will end up with the same theory knowledge as the professional programmer already have.

The end-user programmer is one who do programming, but not as the primary job func-

tion. Generally, the end-user programmers have no interest in the programming process itself, but use programming to support in their primary job such as accounting, designing a web page, doing office work, scientific research, entertainment, etc. [BAM06]. Such programming is end-user programming. For example, a teacher creates a spreadsheet to track the grades of the students, and a graphical designer creates a script to apply modifications to a large number of photos.

2.2.2 End-User Programming Approaches

The term *end-user programming* covers the area of programming which aims for end-users whose primary jobs are other than programming. Furthermore, programming in this context is not only meant as traditional semantic programming with general purpose programming languages like C, C#, Java etc., but can be as simple as altering predefined application preferences. The key purpose with end-user programming is to empower end-users with the ability to customize an application or solve computable tasks through programming, and thereby, obtaining better results or solve problems more effectively. In the following, we describe different end-user programming approaches, starting with the simplest one.

Preferences Programming

Preferences are predefined alternatives provided by the application designer and used to accommodate the need of different types of end-users and usages. They are an easy way for an application designer to add alternatives, but they easily get too complicated which sets a limitation on how many preferences there can be in an application.

Figure 2.1 is a screenshot of the preferences dialog available in the Opera Browser. It is a good example of how several preferences can be changed to fit the need of as many end-users as possible. Most preferences, however, only have a few prefixed options to choose from through either a drop-down selection box or a checkbox. This limits the amount of possible changes an end-user can really do. Figure 2.1 also shows that there are several other pages placed as tabs in the same dialog box. This amount of choices can be overwhelming to any user and even the programmer behind the application. Many larger applications have an even more extensive preferences dialog, such as Adobe Acrobat and Visual Studio.

In general, preferences give only a limited number of alternatives and is not generalized enough to be considered as traditional programming [CHK+93].

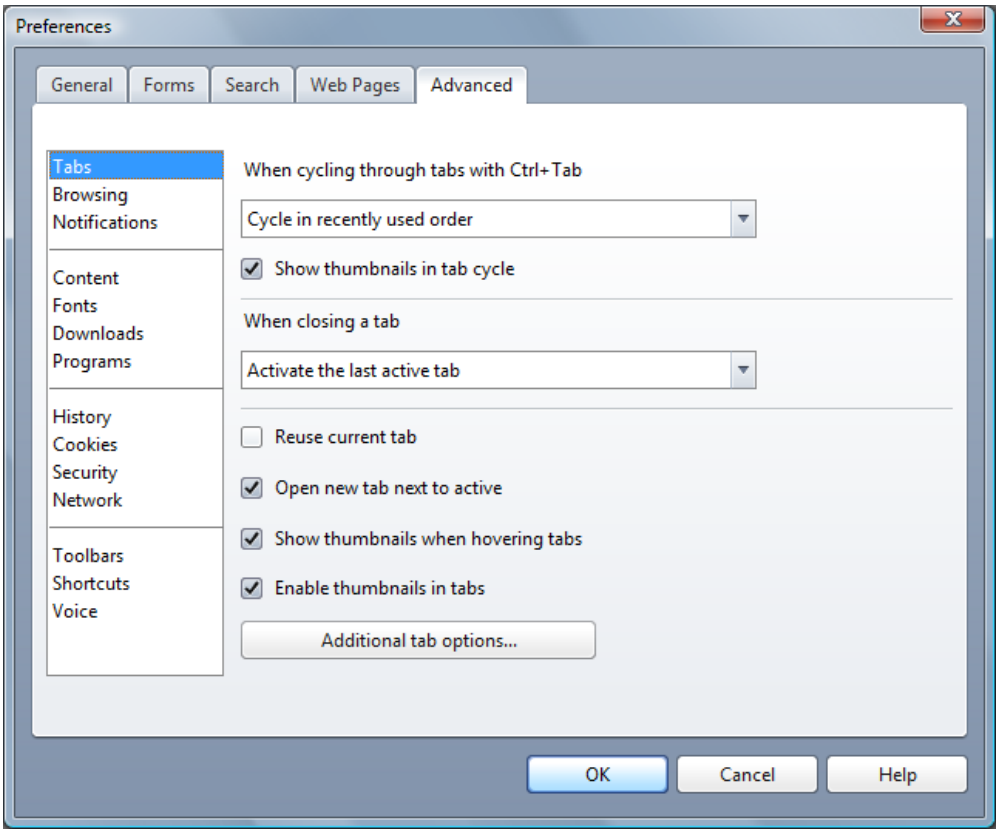


Figure 2.1: The preferences dialog available in the Opera Browser.

Programming by Demonstration

Macro recorders are used to record users inputs for later to repeat them. These recorded inputs are low-level events such as a mouse click at *position x,y* which makes it hard to reproduce the exact same effect, if for instance the end-user was clicking on a button at the *position x,y*, and the window of the application is moved when re-executing the macro. Programming by demonstration, also known as *programming by example*, is basically a macro recorder which instead of recording low-level events, produces a generalized program of mid-level events or user actions. So instead of *move mouse to position (x,y) and click*, it would say *click on button Z* or *invoke command of button Z*. In general, the end-user is teaching the PBD system what to do by showing the actual actions to the system.

An example of a PBD based application is Stagecast Creator created by David Canfield Smith, Allan Cypher and Larry Tesler [SCT00b] aimed for children to learn to program by using rule-based programming style. Stagecast Creator is a simulation game which lets the end-users create rules by demonstrating the simulation the rules, and by combining the rules the end-users can create small games or simulations. Figure 2.2 shows a picture

of a pirate game created in Stagecast Creator. In this game, the creator has defined some rules for when the pirate moves round on the map and what should happen when the pirate hits a chest—defining if-condition rules. The Stagecast Creator also supports use of variables which gives end-users opportunities to create more complex rules. Other similar applications as Stagecast Creator are ToonTalk[1] and Scratch[2] which are also aimed for children.

There exist two PBD systems, Eager created by Allan Cypher[Cyp93] and Familiar by Gordon W. Paynter[Pay00], which are not game simulations and aimed for a different group of end-users, namely adults. We will return to these PBD systems later in Section 3.2.

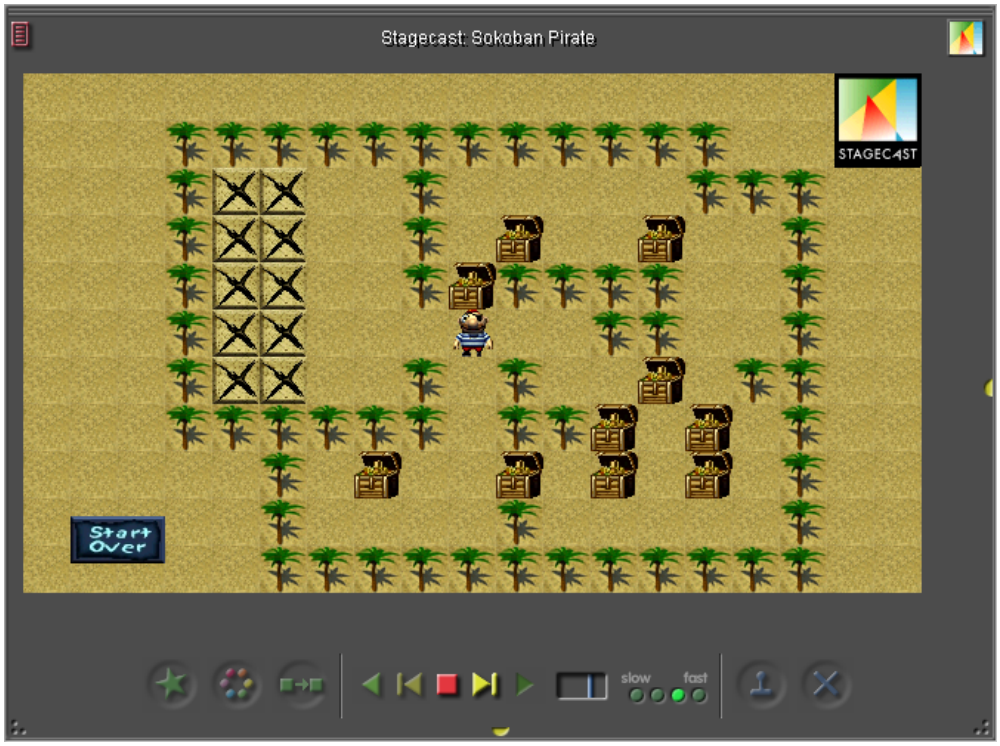


Figure 2.2: A screenshot of a pirate game in Stagecast Creator.

[1] More about ToonTalk at `http://www.toontalk.com`
[2] More about Scratch at `http://scratch.mit.edu`

Spreadsheet Programming

Spreedsheets are one of the most used systems in the world, both businesses and individuals use spreedsheets to create simple calculations or even large and complex financial models [RA09]. In spreadsheets end-users create formulas and build models in the form of functions and cell references, thus, a spreadsheet becomes a program in some sense. In essence, the end-users do first-order functional programming in spreadsheet programming [RA09]. Spreadsheet programming does not require any programming skills, since end-users create functions as mathematical expression and can switch between graphical interaction and programmatic interaction [DB09]. Furthermore, spreadsheets also constantly give feedback to the end-users as they make progress even when the program is not complete or contain errors. This means that the end-users can keep making progress without getting interrupted with compile and tests of programs as with textual programming.

Microsoft Excel is an example of a very popular spreadsheet application, shown in Figure 2.3[3] This example shows how rich Microsoft Excel is with functionalities for creating charts and models.

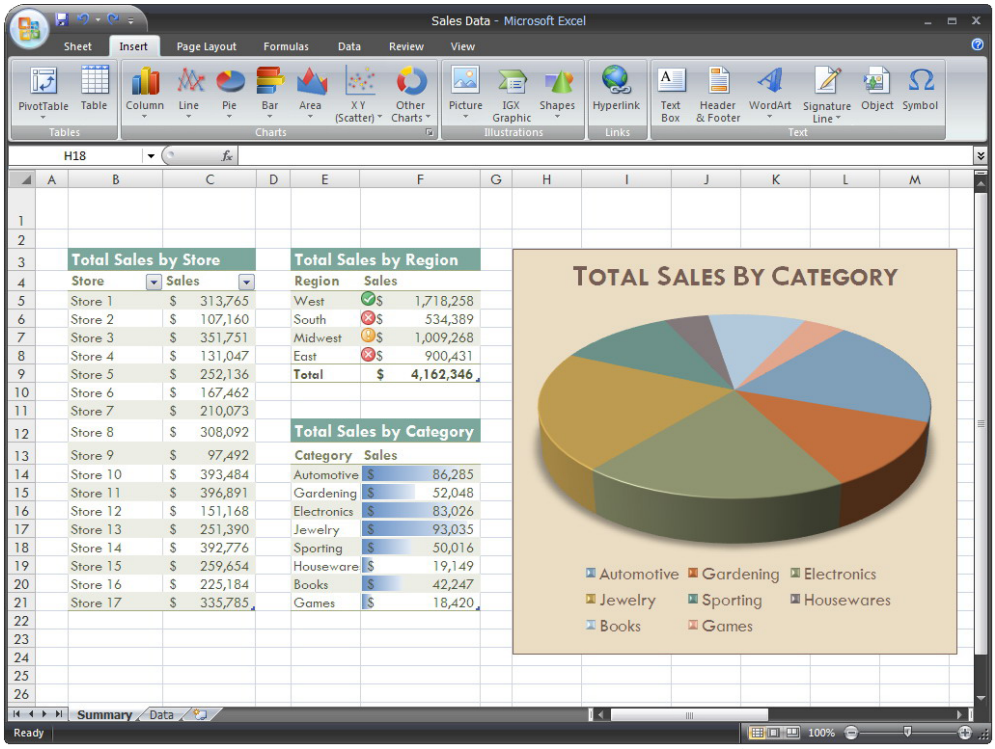


Figure 2.3: A picture of Microsoft Excel which contains a lot of spreadsheet related functionalities.

[3]This picture is found at `http://pubpages.unh.edu/~pit2/excel.htm`.

Script Programming

Script programming is when programming using a scripting language. A scripting language is a small language, often with a vocabulary specially designed for the application and its domain. These languages are also known as extension languages or embedded languages, because they differ from the application code. Scripting languages are the one of these programming approaches which is closest to a system programming language. Examples of such languages are Tcl[4], Lua[5] and Visual Basic for Application (VBA)[6]. Typically, these languages are tools for end-users to customize and control one or more applications and are less daunting than system programming languages, but can still be too complex for some end-users. This approach is considered as programming with some limitations within the application domain. E.g. scripts in Microsoft Word's VBA can only be used for functionalities within the application itself such as open a new document and apply a template.

John K. Ousterhout[Ous98] furthermore describes scripting languages as "glue languages". This is because they are also often used to glue together components that might have be written in a system programming language. One of the great advantages of using scripting languages for such a task is because they tend to be typeless. That way their interfaces do not need to be prepared for all kinds of types that the different system programming language have. Scripting languages are also usually not compiled, but instead interpreted. This of course reduces performance as the interpreter needs to do error checking every time the script is executed. It will, however, also decrease development time as the programmer do not need to compile before testing. This is an advantage for end-user programmers as they can see the result of their newly written code much faster.

Just-in-Time Programming

Unlike the previous programming approaches which are programming solutions, the just-in-time programming approach is a *situation* wherein an end-user can be and try to solve through programming [Pot93]. By programming solutions, we mean the different way of programming as described in the previous programming approaches such as rule-based programming in PBD system, first-order functional programming in spreadsheet programming etc. The fact is that any programming system or language can be used in just-in-time

[4]Tcl website: `http://www.tcl.tk`

[5]Lua website: `http://www.lua.org`

[6]VBA website: `http://msdn.microsoft.com/en-us/isv/bb190538.aspx`

programming, e.g. C, PBD or a scripting language.

This programming approach occurs when an end-user encounters a computable subtask and creates a program to solve it [Pot93]. The end-user creates the program, executes it and then discards it. When the subtask has been solved with the program, the end-user continue with the original task. The goal of just-in-time programming is to let the end-user programmatically solve a subtask of a main task, which is already in progress, more effectively. In other words, the end-user creates a program when needed—just in time programming.

2.3 Learning Barriers

Ko, Myers and Aung [KMA04] describes six types of learning barriers for end-users when they try to learn and use programming languages and their environments. They made a study of beginner programmers learning Visual Basic.NET as a base for these barriers. Following, is a list of these barriers where the headline (bold text) and the short description (italic text) are taken from their article, while we add our interpretation of it:

Design *"I don't know what I want the computer to do"*

Many beginner end-user programmers simply do not know how to look at a problem in an algorithmic way. This makes it impossible for them to even begin expressing the problem in a way a computer would understand.

Selection *"I think I know what I want the computer to do, but I don't know what to use"*

If an algorithmic solution has been found, the next big problem is how to express it in a certain language such that the compiler/interpreter can understand it. This includes using the correct components, structures and functions which might be difficult to find even if a help-system exists as the end-user might not now what to search for.

Coordination *"I think I know what things to use, but I don't know how to make them work together"*

This barrier exists when the end-user know what components to use, but do not know how to make them work together, e.g. as a result of wrong assumptions of the components behavior. One test-subject called it "the invisible rules".

Use *"I think I know what to use, but I don't know how to use it"*

The use barrier happens when end-users know what to use, but not how to use it.

This is often because the actual use of the component or structure is somehow hidden or obscure.

Understanding *"I thought I knew how to use this, but it didn't do what I expected"*

The understanding barrier often exists at compile or runtime when the code do not do as expected, and the end-user do not know what part of the code is right and wrong. At compile time this might be due to a bad respond from the compiler which might say it is missing a symbol at some line, but the end-user do not know where this symbol should be.

Information *"I think I know why it didn't do what I expected, but I don't know how to check"*

If an end-user has an idea of what might have gone wrong at compile or runtime, the end-user might have problems figuring out how to check for this. This could be because, the end-user simply does not notice different information given by the system, but the system might also lack these information all together.

Ko, Myers and Aung relates these six learning barriers to Norman's Gulf of Execution and Evaluation [Nor88]. The gulf of execution is the difference between a user's intentions and the available actions, while the gulf of evaluation is the effort of deciding if the expectations have been met. *Design*, *Coordination* and *Use* share characteristics with only gulf of execution while *Understanding* only shares with gulf of evaluation. The last two, *Selection* and *Information* shares characteristics with both gulfs. To bridge the gulf of execution Norman suggests that the system should have a visible constraints on what actions are possible, e.g. by explaining the "invisible rules" mentioned with the *Coordination* barrier. Bridging the gulf of evaluation Norman suggests that the system states are accessible and understandable relative to a user's expectations, e.g. by providing an extensive debugging tool which shows the content of each variable or otherwise illustrates what the program has done and is going to do.

The six learning barriers are related in such a way that fixing one of them might reduce how much another barrier occurs. Ko, Myers and Aung found that in Visual Basic.NET Selection, Use and Coordination barriers most often led to Understanding and Information problems which could imply that many problems related to the debugging phase actually originates from invalid assumptions of the usage of different components and structures. Locating these barrier relations for a system can therefore give a good illustration of where the main focus should be when changing the system.

2.3.1 Programming Barriers

Kelleher and Pausch [nov] defines a taxonomy for end-user systems. The taxonomy separates all of the systems into groups of what their main purposes are, starting with whether they are meant to teach the end-user how to program or if they are used to empower the end-user with a new tool to solve a task. They also explain what kind of problems or barriers exists within each of these groups. Furthermore, they describe two kind of more general barriers, and these are mechanical and sociological barriers.

The mechanical barriers are primarily about the programming language which the system is using. One common problem is that they are heavily influenced by the prevalent general-purpose language of the time of the system. Typically this means that C have influenced many systems from the 80s, Java have influenced systems from the 90s and C# have influenced recent systems. Many end-user programming systems have, however, been inspired by existing end-user systems such as Logo. This both include the basic concept behind the system, e.g. the Logo Turtle, and the language syntax, as a dialect of Lisp. The main problem with this is the fact that many of these systems are either too difficult for an end-user, or they get obsolete to quickly as a new prevalent general-purpose language arrives. Those systems which are not heavily influenced by a general-purpose language, have a different problem. They are too different which makes it more difficult for an end-user to make the move from the learning language to a general-purpose language. Kelleher and Pausch bring up the idea of an intermediate language between the learning languages and the general-purpose languages to make the transition easier.

The sociological barriers is about why some end-users choose not to program. The first reason is because, they do not see a reason to do so. This would require a advertising campaign rather than a system. The second reason is the social support. It is more fun and easier to learn in groups, but in the area of programming, it can be difficult to find such a group in the near vicinity of the end-user. Some systems try to handle this problem by creating online communities where end-users can share examples and their experience.

2.4 Main End-User Programming Research Threads

End-user programming has many research threads, and they all deal with the problem of end-user programming from different perspectives. In this section we introduce two main research threads and clarify which direction we cover in this thesis.

2.4.1 End-User Development

This research thread is also known as *end-user software engineering* [KW09]. It is not about turning end-user programmers into software engineers, nor does it ask them to think in software engineering terms in a software engineering cycle—requirements specification, design and reuse, test and verification etc. [Bur09, KW09]. The focus of this study is to develop methods to empower end-user programmers with the ability to create reliable software using software engineering concepts. The idea is to create *behind-the-scene* reasoning tools which monitor end-user programmers as they do programming. The tools functioning as guards against possible software errors and help the end-user programmers to detect and localize them. Furthermore, the target group of end-users for this research is those end-users who already do end-user programming.

Because of the huge number of end-user programmers, there is a potential danger with software programs created by these programmers, they riddle with errors [Har04]. Spreadsheet programming is widely used by end-user programmers [Bur09], and it is also in this programming approach where we find many examples of serious consequences caused by errors in spreadsheet programs[7]. Margaret Burnett, in her research of end-user development, propose a tool *WYSIWYT - What You See Is What You Test*[8] which help test and verify data dependencies in spreadsheet programs. This approach helps end-user programmers to be aware of errors in spreadsheet programs which otherwise are not visible—recall that in spreadsheet programming, the end-user programmer is not interrupted by errors.

2.4.2 Programming by Demonstration

There are two types of PBD systems which aim at different groups of end-users. The first type is *simulations* (Stagecast Create, ToonTalk), they are aimed for children, but study has shown that also adults find these PBD systems interesting [SCT00a]. The goal of

[7] Spreadsheet mistakes stories can be found at `http://www.eusprig.org/stories.htm`

[8] More about WYSIWYT can be found in [Bur09], [KJR00]

these systems is to learn end-users to think algorithmic while solving problems regarding creation of simulations and games in PBD environments. E.g. end-users indirectly learn if-statements when creating rules in Stagecast Creator. Imperical evidence shows that children who uses Stagecast Creator tend to continue with programming, and some children also turn to textual programming later on.

The second type of PBD systems is *recorder*, aimed for adults. The goal of these systems is not to learn end-users programming, but rather aid them in their daily work with a computer. Macro recorder, Eager and Familiar are examples of such systems. Unlike Eager and Familiar, the macro recorder requires end-users to implicitly demonstrate what it should do. It is therefore more appropriate to categorize the macro recorder as a simulation, but is mentioned under recorders, since we have chosen to divide the PBD systems after their goals.

2.4 Summary

At the beginning of this chapter we made several definitions starting with our understanding of what programming is. Afterwards we defined what an end-user and end-user programmer is and what the difference between repetitive and iterative tasks are. We then looked at different programming approaches such as preferences, programming by demonstration, spreadsheet programming and script programming. Here we also introduce the term of just-in-time programming.

The next part of the chapter described the different learning barriers that an end-user might encounter named design, selection, coordination, use, understanding, information and coordination.

Finally, we describe the two main threads in the end-user programming research area.

Chapter 3

Analysis

In this chapter, we describe and discuss about more domain specific topics regarding to our prototype. The chapter starts with a section about our decisions based on the previous background chapter. These decisions lead our focus on two main topics, Programming by Demonstration and Learning by Observation. We describe some related work, and how the work can be used for our prototype.

3.1 Our Focus

Based on the background chapter, we here take some decisions regarding to the end-user programming area. The decisions are to narrow down our focus on some topics related to our intentional prototype.

From the broad analysis of end-user programming, we find an interesting problem about end-users. Recall that end-user programming is about empowering end-users with programming skills or empowering end-user programmers with software engineering principles.

We choose to focus on end-users and not end-users programmers, but our goal is to turn end-users into end-user programmers. We have seen that PBD is aimed for end-users, and we, therefore, decide to look further into related work in this area. However, PBD is time consuming, and end-users are not inclined to learn to program, but this is due to the time required of the learning process. Therefore, we believe that a concept called Learning by Observation is the solution for the time consuming problem.

3.2 Programming by Demonstration

In this section we describe two systems that uses programming by demonstration and attempts to recognize iterative work patterns, Eager and Familiar.

3.2.1 Eager

Eager is an old tool created by Allan Cypher [Cyp93]. It is written in LISP and uses the programming by example approach by monitoring user activities in an HyperCard environment. Every action is monitored and when Eager detects a repetitive pattern in the user actions, it starts predicting the next action of the user. If the actions matches what Eager has predicted, it creates a program which will be executed and complete the task automatically. The goal of Eager is to help end-users solve repetitive tasks more effectively without doing textual programming. However, unlike PBD systems like Stagecast Creator or ToonTalk, the end-users use indirectly programming by demonstration. This means that the end-users do not explicitly create rules, like in Stagecast Creator, but rather just do their tasks.

Figure 3.1 shows an example of how Eager works. A user has a stack of messages each with different subjects (a), and the goal is to create a list with all subjects (b). When the user copies the subject from a message and paste it into the new list the second time (c), Eager will detect this repetitive pattern and start its prediction. It marks the right arrow (c), since it anticipates, the user will click there next. In (d) it anticipates, the user will copy the subject, type "3." in the subject list (e) and then paste the copied subject into the list (f). If the user then clicks on the Eager icon (cat face), Eager will create a program and execute it which will complete the task automatically.

3.2.2 Familiar

In Gordon W. Paynter's PhD thesis from 2000 [Pay00] he introduces a system called Familiar. It runs only on a Macintosh and uses AppleScript to record a users action and represent them as a script that can be altered. The goal of Familiar is to let an end-user record his actions so it can be used as the first iteration in a loop or just as a general macro that might be used later. Many of the examples Paynter describes involves working with folders and files in Macintosh Finder.

The actions that Familiar records are what Paynter calls mid-level events. He uses the

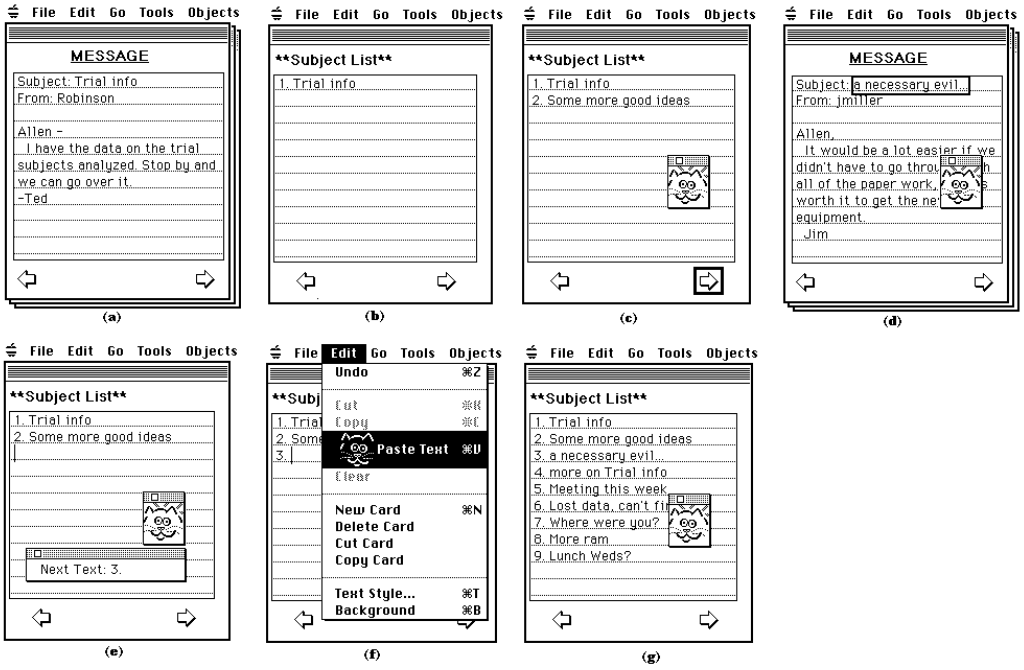


Figure 3.1: Example of how Eager works. This tool is created by Allan Cypher [Cyp93]

same kind of division in events as we did in our SW9 project[DB09], meaning low-, mid- and high-level where low is the individual mouse and keyboard events and high are very specific description of what has been done. The mid-level events are not as expressive as the high-level but more general such that it can be used for several situations. These events are represented in AppleScript and can therefore easily be used by both the end-user and the Familiar system.

Paynter uses machine learning *to guide prediction and infer conditional rules*. For this he uses the *C4.5 decision tree learner* described by Quinlan in 1993 [Qui93]. These decision trees are created from training examples which consists of one instance attribute value and several argument values. The instance attribute is the result from a prediction. Therefore the decision tree is constructed using the relations between the argument values with all leaf nodes being a value of the instance attribute. An example Paynter uses takes a list of matches that were either played or not depending on the weather conditions, such as temperature, humidity and wind speed. Each weather condition is an argument and whether the match is played or not is the instance attribute that can be one of two values: yes and no. A prediction is therefore done by traversing the tree using the known attribute values and ending at a leaf node.

3.2.3 Version Space Algebra

Tessa Lau, Steven A. Wolfman, Pedro Domingos and Daniel S. Weld [LWDW03] defines the problem of solving repetitive tasks using programming by demonstration as:

> "*A repetitive task may be solved by a program with a loop; each iteration of the loop solves one instance of the repetitive task. Given a demonstration, in which the user executes the first few iterations of the program, the system must infer the correct program.*"

They setup this as a machine learning problem, and have developed a framework that uses the concept of version spaces algebra to learn a machine complex functions and related inputs and outputs. Version space algebra is the authors own expansion of the version space concept described by Mitchell in 1982 [Mit82].

Version Space

Mitchell describes different approaches to the problem of formulating a general description of a class of objects from a set of examples, which he sees as a search problem. He divides the problem into three main parts:

1. Input observations (or instances) represented in some language, called the *instance language*.
2. Learned generalizations corresponding to sets of the instances and formulated in a second language, called the *generalization language*.
3. A *matching predicate* that matches generalizations to instances.

Using these the program needs to infer the identity of a *target generalization* using a set of *training examples*. These training examples are instances defined to be either part of (positive) or not part of (negative) the target generalization.

An important aspect of each of the approaches to solving the problem is the structure of the items in the generalization language. He uses the relation known as *more-specific-than*. This gives that every generalization is more specific than the previous, meaning it is a proper subset of the previous. This will give a tree of generalizations that can be searched through using different strategies: Depth-first, specific-to-general breadth-first, and version space.

In depth-first searching one generalization is selected as the current best hypothesis which for each new training examples is made more and more general. If it at some

point cannot be consistent with a training example the algorithm must backtrack and find a different hypothesis that does.

Breadth-first search will have several alternative hypotheses, thereby avoiding backtracking, but it also requires more checks for each new training example as many hypotheses might still be consistent.

In both of the previous strategies it is required to keep a list of all previous training examples and test them on any new hypothesis. The version space strategy avoids this by also removing hypotheses that are to general to be consisting with a training example. It can be seen as having both an upper and lower bound on possible hypotheses.

The Algebra

The version space algebra consists of the possibility to combine several version spaces using *join* or *union* and to *transform* one version space to another. As the theory behind the version space algebra is not relevant for this project we do not describe it any further.

The training examples used in version space to teach the machine would in our case be the users actions performed on the interface as part of an iteration.

Tessa Lau et al. illustrates the use of version space algebra by creating a small text editing tool they call SMARTedit. In this application they define the inputs and outputs to be application states of the form (T, L, P, S) where T is the text buffer (the whole text currently in the editing field), L is the caret location (R, C) consisting of a row and a column, P is the content of the clipboard buffer and S is the currently highlighted or selected substring which will always be a subpart of T. The machine is then fed a start state and an end state as the input and output. These states will be a combination of several user actions such that the difference between two states could be the complete movement of the caret to a new location or the complete deletion of a word. This will remove noise created by user mistakes such as moving to far or deleting to much.

The authors also raise the question of when an iteration starts and how to find it. They divide this in three forms of *segmentation*:

Full segmentation The user tells exactly when an iteration starts and when it stops.

Start segmentation The user only tells exactly when one iteration starts but not when it stops. Therefore the sequence of actions could contain more than one iteration and the last could even be only partial.

No segmentation The user never say when an iteration starts or stops. The machine will

here contain a complete list of all user actions since the start of the application and treat each of them as a possible start of an iteration. This will create a lot of version spaces and will eventually require a large amount of memory and CPU power to handle.

We believe that they treat the *no segmentation* option wrongly introducing to much overhead. There should only be created version spaces for action that could possibly have anything to do with the recent actions. This will continually remove unusable version spaces and only create new ones if no version space is currently available (else they would be part of some other possible iteration). The concept of version space algebra is only used for actually finding iterative tasks. The authors do not focus on how this could eventually be used for generating a script the end-user could edit and use.

3.3 Learning by Observation

Originally the CAD application AutoCAD were created by AutoDesk as a command line software where different commands, such as `rectangle` and `undo`, could be invoked to manipulate the drawing. Eventually AutoCAD adopted an interactive GUI where the same commands could be invoked using toolbars and the mouse. The command line still exists and DiGiano, Chris and Eisenberg[DE95] describes how a user learns how to use it by first using the toolbars and then observing how the commands are expressed in an output field. This concept of learning is called *learning by observation* (LBO) and it is not only limited to computer science. Children of both humans and animals do the same when growing up, as they replicates what their parents are doing.

3.3.1 Self-Disclosing

When AutoCAD display the commands invoked in an output field it is called *self-disclosing*. Self-disclosing is the term for when you expose some information about yourself and in the area of computer science it can more accurately be expressed as when you perform an action in an application and this application tells you more precisely what you have done. This is done in a language that the application can understand such that a disclosed command can be used again to perform the exact same action.

During their work DiGiano and Eisenberg[DE95] described three properties for self-disclosing programmable applications which enables learning by observation, such as Au-

toCAD. If these properties are fulfilled the self-disclosing of commands will work as what they call *context-sensitive help systems* which teaches the command language while the user is completing their original task. The following list of the properties were originally part of our previous report[DB09]. Digiano and Eisenberg formalized the properties as following (italic text) where we have given each a headline (bold text) and a short explanation of how we understand it:

1. **Disclosing for every action**: *For every elementary mouse action which has a command language analog, the system will disclose that expression to the user.*
   The display of the command in the output field for the user to see.

2. **Groups of disclosed expressions**: *The system will indicate groups of disclosed expressions connected with a single operation.*
   Any group of commands are visualized in some way, e.g. by using indention or by not writing a special command start as it is done in AutoCAD.

3. **Identical result from programming**: *Had the user entered the most recently disclosed group of expressions instead of specifying the operation through direct manipulation, the results would have been identical.*
   Writing the commands for creating a rectangle in AutoCAD with the same input would create the exact same rectangle as done with the mouse.

Digiano and Eisenberg also describes six guidelines for the pedagogical use of self-disclosure. These are formalized as following (italic text) where we have given each a headline (bold text) and a short explanation of how we understand it:

1. **Generalizable**: *Disclosures should be maximally generalizable.*
   E.g. consistency by objects in the language having a connection to objects in the application.

2. **Experimentation**: *The system should facilitate experimentation with disclosures.*
   E.g. by having the possibility to undo and redo, easily editable arguments and reinterpretation, and easily use of methods in other scenarios (like in a loop).

3. **Scaffolding**: *Self-disclosure should be scaffolded.*
   Incremental learning by e.g. gradually introducing more complex parameters for a method.

4. **Coverage**: *Disclosures should provide coverage of essential programming concepts.* As much as possible of the application (toolbars, menu commands, user actions) should be disclosed.

5. **Combination of access**: *Designers should be able to specify operations through a combination of direct manipulation and textual commands.*
   Like in AutoCAD where you can start the rectangle command through the command window and then select the corners with the mouse. This will gradually increase the knowledge of how to manipulate objects.

6. **Unobtrusive and browsable**: *Disclosures should be unobtrusive and browsable.*
   Meaning the disclosing should not interrupt the end-users work, but still be easy accessible e.g. through a history of disclosures (like the output field in AutoCAD).

We fulfilled all of these properties and guidelines, except for *scaffolding* in the previous semester, and we will continue fulfill them.

3.4 Discussion

Following is a discussion about what we have found in our analysis. We use this as a stepping stone towards our problem statement.

3.4.1 The End-User Problem

There are three main groups of end-users, those who want to learn to program and the programming process itself, those who want to use programming as an aid to their primary job function, and those who see no point or need with programming. End-users in the third group have typically no interests in computer or not more than necessary to use it as an entertainment source for gaming, film, music etc. Therefore, we omit this group in our discussion.

End-users who want to learn to program are inclined to use the time it takes to learn to program. These end-users are typically children, but can also be adults who like to create small games and simulations on their computers. For this group of end-users, the problem is less about the time required; the bigger problem is how to teach them to program. PBD systems like Stagecast Creator and ToonTalk are examples of simulations which let end-users create small games or simulations by making rules and manipulate objects. The

underlined purpose is to teach them programming principles, like the if-statement principle when they create rules for object.

For the second group of end-users who are not interested in the programming process itself, they are not inclined to spend time to learn it. This group is typically adults who need to do their daily work more efficiently. They consider the relation between the costs, risks and benefits of spending time on other things than their primary tasks. Investing time on learning to program can be very risky, even for a small specific task. The result may not reflect the actual intentions of the end-users, and that is a big drawback for the end-users, because their goal is to finish the task and not learning to program. In such situation, the benefits seems to be none. However, if the result is positive, not perfect, but positive enough in such a way that part of the task could be automated, that gives a huge benefit in the long run. The cost of time in total will be less than doing the task manually for larger tasks. PBD systems like Eager, Version Space and Familiar can help the second group of end-users to do their tasks more efficiently by providing a script to finish the task. However, the purposes of these systems are to help them solve the task more efficiently rather than learning them to program. Their focuses are on how to detect repetition and generation of scripts.

The Possible Solution

We propose a tool which uses PBD principle and the Learning by Observation (LBO) concept. Chris DiGiano in his work with LBO and self-disclosing pointed out that users of AutoCAD have gone from using mouse to draw to using commands after a period of time [DiG96]. This result is useful for us, as our goal is to help the second group of end-users to start programming, so they can, through programming, do their tasks more efficiently. Our idea is to introduce end-users to their actions as disclosures, so they can learn how to do the same actions by commands. Furthermore, we introduce the end-users to loop constructions by a PBD system which detects iterative actions and propose a script to the end-users as a disclosure as well. The end-user can use the auto-generated script to finish their tasks. Ideally, the end-users detect and understand the disclosures and thereby learning to program. The overall principle of our tool is to teach the end-users to program while they work in their normal working environment. They do not need to put away their work to learn to program, but can gradually get insights in the programmatic access to their working applications—they can do just-in-time programming.

3.4 Summary

This chapter started with a clarification of what part of the end-user programming area will be our main focus, turning end-users into end-user programmers. We then explore this area further, and we start with three programming by demonstration systems, Eager, Familiar and Version Space Algebra. All of these systems tries to recognize iterative tasks performed by the user, using different algorithms and approaches. Afterwards we explorer the concept of learning by observation and specially self-disclosing. Here we introduce three properties and six guidelines for applications that uses self-disclosing. The chapter ends with a discussion about the end-user problem which results in a description of a possible solution. This leads to the problem statement in the next chapter.

Chapter 4

Problem Statement

In this chapter we specify the problem we work with during the project period. First, we define a set of hypotheses we are going to prove. Secondly, we need to restrict the project to both what we believe is possible within the time frame of the project, but also to specify more precisely what parts of the end-user area we are working with. In Section 7.1, we give the answers to the hypotheses based on our usability test results.

4.1 Hypotheses

To help us keep focus on the problem, we are dealing with in this project period, we have defined two hypotheses. Following is a short description of each of the hypotheses.

Hypothesis 1: We can teach end-users, with basic computer knowledge, the fundamental principles behind programming by using the concept of learning by observation and an improved environment with several kinds of feedback.
We believe that the concept of learning by observation is a good approach for end-users to learn a scripting or programming language. The concept combined with an environment which contains several kinds of feedback such as syntax highlighting, command and argument information, abstraction to scripting, and debug and error reporting, works as an assistant to end-users. In the long run, they learn how to program by observing the disclosures, both as command and generated script, and the usage of abstraction to construct scripts.

Hypothesis 2: A task with iterative actions is completed more efficiently by an end-user who uses a script constructed by an automatic recognition of these actions.
This hypothesis is a continuation of the first one. By analyzing the actions of the end-user, we detect redundant actions within a task and predict how to automatically solve the rest of the task. The result of the prediction is a script generated by our iteration recognition tool. There are two usages of the generated script. One usage is to reduce the time an end-user spend on the task or solving it more efficiently by using such a script. The other usage is part of the learning by observation concept and uses it to teach end-users how to create loops.

4.2 Delimitations

In this project, we continue working with the drawing tool application, DrawTools, from the previous project. This section shortly describes our focus in this project.

4.2.1 Target Groups

We believe it to be impossible to learn end-users, who have no computer knowledge, the fundamental principles of programming. Likewise, end-users who have had extensive training in programing, such as novice and professional programmers, are not part of our target group either. This group may feel that different features used to help an end-user are actually annoying and are barriers. Therefore, our target groups are end-users with basic computer knowledge and end-user programmers who have tried only very little programming. We have depicted in Figure 4.1 how our prototype covers the end-user group and end-user programmer group.

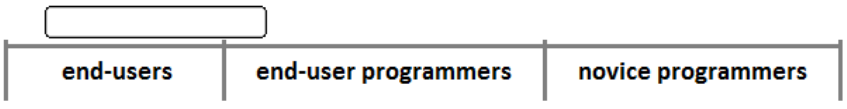


Figure 4.1: An illustration of our target groups for our prototype.

4.2.2 Iterative Task Recognition

We will expand our application with several new features, including iterative task recognition. The goal is not to create a general purpose iterative task recognition system, but

to create a system which is specific for the drawing tool application and for some specific iterative tasks within the application. In many cases, however, such iterative tasks could be transformed into another domain such as text editing, without major changes. Therefore we will have that in mind during the design, but not if the situation requires it we will choose a domain specific solution.

We have chosen three iterative tasks from the list of examples from the previous report [DB09] which our iterative recognition system must at least be able to handle.

- Drawing the same shape many times in a pattern.
  E.g. 360 lines in a circle or many rectangles inside each other. This will contain differences in positions and size that follow a pattern.

- Drawing a gradient.
  Fading from one color to another color. This will contain differences in color that follow a pattern.

- Drawing a specific figure several times.
  E.g. a night sky of stars. This will contain a set of shapes that relative to each others position follow a pattern, here forming a star.

In addition, we will also refine the visual feedback of our tool in the drawing application. However, the refinements are limited to code and syntax highlighting, code assist and error reporting. These features will improve the usability of our tool and will give better assistance to the end-users when learning to program. Even though these features are not our major concern in this project, we will still implement these to give a feel of completeness of our tool which will give better and more reliable test results at the end.

Chapter 5

Design

In this project we develop a prototype application to illustrate and test the ideas, we have for end-user programming. We call the application DASA which is an abbreviation for Drawing And Scripting Assistant, and in this chapter we describe the design of it.

In the previous semester we tried to leave as much of DrawTools, the original application, untouched as possible as we tried to create a sort-of plugin that could be used with any application. This semester, however, we decide to make several changes to the original DrawTools code simply to create a better application which we can use for testing some of our ideas for end-user programming.

To add new ideas and improve already implemented ideas, we also need to make changes to what we already added to this tool. Some of these changes includes the scripting language, Simple Disclosure, Command And Scripting Language (SDCASL), architecture and the addition of an algorithm for discovering and solving iterative tasks. In the following sections, we describe some of these changes and additions, we make to our prototype.

At the end of this chapter, there is a short description of the actual implementation of the design. It does not focus on the actual code, but on the different areas of the user interface, and how well we are following the self-disclosing properties and guidelines.

5.1 Scripting Language

In the following sections, we describe different aspects of our scripting language such as commands and language structures. We call the language SDCASL, which is an abbreviation of Simple Disclosure, Command And Scripting Language.

The bases for the scripting language were developed during the previous semester. Back then, we first tried to develop it for a command line and, therefore, it was only possible to express commands. Later, we turned it into a scripting language by adding common language structures such as if-statements, for-loops and variable assignments. These are more or less kept the same way in the current version, but other structures have been added since.

Because it was first made for a command line, we decided that each line in a script would only contain one command or structure element. This removed the need for an end-of-command character, such as semicolon (";"), which is used in several other languages. This will also remove a common error with missing semicolon made by an end-user with limited knowledge in programming.

5.1.1 Commands

A command is the same as an action, you want the application to perform. An example of a command is `rectangle` which will draw a rectangle. Such a command has a name, and often several synonyms. When the command is used in a script (or in the command line) it can be called by either its name or any of its synonyms. We have added the usage of synonyms for two reasons. The first reason is to make it faster to write a command that usually have a long name (such as `movetofront` which has the synonym `mtf`). The second reason is to make it easier for an end-user to guess a command and get it right. This is at first based on our guesses of what names end-users might think of, but after a testing round we might discover more possible synonyms. A good example of this is `drawline` which is a synonym for the `line` command, which actually *draws a line*.

Most of the commands can take arguments. In the last semester, we experimented with three kinds of argument approaches: Ordered, keyword and sloppy. These approaches are illustrated in Listing 5.1. Furthermore, we experimented with variable-length arguments which we do not explain further here, as it can be seen as an expansion of the ordered argument approach.

```
1 method ( width : integer, message : string, show : bool )
2
3 ordered ( 100, "hello", false )
4 keyword ( message="hello", width=100, show=false )
5 sloppy  ( false 100 "hello" )
```

Listing 5.1: The three argument approaches, ordered, keyword and sloppy, are here used to invoke the method with a parameter list defined as shown.

Ordered arguments is the most commonly used in programming languages. Here the definition of a method (or in our case a command) has a specific set of parameters in a certain order. When this method is called, the arguments must be in the same order and all of them must exist. By looking only at the code, it is in this case not possible to see what parameter each argument corresponds to. It is, however, possible in the keyword arguments approach. Here each argument needs to have a keyword specifying what parameter the value corresponds to. This also means that the arguments can be written in any order. The sloppy arguments approach requires a kind of heuristic algorithm which tries to predict what parameter each value corresponds to. It does this by looking at the type of the value. If there exists more than one parameter of the same type, we will always assign the value to the first parameter which have not yet any value assigned to it.

We tried and succeeded in creating the algorithm for the sloppy arguments, but after a few tests we realized that it simply was not as intuitive and helpful as, we first thought. Therefore, we changed it to use the keyword argument approach instead. One of the reasons for this, was the great information an end-user would get from seeing existing code, e.g. through the disclosures, which we explain later in Section 5.2.5.

When we had found the approach, we wanted to use, we made it possible for each parameter to have several synonyms as well. Again for the same reasons as for the synonyms for the commands.

In this semester, we will only make minor changes to the commands. First several commands had shortcut synonyms of only one character, e.g. `r` for `rectangle` and `x` for `exit`. Especially the last shortcut (x) gave us a problem as many users, including ourselves, would use a variable called `x` which would give unexpected results, because of a minor bug in the parser. This made us aware of it, and we decided that all of these single character synonyms should to be removed, and all other synonyms needs to be checked for similar often used variable names. Our language do support having variables and commands in different namespaces such that a variable can have the same name as a command, but we still felt, we had to change them as we expect, it could be confusing for

an end-user.

Other changes include the addition of new commands, restructure of parameters for some commands and a different order of some of the parameter synonyms such that more commonly used or more readable synonyms are used as the name of the parameter.

5.1.2 Language Structures

In the previous semester, we only added three common language structures: if-statements, for-loops and variable assignments. This provided the three elements required (conditional branching, looping and variables) for a language to be Turing-complete. In this semester, we have added one more structure: foreach-loop. For all of these structures, we used English verbs to make it easier to understand. This is instead of the more common use of different kinds of brackets in other languages such as C and Lisp. We considered using indention as a scope rule as well, like it is done in Python, but we decided not to as it increased the possible errors made by the end-user.

If-statement

A normal if-statement needs a condition, a body to be performed when the condition is true, and a body to be performed when the condition is false. The whole if-statement starts with the verb `if` which is also used in several other languages. For the same reason, the two bodies are separated by the verb `else`. It should be noticed that the else-part is not required. The whole if-statement is ended with the verb `end` which is also what ends the for- and foreach-loops. To make a clear separation between the condition and the first body, and also to make it look more like a regular English sentence, we have added the verb `then`. In this semester, we also need to allow the verb `do` here as it is used in all the other statements and therefore seem logical for some end-users to use.

The condition in the if-statement were designed to include both regular English verbs, but also characters commonly used in programming. Therefore, it is possible to express *and* using both `and`, `&&` and `&`, and *or* using both `or`, `||` and `|`. The single `&` and `|` are normally used for bitwise operations, but our language should not support such details, and we have therefore added them as condition operators for convenience. For the *is-equal-to* operator, we have added both single = and double ==. The reason being that we believe most end-users see a single equal as the most logical way of expressing it. We have also added two ways of expressing *not-equal-to*, `!=` and `<>`, as both are fairly common in other languages.

```
1 if color = red then
2   # true statements
3 else
4   # false statements
5 end
```

Listing 5.2: A basic if-statement in SDCASL with both bodies.

For-statement

A regular for-loop structure will increase the value of a variable by one for each iteration it performs. Some languages give the programmer the possibility to increase it by more than one, or decrease it or even handle several variables in the same loop. In the last semester, we decided to keep it as simple as possible, therefore the for-loop in our language can only increase the value by one. However, we still wanted to support changing the name of the counting variable, which gives the possibility to have several nested loops without any interference. Just as with the if-statement we have tried to have the for-loop sound like a normal English sentence saying that we want to count the value of a variable from the start value to the finish value. We make use of noise verbs such as `to` to increase the readability. The for-loop structure has not been changed for the current semester.

```
1 for i = 1 to 10 do
2   # statements
3 end
```

Listing 5.3: A basic for-loop in SDCASL counting the variable i from 1 to 10.

Foreach-statement

The new structure we want to add for this semester is a foreach-loop. The purpose of this structure is to loop through all drawn objects one by one. Together with special variables (see Section 5.1.3), this will give the end-user the possibility to check different values about each object and set different values accordingly. To avoid always looping through all the objects in the drawing, we want to add a variable keyword that express what type of object we intend to loop through. These keywords will be: `all`, `rectangle`, `ellipse`, `line` and `selected`. The selected keyword will make it loop through the currently selected objects no matter their type. We will add synonyms for these keywords as well as some end-users might want to write them in plural instead or use another word such as `square` or `circle`. Since it sounds a bit weird to say *foreach all do*, we will add the

structure `forall do` which does exactly the same. This structure is domain specific, but could easily be transformed to similar uses in other domains by changing the keywords.

```
1 foreach rectangle do
2   # statements
3 end
```

Listing 5.4: A basic foreach-loop in SDCASL that loops through all rectangles in the drawing.

5.1.3 Other Features

Our scripting language, SDCASL, have a few other features which we will describe here: Random, mathematical calculations, special variables and commenting.

Random

To have the possibility to draw an object at a random location with a random size or with a random color, we need a *random structure*. In most other languages it is done by calling a special function or instantiating a special object and then calling its methods. Our language does not support these approaches and instead, we added a special keyword (named `random`) in the previous semester. This keyword will generate a random number between 0 and the number written right afterwards, such that `random 100` will generate a random number between 0 and 100, both included. It is fairly simple to generate a number between to other values simply by adding the start number as an offset. So `25+random 50` will generate a random number between 25 and 75. We were aware that this might not be as intuitive for all end-users, and we therefore had to add an easy way to insert this keyword through interaction with the user interface where the user could specify the number range. This eventually made us add similar interface approaches for all the other structures in the language. For this semester, we have not changed the way the random structure works.

Mathematics

In the previous semester, the language did have the ability to express mathematical calculations using the very common operators of + (addition), – (subtraction), $\star$ (multiplication) and / (division) together with numbers (integers and doubles) and variables. However, the mathematical operations did not follow normal mathematical conjunction rules; Multiplication and division should be calculated before addition and subtraction. Instead it calculated each operator, one by one, from left to right, meaning that an expression like

$5 + 2 * 2$ is calculated to be 14 and not 9 as what would normally be the case. In this semester, we need to change this, and here we can look at how we did the conditions (in the if-statements) where normal conjunction rules between `and` and `or` did apply.

Special Variables

As a result of the added foreach-statement (see Section 5.1.2), we also need to add what we call *special variables*. These are variables that can be used just like any other variables in the scripting language, but instead of just containing a value, they will be linked to the currently selected object. One good example of a special variable which we need is `color` which simply contains the color of the currently selected object. If we then add a new color to this variable the color of the selected object should also change. When used together with the foreach-statement, we can e.g. check the color of an object and then change it to something else accordingly. As with almost anything else in SDCASL these special variables should also have synonyms for both convenience and helpfulness. It is worth noticing that all of these special variables will be domain specific, but the concept itself might still be useful in several other domains, e.g. the `color` in a text editor.

Comments

Almost all other languages have a way to add pieces of text to the code which the parser will ignore. We usually know these as comments, and it is a great way to describe what is happening in a piece of code that could otherwise be difficult to understand. In the first semester, we did not add any way to add comments to scripts which we will change in this semester. The main reason is the possibility to add explanations to generated scripts, and for example to scripts that can be of great help for end-user to understand the code. To follow the rules set for our language, only one element per line, we have decided to only add what we call *full-line* comments. Contrary to normal *inline* or *single-line* comments, it is not possible to add a comment after another statement in a *full-line* comment. To indicate that a line is a comment we want to use the hash character (`#`) as it is both readable and commonly used in several other languages. At the same time, `#` is not used for anything else in the language. Because we use the *full-line* approach, we do not need the use of *block* or *multi-line* comments.

5.2 Architecture

The following sections describe some of the more important parts of the architecture in DASA.

5.2.1 DrawTools Elements

One of the key elements of the original DrawTools is the *shapes* which you draw. A shape in this application is either a rectangle, ellipse or a line. They all only consist of a border around the edges (one edge for the line), and the border has a certain width (the penwidth) and a color. Unlike most other drawing applications, they do not have a fill color. DrawTools contains methods for handling selecting, moving and resizing a shape. We do not need to make many changes to these methods.

The shapes are drawn on a special *draw area*. It keeps track of already drawn shapes, and it also handles the undo/redo of draw actions. In this semester, we need to change the undo/redo such that we can easily undo the whole result of a script execution rather than just a single action. This feature will give an end-user the possibility to preview the effect of their script before deciding whether they want to keep it or not.

Finally, there are *tools* which is used for the actual drawing. These change the way the mouse interacts with the special drawing area. The default tool will handle selecting, moving and resizing shapes, while other tools will draw one of the shapes, depending on what tool has been selected. In later sections, we will describe changes, we have made to disclose actions, and these changes requires changing the tools as well.

5.2.2 Parser and Interpreter

In our application, we will have several places that need to parse a script and some of these need to interpret it as well. Because of this, we need to have both the parser and the interpreter to be as general as possible and accessible from anywhere. The solution to the latter requirement is to use a Singleton pattern. This is what we did in the SW9, and we do not intend to change that. But both the parser and interpreter need improvements. First, there will be several additions to the scripting language, as previously explained in Section 5.1, which of course both the parser and interpreter need to be able to handle. Second, the error reporting of both the parser and interpreter were lacking and especially the interpreter did not handle errors in a script well.

The error system needs to work in such a way that if an error have been found, a message describing the error, including the line number, should be generated and saved. If the error happens during interpretation, it should stop while it tries to continue during parsing to find as many errors as possible. The error description will have to be as expressive as possible, include suggestions on why it occurred and code examples of how it can be done correctly. Many error messages, we normally would see in an IDE such as Visual Studio, are simply too complex for most end-users with no training in programming or debugging.

In the design of the scripting language, we decided that each line could only contain one language structure. Therefore, the actual parsing is simply done line by line. Because of this, and how the rest of the language is designed, we can also make assumptions of what language structure a line contains simply by looking at the first word. If it is recognized as a language keyword, e.g. `if` or `foreach`, we can parse the rest of the line according to that. If not, then we can try to parse it as either a variable assignment (where a variable will be the first word), a procedure call (where the procedure name will be the first word) or a command (where the command name will be the first word). The parsing of each line will construct one *token* which contains the important data from the line and which can then later be used in the interpretation. E.g. an *assignment token* contains the name of the variable and an expression it should be assigned to, but not the '=' character or any whitespaces as these are not important for the interpretation. The expression is also parsed and saved as a *type* which will be described later in Section 5.2.4.

5.2.3 Commands

We developed the commands during the previous semester, SW9. Each command has a unique name and several unique synonyms, e.g. the `rectangle` command can also be written as `rect`, `rec` and `drawrectangle`. Each command can also have several parameter sets. These parameter sets are lists of *types*. To continue the example, the rectangle command have two parameter sets:

- Two *positions*, specifying the top-left corner and the bottom-right corner of the rectangle.
- One *position*, specifying the top-left corner, and two *integers*, specifying the width and height of the rectangle.

These parameters also have a set of synonyms such that e.g. the width of a rectangle could be written as both `width` and `w`. Finally, these commands can be executed to perform their

actions with the parameter set which have been set to be used. The rectangle command can only draw a new rectangle shape, but other commands can do several actions, such as the `set` command which can change both the color and/or the penwidth of the shapes.

For this semester, we need to expand these commands such that we can get more information about each of them, e.g. for parsing error reporting or the iterative tasks recognition algorithm explained in Section 5.3.

5.2.4 Types

SDCASL have special types that are not always directly mappable to native C# types. One example being a color which can both be expressed as a English color name (e.g. `blue`), a red-green-blue value (e.g. `255,37,128`) and in hexadecimal (e.g. `#FF2590`). Therefore, during the previous semester, we developed a type system that can handle these special types. This type system was originally used for the command line with sloppy arguments (see Section 5.1.1). Later, however, it was also used for the ordered arguments and then again for the scripting language.

The type system used in the previous semester had limitations, meaning that these types could only be used as the arguments for commands. To make it possible to use these types in variables and in the condition of an if-statement, we have to expand the type system. This requires the use of operator overloading in the application language C#. This will allow us to add two positions together or check whether a color is red.

Some types are aggregated by one or more types. A good example of this is a position which consists of two integers divided by a comma (,). In SDCASL it is possible to express these integers as expressions which includes the use of variables and mathematical operations. These expressions are parsed and *expression tokens* are created to later be interpreted. If the parsing fails, it means that there is something wrong with the script, and the parsing of the whole script should fail. All of this were done during last semester, but the mathematical operations did not follow normal mathematical rules as described in Section 5.1.3.

As with any other language the conditions of an if-statement can be evaluated to either *true* or *false*. When this condition is parsed *condition tokens* are created which are later used during interpretation.

5.2.5 Disclosures and Actions

In the previous semester, each shape was expanded such that it could disclose the creation and manipulation of itself. For two reasons, we need to change this. First, it is no longer only the creation and manipulation of shapes that should be disclosed, but also many other actions. Second, we need an easy way to add actions to our iterative task recognition algorithm (see Section 5.3) which is best done in the same way these actions are disclosed. Figure 5.1 is an illustration of how the drawing of a shape will create an action that contains all the information needed to both disclose it and add it to the algorithm.

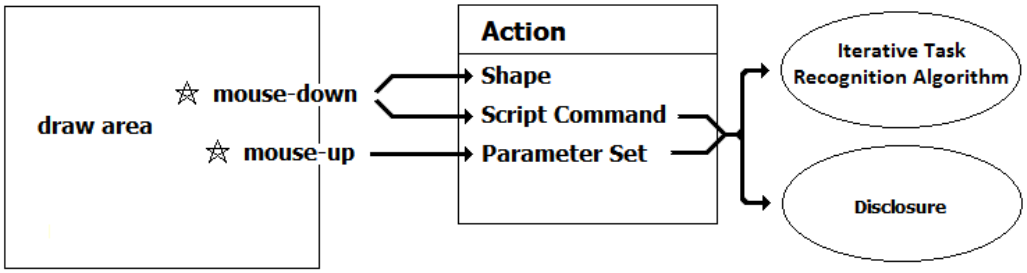


Figure 5.1: Illustration of an action performed on the draw area.

5.2.6 Procedures

Iterative tasks are the same task repeated continuously, recall the definition from Section 2.1.3. In programming, this can be done using a loop. *Repetitive tasks* on the other hand are the same task done several times, but with possible other tasks in between. Therefore, it cannot be solved using a loop, but instead by making an abstraction of it which then can be invoked when needed. This is what a macro recording in an application or a function in programming can be used for.Our iterative task recognition, described later in Section 5.3, will discover the iterative tasks and give a solution to them, but cannot handle the repetitive tasks. Instead we will introduce *procedures*.

A procedure in DASA can be seen as a small script that can be invoked with arguments. It should be able to create one from both the current script, but more importantly also from the disclosure box. When it is created from the disclosure box, it will contain a series of commands where the arguments can be generalized. The arguments to be generalized is positions, colors and penwidths. This makes it possible to draw new objects that have the same sizes as the original objects, but are located at a different position and with a different color and penwidth.

5.3 Iterative Task Recognition

In the following sections, we describe the design of our iteration recognition algorithm. First, we study the problem of recognizing iterations in a stream of actions. We then design the actual algorithm followed by how to generate a script to complete the iterations.

5.3.1 The Problem

In the following, we will describe the problem behind discovering iterative patterns in user actions.

Figure 5.2 illustrates how a tree have been drawn using a circle as the crown and a rectangle as the trunk. When we draw a second tree using the same construction, meaning the same type of commands, we have done a repetition which we want the algorithm to recognize. The input for the algorithm is then the commands shown in Listing 5.5. Each of these commands are used as input one by one, and not all together at the same time. The first two commands are part of the first iteration while the last two are part of the second iteration. Figure 5.3 is an illustration of this scenario with a longer history than just the four commands. For the algorithm these commands will be objects which contain the type of commands, the name that should be used in a script, and all its arguments and their names and values.

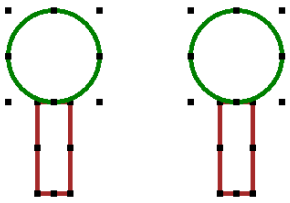


Figure 5.2: Two trees drawn using a circle and an rectangle.

```
1 rectangle first=132,200 second=168,300 color=brown penwidth=5 (first begins)
2 ellipse first=100,100 second=200,200 color=green penwidth=5
3 rectangle first=332,200 second=368,300 color=brown penwidth=5 (second begins)
4 ellipse first=300,100 second=400,200 color=green penwidth=5
```

Listing 5.5: Input for the algorithm. A circle drawn will be disclosed as the command *ellipse*. Command 1 and 2 are part of the first iteration while 3 and 4 are part of the second iteration.

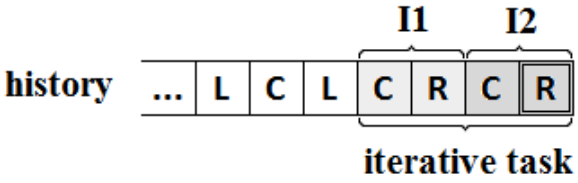


Figure 5.3: Illustration of the history, list of commands, and how the last four commands are recognized as the first and second iterations of a complete repetition which makes them part of an iterative task. Here for illustrative purpose L, C and R are the commands for drawing a line, a circle and a rectangle. Other commands can also be in the history.

The output from the algorithm should then be a script, using SDCASL. This script should consist of a loop which contains the two commands, `rectangle` and `ellipse`. The arguments of both of these commands need to use a loop variable to construct values that would construct the same two trees on the first two iterations. Following runs should construct new trees at the same distance as the distance between the first two trees. The end result is illustrated in Figure 5.4, and the script that constructs this is shown in Listing 5.6. In this situation only the two positions, which represents the upper left and lower right corners of the drawn object, are different.

```
1 for i = 2 to 10 do
2   rectangle first=i*200+132,200 second=i*200+168,300 color=brown penwidth=5
3   ellipse first=i*200+100,100 second=i*200+200,200 color=green penwidth=5
4 end
```

Listing 5.6: Script that is outputted from the algorithm. Notice that the loop starts on 2, as the first two iterations, 0 and 1, have already been done by the user.

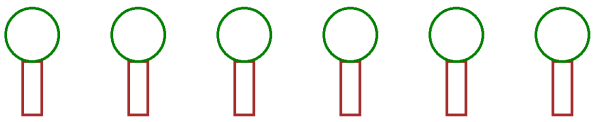


Figure 5.4: The outcome of running the script that was outputted by the algorithm.

In the following sections, we will refer to iterations in several situations. In the example above, the end-user has performed two iterations, each consisting of two commands. We refer to these as the first and second iterations. Since this algorithm only checks for iterative tasks, the first command in the second iteration will come right after the last command in the first iteration, in other words, no commands between the mentioned commands.

5.3.2 Iterative Tasks Recognition Algorithm (ITRA)

In this section, we describe how we design our own algorithm for discovering iterations and the generation of a script.

This algorithm is inspired both by Version Space Algebra, see Section 3.2.3, and Familiar, see Section 3.2.2. However, we do not use any of them directly as we find them too complicated compared to the our need for the prototype. The algorithm, we design is much simpler.

Algorithm

Before we can go deeper into what the algorithm does, we need to have a better understanding of the different elements of the problem. This algorithm is run each time a new command is added to a history of used commands. The history will start from the beginning of the application which Tessa Lau et al. calls *no segmentation* as the user will never specify when their iterative task actually starts (see Section 3.2.3). The history can be formalized as:

$$history = \{c_1, c_2, \ \ldots, c_m\}$$

where m is the total number of commands in the history, and c_m is the newest added command. The algorithm is looking for repetition in the form of two *complete* iterations, right after each other, which consists of similar commands.

Two commands do not need to be exactly the same to be similar as that would rarely be the case for drawing actions. Drawing the exact same circle on the exact same location have no extra effect. It is possible to do the same action with two different commands, e.g. drawing a circle with either the `circle` command or the `ellipse` command, recall that circle is a synonym for ellipse. Furthermore, we also have situations where the same command might have been used, but with different parameter sets which gives two completely different results. The `set` command is an example as it can be used to change either the color or the penwidth of a shape (or both at the same time). More formally, we can express similarity as $c_a \approx c_b$.

We call the two completed iterations for the *first iteration* and the *second iteration*, and they both consist of at least one command. We can formalize the iterations as[1]:

$$I1 = \{c_i, c_{i+1}, \ldots, c_{i+n-1}\} = \{I1_{c_1}, I1_{c_2}, \ldots, I1_{c_n}\}$$

$$I2 = \{c_{i+n}, c_{i+n+1}, \ldots, c_{i+n+n-1}\} = \{I2_{c_1}, I2_{c_2}, \ldots, I2_{c_n}\}$$

$$\langle I1_{c_j} \approx I2_{c_j}, j = 1 \ldots n \rangle$$

n is the number of commands in each iteration, which can be from 1 to $\frac{m}{2}$, and i is the index in $history$ of the first command in the iterations. This means that this first command is not necessarily the first command in the history ($I1_{c_1}$ is not necessarily equal to c_1).

For these two iterations to be complete, which is required for us to have discovered an iterative task, two rules need to be fulfilled.

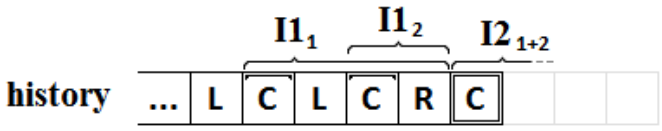
- The last command in the first iteration needs to be directly followed in the history by the first command in the second iteration. Or more formally: $I1_{c_n}$ needs to come right before $I2_{c_1}$ in $history$.
- The last command in the second iteration needs to be the last command in the history. Or more formally: $I2_{c_n}$ and c_m is the same.

Figure 5.5 illustrates the above formalization and how the possible iterations change when new commands are added to the history. It also shows how a new command can ruin complete iterations, but at the same time create new possible iterations.

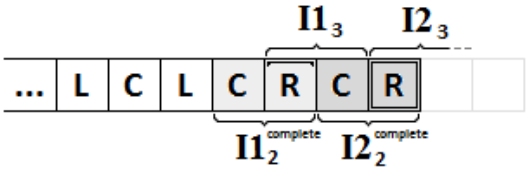
During the search for iterative tasks, we might find several possible complete first and second iterations. The most frequent cause being the continuous use of the same command. E.g. drawing six lines in a row will discover three different complete iterations, of one, two and three commands each, illustrated in Figure 5.6. For generating the script we will need to pick the most likely of all of these. There can be several ways to find this, but for simplicity we will only look at the number of commands in the iterative tasks and pick the one with the most (meaning the iterations with the highest n). Another approach could be to calculate a score based on the commands in use and the relationship between them.

Listing 5.7 is the algorithm expressed in pseudo-code. The `newCommand` is the performed command while the `history` is the history of all the commands performed (except for the newest). `possibleRepetitions` is a list of the currently *possible repetitions* found during previous runs of the algorithm.

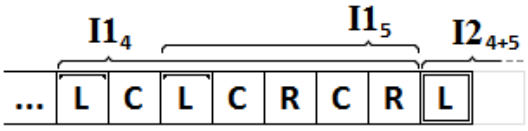
[1]The second formalization is for simplicity later on.



(a) The situation after a C command is added.



(b) The situation after an R command is added.



(c) The situation after an L command is added.

Figure 5.5: Example of the history, list of commands and how the possible iterations change when new commands are added. In Figure 5.5(a) a C command have just been added. Two possible first iterations ($I1_1$, $I1_2$) have been found as they both start with the C command. The iteration (L C L C) has previously been found, but is no longer possible, because of added commands. In Figure 5.5(b) an R command has been added, and the second possible first iteration ($I1_2$) from before is now complete ($I1_2^{complete}$). The other possible iteration ($I1_1$) has been removed as it is no longer possible. Furthermore a new possible first iteration ($I1_3$) have been found. In Figure 5.5(c) an L command has been added which make the complete iterations ($I1_2^{complete}$ and $I2_2^{complete}$) from before impossible, and they are removed together with the $I1_3$ which is also impossible. Instead two new possible first iterations ($I1_4$, $I1_5$) are found.

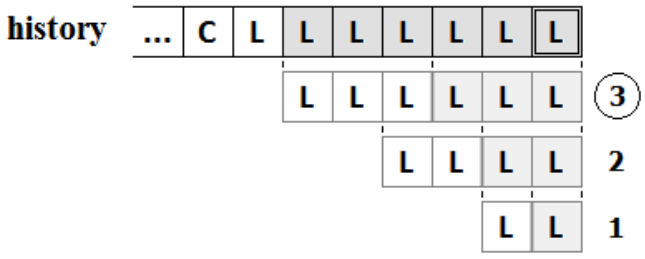


Figure 5.6: An illustration of how the continues use of the same command, here line (L), will produce several complete iterations. The one with the most commands (best score), here 3, will be chosen as the default repetition.

```
1  function AddRepetitiveCommand(newCommand, history, possibleRepetitions)
2    foreach oldCommand in history do
3      if newCommand is similar to oldCommand then
4        add new Repetition(firstIteration=oldCommand, secondIteration=newCommand)
             to possibleRepetitions
5
6    add newCommand to history
7
8    foreach possibleRepetition in possibleRepetitions do
9      if firstIteration in possibleRepetition overlaps secondIteration in
           possibleRepetition then
10       remove possibleRepetition from possibleRepetitions
11     if newCommand is similar to nextCommand in firstIteration in
           possibleRepetition then
12       if nextCommand in possibleRepitition is the last then
13         add possibleRepetition to finalRepetitions
14     else
15       remove possibleRepetition from possibleRepetitions
16
17   if finalRepetitions is not empty then
18     call GenerateScript(finalRepetition with best score)
```

Listing 5.7: Pseudo-code for the algorithm to discover iterative patterns.

Line 2 to Line 4 goes through the history of commands and finds any command that is similar to the newly added command. If one is found a new *possible repetition* is created with the old command as the first command in its first iteration and the new command as the first command in its second iteration. Line 2 to Line 4 goes through all of the currently possible repetitions found to see if they are still possible.

The check on Line 12 will be true if the two iterations in this repetition is *complete*. In that case, it is added as one of the *final repetitions*. A repetition is no longer possible if the new command can no longer be part of the second iteration or the two iterations overlaps (meaning they were *complete* when the previous command were added). It will be removed if that is the case. At the end the method `GenerateScript()` is called to generate the script for the repetition with the best score. This generation will be described in the next section.

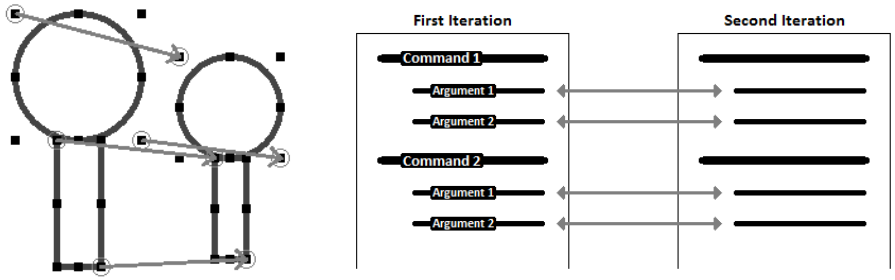
Script Generation

For generating a script to complete the found iterations, we need to go through two phases. First, finding the differences in the argument values for each command, and second, printing a script with a for-loop containing the commands where each argument value is an

expression which uses the found differences together with the for-loop variable.

Finding the differences can be done domain independent as each argument can be calculated without knowing what it is used for. However, knowing the domain can make some of these calculations easier and more reliable. In the following, we will introduce the comparison approach that we will use, we call it *normal comparison approach*. Afterwards, we shortly describe another approach which attempts to make a more correct comparison by using the domain.

In the normal comparison approach each argument of each command in the first iteration is compared to the same argument of the same command in the second iteration. Figure 5.7 illustrates this on a simple two-command iteration situation. The difference between the two arguments are then combined with a base value and a for-loop counter variable to express the correct value. This approach is domain independent as all the calculations behind it are done by the argument type objects and not the algorithm.



(a) Graphical illustration. (b) Command and argument illustration.

Figure 5.7: Normal comparison approach where each position argument is compared to the same position argument in the other iteration.

The main problem with the normal comparison approach is that the slightest error, which is very likely if the objects are drawn by hand, can result in a very unexpected drawing. This is because both the size of each object and its location is changed according to the previous iteration and not the other objects in the same iteration. Figure 5.8 illustrates this.

Copying Shapes

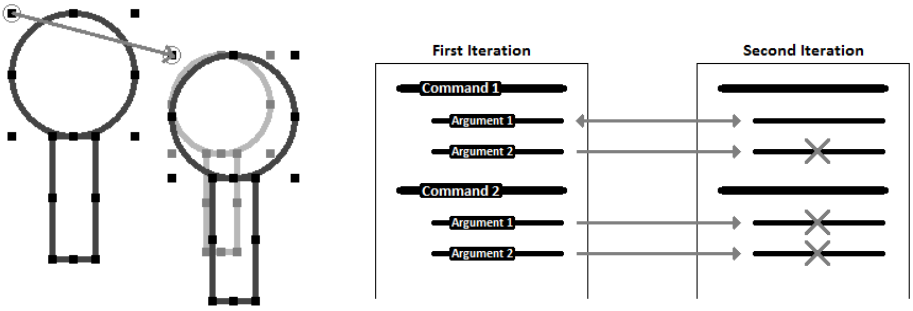
A very simple way to make sure that all shapes are exactly the same is to make a direct copy of the first iteration and use that in the subsequent iterations. This way any small flaw made during the second iteration will be removed. The second iteration is only used to get the distance between each figure. This approach is also domain dependent. Figure 5.9

Figure 5.8: Illustration of how a small error can become quite large after only a few iterations, when the script is generated using the normal comparison approach.

illustrates this.

For the user the result of copying would, however, not be as expected if these minor differences between the iterations were intended, e.g. such that the shapes would gradually increase or decrease in size. Therefore we cannot use this approach as the only solution, but it could be presented as an alternative.

If the user intended to create exact copies of the drawing, it is possible to use the list of disclosed commands instead and fairly easily create a loop. Of course, this is the purpose of ITRA, but it also raises the point that an exact copy, which fixes mistakes, might not be the user's first choice.



(a) Graphical illustration. (b) Command and argument illustration.

Figure 5.9: Copy comparison approach where only the first position from the second iteration (the small light grey tree) is used.

ITRA Improvements

We have found a couple of improvements to the automatic generation of iterative scripts which are not important to the actual algorithm, but will improve the user experience. Even though these improvements are domain specific, the ideas behind them can be used for any domain.

Noise, Knowing the Commands

When a user is using an application, it is very likely that that user will perform actions which in reality does not affect the final outcome, either because another action will overwrite this or because the action is somewhat empty. This is called *noise*. Therefore we need to normalize the command history, before the algorithm can work with it. This is done by removing noises.

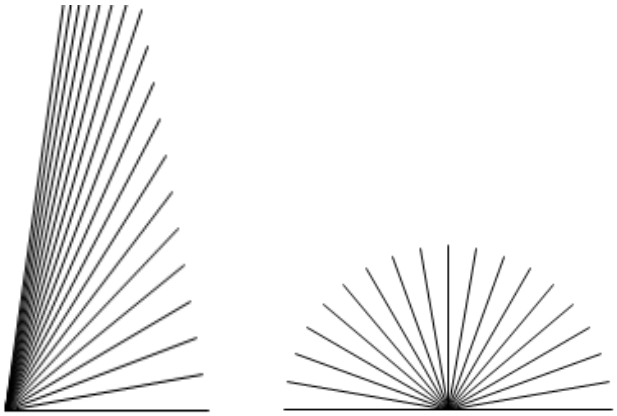
In Section 3.2.3, we described how Tessa Lau et al. removes noises by only using the first and last state of a sequence of similar states, such that the complete movement of the caret in a textbox or the complete deletion of a word is used. We can use a similar approach to detect and remove noise in our system, but just as with Tessa Lau et al., we can only do this for a specific domain.

There are two main reasons why an action can be classified as noise. Either it is overwritten by a following command, or it is empty. The most obvious example of a command that is overwritten, is the use of the `undo` command which will remove the previous command from the history. ITRA needs to detect this command (and its "sister", `redo`) and change its own history accordingly. The `set` command is used to specify the color and/or pen width of a shape. Several `set` commands in sequence may overwrite each other. The same is the case for the `select` command where new shapes will be selected every time it is used. Finally, a `select` command can be overwritten by the `unselect` command.

Empty actions are action that do not change the state of the application. The best example of this is the `unselect` if it is used when no shape is currently selected. This includes situations where it is used several times in a row, but also when the user tries to select nothing which will be treated as an `unselect`.

Alternatives, Knowing the Attributes

Some commands have different ways to handle their action depending on what kind of arguments they receive. An example of this is the `line` command to draw a line. It can either be done by specifying a start position and an end position, or by specifying the start position, a length and an angle. A script to solve an iterative task using the `line` command will look very different depending on what set of arguments are used. Figure 5.10 illustrates this in two different situations: Drawing a circle of lines, and drawing several parallel lines. The generated scripts can be seen in Appendix D.



(a) (b)

Figure 5.10: Illustration of how two sequences of line commands would produce different results depending on what arguments where used in the algorithm. Figure 5.10(a) uses the start and end position arguments while Figure 5.10(b) uses the start position, length and angle.

A good solution to this problem would be to create both scripts and present them to the user. Deciding which might be the best default choice could probably best be accomplished by using version space algebra.

5.4 Minor Design Improvements

In this section, we describe the design of different minor elements, we add to our application to increase usability.

5.4.1 Color Picker

The original DrawTools had a properties form where the user could set the color and pen width of the currently selected and future objects. In this semester, we will add individual forms and buttons on the toolbar. These buttons will illustrate the chosen color and pen width by simply showing them. The selected color will be shown as the background of the button with the text *Color* as the caption. However, there is a problem if both the background and foreground colors are very dark or very bright, therefore we will use a small function which calculates how bright the selected color is and sets the foreground color to either black or white accordingly.

5.4.2 Syntax Highlighting

Having syntax highlighting in a scripting box will increase readability. Having a color that expresses errors in the code as well will also increase writeability. Each line in SDCASL can only consist of one language element, and therefore we have decided to color the whole line instead of just keywords. Another reason for this is to illustrate larger blocks of code instead of small pieces that not necessarily seem to have any connections. This will improve the understanding of how the different language structures are expressed and separate commands from other structures and vica versa.

The actual color used for the different language elements should use mainly blue and green colors as a red color is used to illustrate errors. The reason being that red is often interpreted as danger, and is usually the color used in computer science to express a problem.

5.5 Implementation

In this section, we describe the final implementation of design. In order to keep things simple, we will not go into details on how our ITRA, parser and interpreter is actually implemented, but instead focus on the different areas of the user interface. Figure 5.11 is a screenshot of the final application as it looked in test round 2 (see Chapter 6). Figure 5.12 illustrates the different main areas of the application interface, and the following is a description of each of them:

A The drawing area where shapes are displayed, and where users can draw them manually with the mouse.

B The command line where users can write and execute single commands.

C The disclosure box where all actions performed are disclosed. A second tab shows parsing errors encountered when executing a script.

D The procedure box where all the created procedures are shown and can be manipulated.

E The scripting box where users can write scripts and execute them. The toolbar have several buttons to easily add different language structures to the script.

F The generated script box where the iterative task recognition algorithm discloses scripts it creates.

The following changes were made to the interface after test round #1:

- The question mark button was added to the command line (B).
- The buttons at the procedure box (D) were added.
- The toolbar of the scripting box (E) was changed and expanded.
- The buttons of the generated script box (F) were moved, and the box itself was colored yellow instead of gray.

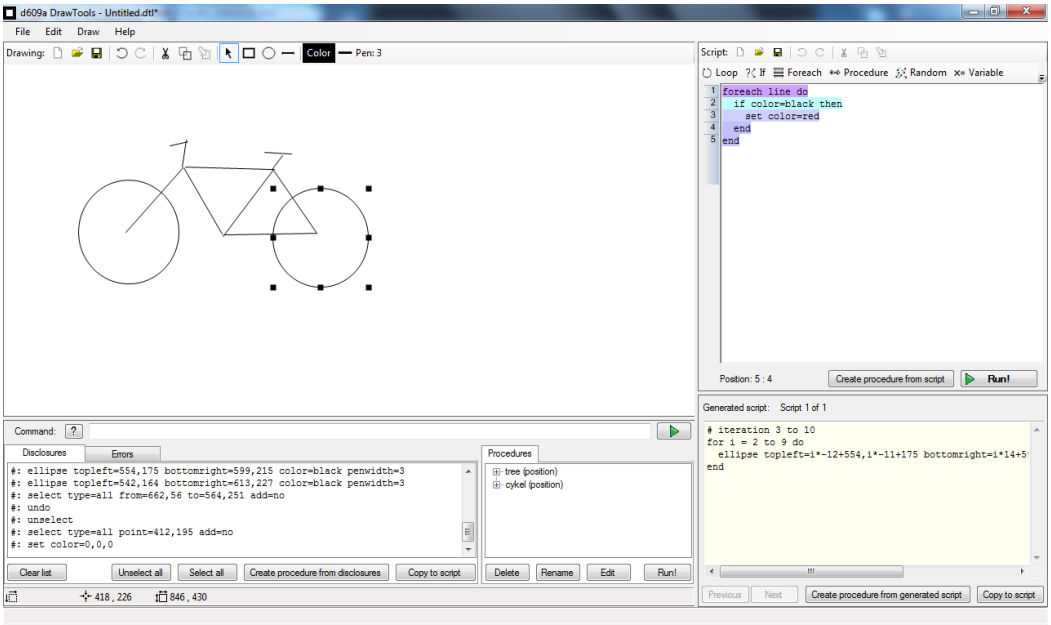


Figure 5.11: Screenshot of the DASA as it looked after test round 2.

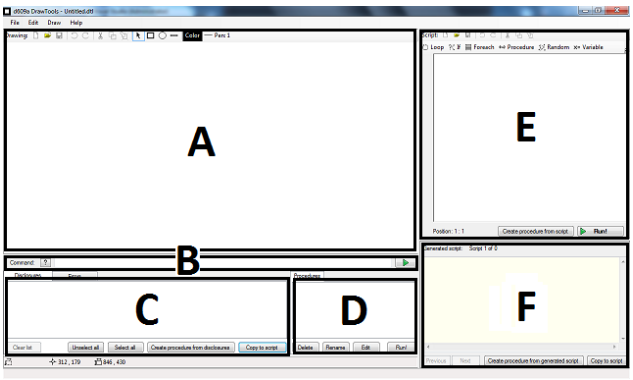


Figure 5.12: The different areas of DASA. A is the drawing area, B the command line, C the disclosure box, D the procedure box, E the scripting box and F the generated script box.

5.5.1 Self-disclosing

In this section we will describe how the final application fulfills most of the properties and guidelines for self-disclosing, see Section 3.3.1. Most of these were already fulfilled in application from the previous semester, but have been improved for this semester.

Properties

- **Disclosing for every action**
  Even more actions than the in the previous semester are now disclosed, with meaningful arguments.
- **Groups of disclosed expressions**
  The few connected actions there exists in DASA is combined into one command instead of being disclosed as a group.
- **Identical result from programming**
  Everything is disclosed in exactly the way the command should be written in the command line or a script to get exactly the same result.

Guidelines

- **Generalizable**
  The commands have names such as `rectangle` which can be directly transformed to a rectangle in the drawing.
- **Experimentation**
  The application have undo and redo features that allow the user to try out a command and then revert the result. This feature will undo the execution of a whole script as well.
- **Scaffolding**
  As the application is used, more and more will be learned through the disclosures. However, what is disclosed will always use the same synonym for a command and the arguments as we believe it to be confusing for the end-user if the same action would suddenly disclose something differently.

- **Coverage**

  Almost everything performed with the drawing area of the application will result in a disclosed command. This is the actions performed that would make sense to use in a script. Disclosing certain actions such as the creation or deletion of a procedure will only confuse the end-user more than providing a useful feature.

- **Combination of access**

  Most actions which requires more than one mouse gesture can still be done by only one command, but the complete drawing can be done with a combination of mouse gestures, command line, procedures and scripts.

- **Unobtrusive and browsable**

  Both disclosure and the generation of a script is done without interrupting the user in any way. If more than one script is generated all of them can be seen, and all disclosed commands will exists

5.5 Summary

In this chapter, we have described how the scripting language, SDCASL, has been designed in the previous semester, and what additions and improvements we needed to add in this semester. These included addition of new commands, improvements to existing commands and addition of `foreach` loop and special variables.

Furthermore, we described the improvements made to the architecture of DASA in order to accommodate the new features such as ITRA. The purpose of ITRA was to recognize iterations and generate a script. However, there were several problems, like noise and attributes comparison, which we discussed how could be solved.

Finally, we described primarily different areas of the implemented user interface in DASA, and how well it followed the properties and guidelines for self-disclosing.

Chapter 6

Usability Test

In this chapter, we first describe the purposes and our goals with the usability testing. Based on the purposes and goals, we describe some testing methods which we are going to use. We then describe the approach and the preparation for the usability tests. Finally, we document the test results along with the general feedback from the test participants, and the result are discussed with relation to the end-user programming and DASA.

6.1 Purpose and Goals

The overall purpose of our usability testing is to achieve test results which can make base for the proof of the hypotheses. We are interested in showing that the concept of learning by observation as used in DASA can teach the end-users to start programming or using programming to solve their tasks. Moreover, we want to see if DASA can be of any help or has any effect on the way end-users interact with an application—that is, changing their work habits.

Furthermore, we want to see how well DASA and the chosen level of abstraction match different groups of end-users. We also have different expectations for each group of end-users which are described in Section 6.4.2. Additionally, we partly focus on how learning by observation and DASA can best be presented to the end-users. In the long run, we expect to get some insights of how end-users, when they use programming to solve a task, think and how they go about the transformation from a solution to a piece of code.

6.2 Testing Methods

Based on our goals for this usability testing, we use a combination of different test methods. In the following, we describe the chosen methods and argue for the choices.

6.2.1 Pre-Test

Before we start the actually usability test, we make a pre-test with a random selected test person with knowledge about programming. This pre-test is a black-box test with the purpose of finding possible bugs. The test person simply try out DASA. The primarily focus is on finding and fixing possible bugs in the application before the actual usability test.

6.2.2 Usability Test

For the testing of DASA, we use usability testing[HS07], because DASA is a GUI-based tool which makes usability testing very suitable. This kind of test is, however, very time consuming regarding to test preparation, the test itself and test result analysis; All this sets the limitation on how many test persons, we are capable of using for our test, and we need to choose carefully the right test persons. Instead, we can choose to make an open beta test which requires less preparation, and we can reach out to more test persons. In such a test, we can simply put DASA on a website with a feedback form, and then everyone can participate in the test and give feedbacks through the website.

One of our goals for the testing is to get insights in working behavior of the end-users. With an open beta test, we cannot follow or monitor the test persons during the test. We need to be able to observe how the test persons react or think when they tumble upon a problem and try to solve it by programming; Why it did or did not worked. These observations cannot be obtained by using an open beta test. The test persons can even omit to submit feedbacks in such a test which is a loss of potential valuable test data. Another problem is that we need to test DASA on some specific group of test persons which we cannot be sure to get in the open beta test. In other words, we lack control in an open beta test.

6.2.3 Interview and Questionnaire

There are mainly three kind of interviews[HS07]: open-ended, structured and semi-structured interviews. In an open-ended interview, the interviewer is given the control of the interview by asking very opened questions. There is no agenda for an open-ended interview. As the name imply the structured interview are very structured. Each questions is planned upfront, and the answers are to be about these questions only. Finally, in a semi-structured interview there are some topics which must be covered, but within these topics the interviewee takes control of the interview.

A questionnaire is similar to an interview, but has no interviewer. It can consist of open or closed questions. The test persons answer the questions alone by themselves which removes the factor of influence by the interviewer. The results of this technique is democratic, because all the test persons will have exactly the same questions and the same answer options.

We choose to use an open-ended interview, because the test persons can give their personal opinions about DASA. In addition, we use a questionnaire to check if the test persons have learned or remembered anything from the usability test.

6.3 Approach

The test phase is structured as three iterations and consist of a pre-test and two usability test rounds; We call the testing iterations *pre-test*, *test round 1* and *test round 2*, respectively. The results of each test iteration will make base for improvements to the application in order to remove critical bugs and problems. Therefore, the conclusion is based mostly on the final usability test round rather than the pre-test and first usability test.

In the following, we describe our test approach in details and the testing methods used in each iteration.

6.3.1 Pre-Test

In the first iteration, we make a pre-test with one test person who are not part of the project group. The main purpose of this test is to get general feedbacks, bugs and usability problems with DASA. This pre-test is a black-box testing without any special setup, only note takers to record bugs and problems. The test person simply tries out DASA and report any issue to the note takers.

The improvements of DASA are based on the results of this pre-test, and all reported bugs and usability problems are rectified before the next testing iteration.

6.3.2 Test Round 1 and 2

The structure of test round 1 and 2 are the same, but have slightly different goals. In the first round, the goal is partly to test the usability of DASA and partly to find bugs and usability issues. A sub-conclusion about end-user programming is drawn from the results of this test round, and the bugs and usability issues are rectified before test round 2. In the second test round, the focus will be on getting insights and results to make base for the proof of our thesis. The conclusion of the test phase will mainly be based on the test round 2, because critical usability issues are removed within the pre-test and test round 1.

Both test rounds start with an introduction of the usability test. This includes some questions for categorizing the test persons, guide lines and introduction of the graphical user interface (GUI) of DASA, so the test persons have a chance to solve assignments prepared for the usability test (see Section 6.4.5). Only one test person is doing the usability test at a time. After the introduction, the test persons receive the assignments. They have one hour to solve as many assignments as possible; There are eight assignments. A person is beside the test persons during the usability test in order to help the participants if they get stuck—a test monitor[HS07]. When the test persons are finish with the assignments, they receive a questionnaire with questions related to DASA. Finally, we finish the whole usability test with an open-ended interview as an evaluation of the test process, so the test persons can give some general feedbacks.

As for the data collecting method, we use a video camera which records both picture and sound. For the recording of computer screen data, we use an application called Fraps[1] (see Section 6.4.1 for the usability testing setup). Only the usability test and the open-ended interview are recorded, because the introduction part is the same for all test persons.

The usability test results are analyzed after each test round, and changes and improvements to DASA are done between the first and second test round. Only very critical bugs are fixed between the test persons in order to avoid some barriers which can potentially block for some test results or insights.

[1] Homepage of Fraps: `http://www.fraps.com/`

6.4 Preparation

In the following sections, we describe the preparation of the usability test. This includes the setup, questions to categorize the test participants, assignments and questionnaire.

6.4.1 Setup

For the setup, we use a computer with a screen capturing program, Fraps, which captures what the test participants do on the computer. Meanwhile, we capture the reactions of the test participants by a video camera. Figure 6.1 illustrates how the setup looks like from above.

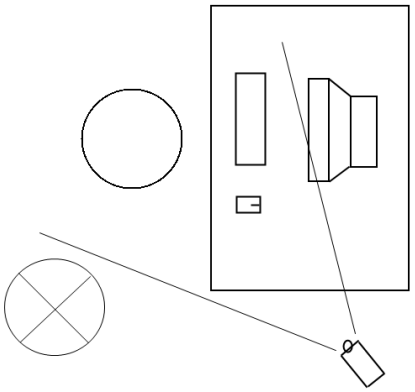


Figure 6.1: An illustration of the test setup showing a computer, a test person (circle), a camera angled at the test person and the test monitor (circle with a cross).

6.4.2 Target Groups

We choose to make the usability test on a range of different end-users with different knowledge level about programming. The range is divided into three groups. The first group consists of end-users who do not know how to program. The second group of end-users who have tried to program a few times, but are not skilled end-user programmers or novice programmers. The third group are novice programmers who are becoming professional programmers.

We strive to get an equal amount of end-users in each group, and of both genders to get a more clear view of the differences between the groups in the way they learn and DASA. Common for these groups is that they all know how to use a computer.

Group 1 (G1) Never tried programming.

Group 2 (G2) Tried programming before, but is not a skilled end-user programmer or novice programmer.

Group 3 (G3) Novice or professional programmer who do programming on a daily basis.

6.4.3 Questions to Categorize Test Persons

In order to categorize the test persons, we have a list of questions primary on programming subject. Based on their answers to these introduction questions, we will place them into the right group. These questions can be found in Section E.1.

6.4.4 Result Expectations

The following sections are descriptions of how we expect each testing group to perform during the test.

Group 1

This group is the most difficult group when it comes to programming, because they only have little basic computer knowledge. By this, we mean that they might be good in one application which they use on daily basis, but have difficulties when working in another application environment. Going from the basic computer knowledge to start doing programming seems to be a huge step. Nevertheless, that is what DASA is for; To help end-users to start programming. We expect this group to be able to solve the assignments which have no programming tasks, but they will have troubles with the programming tasks. However, we believe they do understand some elements and ideas in DASA.

Group 2

We believe this group can understand most of the information presented in DASA, because they already have a basic computer knowledge and a little knowledge in programming. We expect these test persons to have a minor understanding of the conditions and iterations used in programming, but we do not necessary expect them to be able to formulate it in a way, the computer can understand. Therefore, we expect the results will show that they are capable of catching the idea using programming to solve computable tasks and be able to solve many of the assignments with programming. Their scripts might not look perfect, but should do what they expected the scripts to do.

Group 3

We expect this group to solve the assignments without any problems, because they already have training in programming. However, the challenge for this group is still the unknown scripting language and the application environment which counts for all three test groups. Unlike the other groups, this group should be able to understand the scripting language fully after being introduced to it. Therefore, this group is more a control group rather than a test group.

6.4.5 Assignments

In the following, we describe the eight assignments for the usability test in plain text[2]. The precise description of the tasks in the assignments, in the original Danish version, are found in Section E.2. We have written the assignment in Danish in order to remove the language as a factor which can affect the usability testing, as all test participants are Danish. Five of the eight assignments are introduction assignments which introduce the test participants to the different tools available in DASA. The last three assignments let the test participants solve the assignments independently. In addition, the difficulties of the assignments are increasing in each assignment.

Assignment #1 - Basic Shapes

This assignment introduces the test participant to the basic drawing tools, rectangle, ellipse and line. The test participant is asked to draw a rectangle, an ellipse and a line. No requirements of size or color. In addition, we ask the test participant to notice the disclosures box.

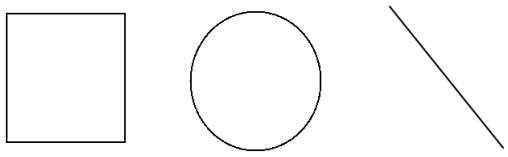


Figure 6.2: An example of the three basic shapes.

[2]For the interested reader, we show how these assignments are meant to be solved in some video clips which can be found in the enclosed CD-ROM.

Assignment #2 - Coloring the Shapes

In this assignment, the test participant is asked to color the shapes from the previous assignment. The rectangle must be red, the ellipse blue and the line green. The purpose of this assignment is partly to introduce the test participant to how to color their shapes, and partly to see whether the test participant can relate their mouse actions with the disclosures.

Assignment #3 - Usage of Commands

The test participant is to draw a circle with 250 pixels in radius and with a center at 300,300. This must be done with a command, and the idea is to force the test participant to use the disclosures box to find the right command for drawing a circle. However, the assignment asks the test participant to draw a circle, but the disclosures box has only shown the `ellipse` command from assignment #1. Furthermore, the arguments to the disclosed `ellipse` command are not radius nor center, so the test participant must intuitively guess the radius and center arguments. The purpose of this challenge is to see if our design of the language is intuitive enough, and how the end-user thinks and do in such a situation where they feel a bit lost.

Finally, the test participant is asked to move the circle with the mouse and then move it back with a command. The circle must also be colored blue, and the pen width must be 5 pixels. These tasks are to see if the test participant understands the usage of disclosures box and its disclosures.

Assignment #4 - Create a Procedure

In this assignment, we want to introduce the test participant to procedures and how these can be created from disclosures. The test participant is asked to draw a bicycle with the mouse. This bicycle must be duplicated five times at different positions and with different colors. The goal in this assignment is to see whether the test participant understands the idea and advantage of making an abstraction to many actions.

This assignment was changed between the first and the second test round. In the first test round the test participants were only supposed to draw a square and then color it before eventually creating the procedure. The result from the first test round have shown that none of the test participants did understand the purpose of the assignments regards to the procedure. With the new assignment, we have tried to make it more obvious that a procedure should be used.

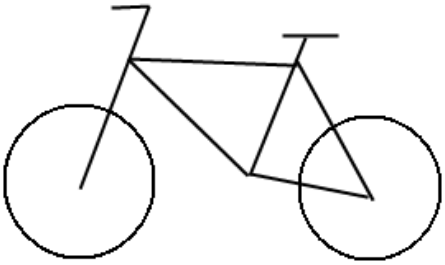


Figure 6.3: An illustration of a bicycle for the assignment #4.

Assignment #5 - Usage of Script

This assignment is different from the previous assignment, because it does not ask the test participant to draw figures, but instead manipulate with an existing drawing with scripts. The existing drawing shows three red rectangles above a black line and three green rectangles below the black line. First the test participant is asked to change the colors of the rectangles, the red ones to blue, and the green ones to red. We have drawn many different shapes on top of the drawing, so the test participant cannot use the mouse to select or move the colored rectangles, but is forced to use a script. After the color changing, the test participant must switch the positions of the colored rectangles (see Figure 6.4).

The purpose is to see if the test participant can transform an idea to a solution into a script to solve the problem. For a non-programmer this assignment seems to be very difficult, however, this challenge can show us if DASA can be of any help in the learning process of creating a script.

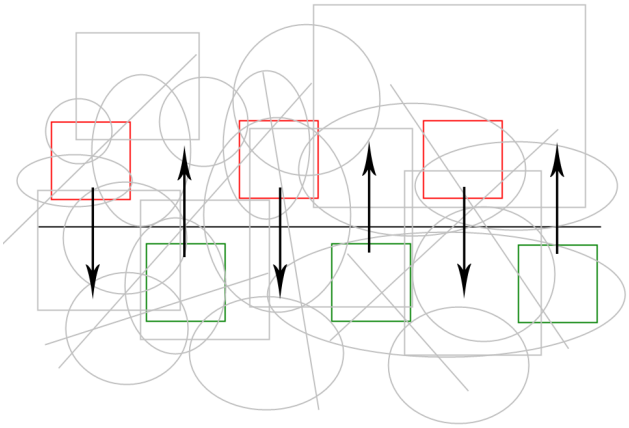


Figure 6.4: An illustration of the task asked in assignment #5.

Assignment #6 - Drawing Circles

This is the first of the three independent assignments. The test participant is asked to draw 30 circles with increasing radius. How the test participant is to solve this assignment is not told in the assignment. The purpose of this assignment is to see whether the test participant is capable of solving the assignment through programming by using the knowledge from the previous assignments. Furthermore, we want to see if the test participant recognizes and modifies a generated script in the generated script box to solve the assignment. In fact, there are many ways to solve this assignment, and the interesting part is how the test participant choose to solve it.

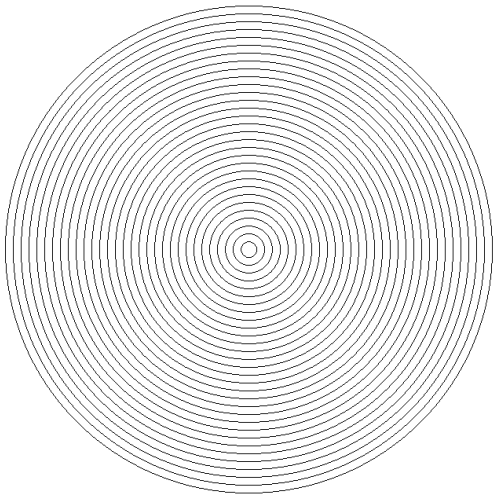


Figure 6.5: An example of how the result of assignment #6 can be.

Assignment #7 - The Wild Forest

In this assignment, the test participant is asked to draw a picture of a forest with a house and a sun. 30 trees represent the forest, and how these are drawn is up to the test participant. The main purpose in this assignment is to see whether the test participant has learned anything from the previous assignments, and if the test participant is able to solve the assignment with a script. In addition, we are interested in seeing how the test participant thinks when trying to solve a problem with programming, and if DASA can ease the programming process. Possible good solutions would be to use a procedure with a loop and a random-structure.

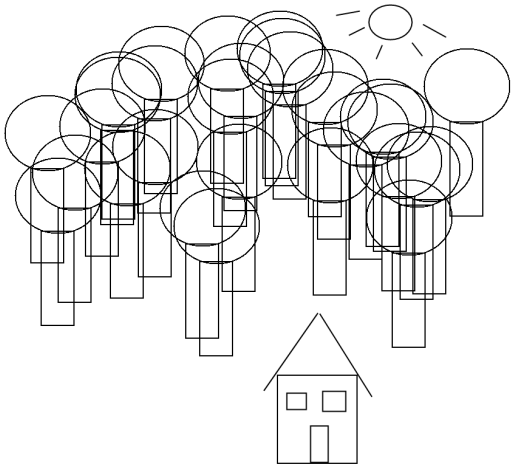


Figure 6.6: An example of how the wild forest with a house and a sun can be drawn.

Assignment #8 - Coloring the Wild Forest

This assignment is a continuation of the previous assignment. The test participant is asked to color the drawn picture, and like the previous assignment, the interesting part in this assignment is to see how the test participant decide to solve this task. Furthermore, we can use this assignment to see what part or how much of the tools and techniques introduced in the introduction assignments are remembered, and in such case why the test participant remembers some tools and techniques and not other. Good choices for a solution here would be the use of foreach- and if-structures in a script.

6.4.6 Multiply Choice Questions

After the usability test, the test participants answer a multiply choice questionnaire consisting of two parts. The first part has a total of eleven questions each with four to six possible answers. Ten of the eleven questions have one correct answer, the last question is about how the test participant will solve a problem wherein all answers to that question are correct.

The second part consists of six questions about how useful the test participant thinks different parts of the prototype is. All questions has a “do not know” option.

The first part of the questionnaire was changed between the first and second test rounds as it was simply not precise enough.

6.4.7 Interview

To complete the test we will have an interview with the test persons. Both group members will be present as this is an open-ended discussion based on what they have answered on the questionnaire and on what happened during the test. From there the interview can go in any direction that we or the test person believe to be interesting and relevant for our project.

6.5 Test Round 1 and Results

In the following, we describe the results obtained during the first test round. This includes both problems, successful results and general observations. The results are divided into the corresponding assignments. The final section is a walk-through of some of the problems found during the usability test, and a description of how we intend to implement solutions for them before the second test round.

6.5.1 Participants

There were 5 participants in the first test round in total. Based on the introduction questions, we divide these into the testing groups and listed them in Table 6.1.

Table 6.1: Participants in the first test round.

Group	participant	Programming skill	Computer usage	Assignments
G1	**A**	Never programmed	Daily	All
	B	Only spreadsheets	3 times/week	All, but 7 and 8
G2	*none*			
G3	**C**	Novice programmer	Daily	All
	D	Novice programmer	Daily	All
	E	Novice programmer	Daily	All

6.5.2 Assignment #1 - Basic Figures

In this assignment, G1 had difficulties with using the basic drawing tools. This issue is not related to our part of DASA, because it is outside our test scope, but it still has an effect on the test results. G1 did not notice the disclosure box in this assignment even when stated

in the assignment that they should be caution about it. A problem here might be the many boxes (disclosures, procedures, generated script, script) give a disoriented effect on the test participants whom already are nervous in such a usability test.

As a success result, all from G3 noticed the disclosures box and also understood what it disclosed. However, some test participants in G3 did not get the point of having such a disclosures box at this point. Unlike G1, the G3 are used to program in heavy editors with even more boxes than in DASA, and, therefore, were comfortable with the GUI.

6.5.3 Assignment #2 - Coloring the Shapes

There were no problems in this assignment, but there were an observation which should be mentioned. There are two buttons which can be used to choose the color of a figure. All test participants from G3 used the left one, which is an icon illustrating colors and lines, while G1 all used the right one which is a button with the current color as the background color and the text “Color”.

A success in this assignment was that almost all test participants understood the disclosures and the purpose of these at this point.

6.5.4 Assignment #3 - Usage of Commands

Only one from each test group managed to guess that `radius` and `center` could be used as arguments to the command `circle`. The rest tried with arguments which were disclosed. In addition, when using the `move` command, only two from G3 tried to call the command with missing arguments. The rest was unaware about the optional arguments. G1 in general, had difficulties with the understanding of arguments. They only typed the values and omitted =, or they tried to set the command equal to a value. E.g. `circle=250` for a circle with radius 250. Also, one from G1 intuitively wrote `circle with radius 250`[3] for the same task.

A success is that all test participants used the disclosures box to find out how to write a command. Also the commands `set` and `move` were quickly discovered. There was a bit confusion about values to the `move` command. Some test participants saw the values as absolute coordinates rather than relative values.

[3] This is a translation of: “cirkel med radius 250” which was originally written.

6.5.5 Assignment #4 - Create a Procedure

The main problem in this assignment was none of the test participants understood the purpose of this assignment. There were also different understandings of what a procedure was in the context of the test. The reason for this problem might be found in the problem that all test participants had problems with creating a procedure. There was no help in the assignment nor the tool to create a procedure. There were only three `Create Procedure` buttons placed three different places, but these confused the test participants even more, because they had different effects.

Finally, when the test participants created a procedure, they had problems with invoking the procedure. One from G3 tried to use the `play` button, belonging to the command line, to invoke the procedure. When the test participants figured out that they could select disclosures and create a procedure, they wanted to edit the existing procedures, but this option was not available.

Without telling how they could select more disclosures at a time, they figured it out by holding down the `ctrl` key. This is a positive observation. Clearly, the test participants have seen such a select box somewhere else or intuitively used the same method as when selecting more objects in e.g. Windows OS.

Another positive thing to say in this assignment was that all test participants did manage to generalize position and color when creating the procedure. Whether this is because of the assignment which told them to work with position and color or the options were intuitive was not clear though.

6.5.6 Assignment #5 - Usage of Script

G1 had problems with understanding loops and if-statement and could not see the benefits of using them. However, they did all manage to create a `foreach`-loop through the icon on top of the script box. While modifying the script, test participants needed to undo changes, but DASA did not support this.

Again in this assignment, the disclosures box had shown to be a valuable tool for the test participants. They used it very often as a look-up dictionary to remember command and argument names. In this assignment, they used it to look-up the `set` command.

6.5.7 Assignment #6 - Drawing Circles

It turned out that none of the test participants noticed the generated script box or even understood the information in that box.

G3 quickly solved the assignment by creating their own scripts. This was what we expected from G3, since none of them noticed the generate script box. This can somewhat be regarded as a success, but not useful information in the sense of the test purpose of this assignment.

6.5.8 Assignment #7 - The Wild Forest

One from G1 did not reach this assignment, and the other from G1 did finish the assignment, but with much of help from the test monitor. The test participant did find out to use a `foreach`-loop, but could not see how to combine with a condition. G3 did solve the assignment easily, but there were many bugs related to the script execution which affected the results for this assignment. Therefore, the results should not be taken too seriously.

A positive notice was that two from G3 and one from G1 remembered about procedures and used these to create trees. In addition, all from G3 used the random keyword, which had not been introduced, to create a script to draw the trees at random positions.

6.5.9 Assignment #8 - Coloring the Wild Forest

Only one from G1 reached this assignment and managed to finish it with help from the test monitor. It was clear, that the test participant in G1 had difficulties with solving the assignment alone, but the test participant did get some ideas to how the assignment could be solved. The problem was to get started.

Two from G3 and one from G1 used a `foreach`-loop on the type of figures with the `set color` command to color the trees. This was a success, because they remembered how to apply changes to a number of figures with the same shape. However, all of them were surprised about the sun and house were colored too when they colored the tree leaves (circles) and tree body (rectangles) respectively—we name to this issue the “sunshine problem”. The last test participant from G3 used the `select` command to select all figures with the type of *circle*, instead of the `foreach`-loop, but wondered about the sunshine problem too.

6.5.10 General Observations

Most of the test participants used common shortcuts, `ctrl+z` (undo) and `ctrl+a3` (select all), when drawing, but they did not exist. This problem confused the test participants, because they are used to use these shortcuts in e.g. Windows OS and other applications such as a text processor. Intuitively, they also expected the shortcuts to be a standard features in the scripting box as-well.

When changing the color of a figure, test participants tend to click on an area showing the current color in the properties box rather than the `...` button, because they did not know what that button means. Furthermore, the commands `ellipse` and `rectangle` were used for drawing a circle and square which confused the test participants. Generally, DASA lacked information about commands, disclosures and scripting language syntax. Test participants from G3 even suggested a code assist for the command line which can provide them with information of commands and their parameters.

If we look at how G1 and G3 try to complete the assignments, it was clear that G3 already know how to program. They had a tendency to find smart solutions to the problems stated in the assignment which sometimes let them to solve a whole assignment in one step. E.g. one from G3 insisted to create procedures for creating a tree, a house and a sun in assignment #7. All these procedures were then executed in one step. One from G1, on other hand, created a procedure for creating a tree, and manually execute each tree, and then added the house and the sun.

6.5.11 Solutions

The following is a description of several possible solutions to the problems we have found during the first test round. We prioritize the problems, so we can make sure, we get the most important solutions implemented before the second test round. Each is prioritized in respect to how serious the problem is and how easy (and time consuming), we expect it to be to implement the solution. Furthermore we have noted whether the problem relates to the application, to the domain or to end-user programming in general.

Use of `topleft` as argument instead of `upperleft` (serious, easy, usability)

> During the test we saw several try to write `topleft` when they were specifying the position of an object, even when they had already seen the same being known as `upperleft`. It is easy to change this as all parameters that use `upperleft` should just had another synonym added.

Buttons below the procedure list (serious, easy, application)

It took a while for all participants before they discovered the right-click menu for the procedure list. Therefore, we need to add buttons for the most important options below the procedure list just as we have with the disclosure list.

Command information window (serious, hard, usability)

All participants needed more information about what commands were available, and how they could be used.

Adding a command information window which shows all its synonyms and all the possible parameters would greatly help. It would even give the users a place to discover new commands instead of only through the disclosure list. Adding the possibility to right-click a disclosed command and then get information about that one would speed up the information search. Our system already have a way to access command and parameter information internally, so it would mostly just be a question of presenting it in an understandable way.

Shortcuts such as `ctrl+z` and `ctrl+a` should be added (cosmetic, easy, usability)

The common shortcuts `ctrl+z` and `ctrl+a` were known by all participants, but our application do not support these.

DASA do, however, support other common shortcuts such as `ctrl+c`, and the other shortcuts should just be added in the same way.

Removing the `firsti` and `lasti` variables from a generated script (cosmetic, easy, application)

These variables were originally added to ease the editing of the script, but it turned out to be confusing for participants who were not used to programming as they did not see them as variables, but more like special keywords.

Simply removing them and writing their value in the `for`-structure is easily done and should be less confusing.

Edit of procedure (serious, difficult, application)

Several participants wanted to edit a procedure, they had created, but this is not supported by DASA.

The solution is to add a new popup window which can be used to edit the script inside the procedure. Because of the way we have created the different parts of the application, it should be fairly easy to add syntax highlighting and other features to this new script editing box.

Appending disclosed commands to existing procedures (cosmetic, easy, application)

In addition to the edit of a procedure. it was requested to have a way to easily append disclosed commands to a procedure.

This feature can be added in the form of an option to the right-click menus of both the disclosure list and the procedure list. This should then simply append the selected disclosed commands to the currently selected procedure.

Changing the looks of the script generating box (serious, medium, usability)

No participants took notice of the script generation box even when a new script had been generated. Some mentioned that the problem might be because of the color gray, and it, therefore, had the look of a disabled windows control.

We need to change its colors and add some other attractions, e.g. a flashing icon, to make user more aware of its existence.

Write out all "create procedure" buttons (cosmetic, easy, application)

Some participants were confused that there were three buttons with the text "create procedure".

A simple solution to this is to write more descriptive descriptions in the buttons.

Auto-completion/suggestion when writing a script (cosmetic, hard, usability)

Many programmers are used to scripting environments where a small box is showing while writing. In this box is a list of all the possible keywords and variables which can be written at the current location. This makes it easy, and very quick to write correct code. This box can also include argument information for a function call.

As with the command information window, our system already have a fairly quick way to access command and parameter information. The difficult part is the box which needs to work perfectly as even the slightest problem with this will give confusing problems.

Shortcuts to the drawing tools (cosmetic, easy, application)

One participants were annoyed with the click on a shape icon on toolbar for each shape to draw. They suggested a keyboard shortcut.

As an idea, we could use `ctrl+p`, `r`, `e` or `l`.

6.6 Test Round 2 and Results

In the following, we describe the results obtained during the second test round. This includes both problems, successful results and general observations. The results are divided into the corresponding assignments. These results are the base for the following sections that discuss the different aspects of DASA in relation to the end-user research field and the drawing domain.

6.6.1 Participants

In the second test round there were a total of 10 participants. Based on their answers to the introduction questions, we have divided them into the three testing groups as listed in Table 6.2. The spreading of the end-users in the groups did not end as we originally intended.

Table 6.2: Participants in the second test round.

Group	participant	Programming skill	Computer usage	Assignments
G1	**F**	Never programmed	Daily	Up to 5
G2	**G**	A little C#	Daily	All
	H	PHP, Spreadsheets	Daily	All
	I	A little PHP	Daily	All
	J	A little PHP	Daily	All
G3	**K**	Novice programmer	Daily	All
	L	Novice programmer	Daily	All
	M	Novice programmer	Daily	All
	N	Novice programmer	Daily	All
	O	Novice programmer	Daily	All

6.6.2 Assignment #1 - Basic Figures

None had problems with drawing the figures and saw the disclosures box, but G1 did not understand precisely what it was at this point.

6.6.3 Assignment #2 - Coloring the Shapes

After the test participants draw a line in the previous assignment, they had to color the figures, but there was a problem with choosing the color while having the line selected. Several test participants could not understand why the line was colored while their intentions were to color the rectangle. They simply missed the selection mark on the figures. Nevertheless, at this point all test participants understood the simple disclosures such as `set color` and command name, but G1 still had problems with understanding the arguments.

While coloring the figures, one from G2 double clicked on a figure and expected it to show the properties of the figure, but there was no such option available. One from G1 and G2 entered the figure properties through the upper menu of the application (Edit->Properties) when changing the color rather than using the color icon in the toolbar.

6.6.4 Assignment #3 - Usage of Commands

A problem in this assignment was that G1 and few test participants from G2 used the `ellipse` command with the arguments shown in the disclosures box, and they all get stuck. After some help from the test monitor, they managed to draw a circle with `circle` command. After drawing the circle, the assignment asked test participants to move the circle. The problem was that when they did so, they did not see the `move` command which was disclosed to help them move the circle back as a part of the assignment. To find help, they found the "command information" feature which is a popup window with information about all the commands. However, the boxes in the command information are all empty by default, and this was a problem, because it confused the test participants. Furthermore, G1 and one from G2 did find information about commands in the popup, but did not understand the provided information.

A success was that all test participants used the disclosures box to track back how to draw a circle, but started out to use the arguments shown with the disclosure (which were `topleft` and `bottomright` rather than `center` and `radius`). One from G2 tried intuitively to use `center` and `radius` as arguments right away, others found this in the command information box. Also when coloring the circle with a command, all test participants found the `set color` in the disclosures box. Some used intuitively the `set penwidth` to set the pen width of the circle, while others again used the help in the command information.

6.6.5 Assignment #4 - Create a Procedure

A big lost at this point was one from G1 gave up. The test participant was too confused and needed very much help from the test monitor. Beside that, most test participants had problems when it came to creating a procedure as the assignment asked them to do. They did not know how to get started. With a little help from the test monitor, they managed to create a procedure from the disclosures, but with only one disclosed command which was not enough. They tried to append the missing disclosed commands to their procedure, but the generalization of arguments was not applied to the new appended disclosures. That confused the test participants to start all over again. One from G3 thought the script box was the procedure wherein the test participant added disclosed commands.

Although there were problems with getting started to create a procedure, the success was that the test participants understood the idea of generalizing the position and color arguments when creating the procedure. In addition, G2 and G3 tried to call their created procedure with a script, but failed. Instead they tried to call it through the command line which was a success. Only one from G1 executed manually the procedure.

An unexpected observation was one from G3 tried to select all shapes in the draw area and drag the whole drawing into the script box. The test participant intuitively thought, DASA would automatically generate a script for that drawing. Few other tried to select the whole bicycle and right click to create a procedure, but this was not possible. Furthermore, one from G2 tried to execute a procedure as an argument to a command `run procedure=bicycle`.

6.6.6 Assignment #5 - Usage of Script

A problem in this assignment is that half of all test participants spread among the three groups started with using the command `select` with `color` as an argument, but this argument was not a part of that command. In addition, one from G3 used the `select` command inside a `foreach`-loop, because the test participant did not expect, the loop would select an object when executed.

Successfully, all test participants did find the `foreach`-loop through the icons above the script box, but G1 and most of G2 did not know how to get further. With some help from the test monitor, they did manage to finish the assignment. All from G2 and most from G3 used a single = in their `if`-statement even when == was also supported. This was a success regarding to our expectations, because we expected most test participants would use the first syntax.

An interesting observation was two from G2 tried to select the rectangles in the assignment with the constructions `foreach red rectangle do` and `select rectangle color=red` respectively. These constructions were not supported, but an intuitive syntax.

6.6.7 Assignment #6 - Drawing Circles

One from G2 detected the generated script, but did not understand what DASA generated and how to use it. This was a problem, because the generated script should help the test participants to solve the assignment more efficiently. A general problem with this generated script was that all test participants, but one from G2 and two from G3, overlooked the it. However, they tried to create a script by themselves which was a positive observation. Those test participants who detected the auto-generated script also changed and executed it correctly. This was only a minor success, because only one from G2 detected it, and we expected all from G3 to find and use the generated script easily.

Most test participants from G3 went straight to the script box and created a loop which draw the circles without any problems. Those who did not remember the command for drawing a circle used the command information and the disclosures box to remind themselves the name.

6.6.8 Assignment #7 - The Wild Forest

One from G2 started with drawing a tree manually and then used copy paste to make more trees. The problem was that this participant did not remember much of the introduced tools in the previous assignments. It did not seem like the test participant had an understanding of DASA. Other problems were related with the syntax of the `random`-structure. The test participants intuitively used the structure with two arguments, `random 100 400`, to get a random number between the defined range.

All, except for one from G2, first draw the tree by hand and then created a procedure of the disclosures. They also created a script with the usage of the procedure. This was a success even when they actually tried different approaches at the beginning, but these approaches were still within programming.

6.6.9 Assignment #8 - Coloring the Wild Forest

One test participant from G2 did not reach this assignment, but for all other test participants in G2 and G3, this assignment might be too easy. It did not challenge the test participants enough as they solved it very fast and easily. However, one test participant from G2 forgot

the `foreach`-loop construction and used the `select` structure again.

A success was though that all test participants from G2, except for one, and G3 used `foreach`-loop as they have been introduced in assignment #5.

6.7 General Feedback

In this section, we shortly describe some of the results from the multiply choice questions and the interviews.

6.7.1 Multiply Choice

The multiply choice part of the test was supposed to give us a picture of whether the test participants had learned what we wanted them to learn. The questions themselves, however, could have been better outlined.

Of the 11 questions in the first part, 8 were answered in the same and correct way by all test participants. One question was answered correctly by half of all test participants. It turned out that the question was ambiguous, so the rest of the answers could also be seen as correct. One question askied the test participants how to solve a task whereof half chose top solve it by programming, and the rest chose to solve it by other ways.

The last part of the multiply choice questions were about how useful the test participants felt, different parts of DASA were. For both the disclosure, procedures and scripting the general opinion were very positive. There were no clear picture of how useful the iterative task recognition was as almost half of the test participants answered “do not know”, and the rest were scattered. This was a result of the fact that several test participants simply did not notice the generated script. The overall opinion about the tool were positive.

6.7.2 Interview

We primarily asked the test participants what they thought about four elements: The assignments, the disclosure box, the scripting language and the application in general. We also asked several users what could be the reason for them not noticing the generated scripts and command information. The rest of the interviews was more specific for each test participant.

The general opinion about the assignments were positive. They were not difficult as first feared, but it was not clear whether they introduced the different aspects of DASA

good enough. All test participants agreed that the disclosures box was a great feature which helped them to learn the commands, but also remember them as they could go back in their history. Most test participants were positive about SDCASL and think it was simple and understandable. The opinions about the whole application were also fairly positive, but several mentioned that they would not use it — some simply because they would require a lot more from a drawing application. One find no need to learn to program, because need of the test participant was already covered by exiting applications.

To the question why some test participants did not notice the generated scripts and the command information, there were no clear answer. Some mentioned that they did see the generated scripts, but they simply did not understand what it was or how to use it, and the command information was not visible enough.

One test participant pointed out that the test participant generally need very detailed explanations in order to understand things. Other also pointed out that they are not so quickly to learn new things. They felt, they did not had time enough DASA, so they found it difficult to use. Also the English language in DASA was a barrier which slowed down the test participants further.

6.8 Result Discussions

In the following sections, we discuss the results from the test in relation to the field of end-user programming and drawing domain.

6.8.1 Results Discussion Related to End-User Programming

The following discussion of results from the usability test involves the learning barriers described in Section 2.3: *Design*, *selection*, *coordination*, *use*, *understanding* and *information*. We point out the problems and relate them to the learning barriers.

The overall general observation of our usability test was that the groups of test participants were solving the assignments very differently. So in order to give a structured discussion, we divide the discussion into smaller parts with relation to their respective test groups. We refer to the test participants to those from test round 2 if nothing else is mentioned.

Test Group 1

As expected, group 1 had problems with programming. One test participant gave up after assignment #4 (Section 6.6.5), but was urged to continue and did finish assignment #5 with very much help; DASA could not help the that participant to overcome the design barrier regarding to programming. When it came to creating procedures as an abstraction or grouping of commands, the test participants did not understand the idea at first (Section 6.6.5). From test round 1 results, we saw that one of the test participants in test group 1 did not reach further than assignment #7 (Section 6.5.2), but required very much help from the test monitor.

The general observation of group 1 was that they did know what to do, but did not know what to use in order to obtain their goals (Section 6.5.9)—the selection barrier. However, with some help they did manage to use programming to solve some tasks, but could not understand the idea and meaning of it—the use barrier. Also the barrier of coordination occurred when they figured out that they need an if statement in an assignment, but they did not know how to combine it with other structures such as a foreach-loop (Section 6.5.8).

If we solely base our sub-conclusion on the test results, we saw that DASA was too difficult for test group 1 to use. It did not totally break or lower some of the learning barriers. Nevertheless, the open-ended interview with the test participants after the usability test revealed some factors which can explain why DASA did not work for this group (Section 6.7.2). One test participant said that everything must be explained e.g. why to use a foreach-loop, before the participant can make use of it. Other test participants said that they just need more time to use DASA, because at the end of the usability test, they began to understand the idea of solving iterative tasks by programming, but need more time to get more comfortable with it. Furthermore, some test participants said that the difficulties were also caused by the English language in DASA, as they could not understand some of the keywords. So more in-depth explanations, changing language in the application to their speaking language and more time, and DASA might give some positive results for test group 1. Nonetheless, the general result and observation is that DASA was too difficult for end-users with only little basic computer knowledge.

Test Group 2

The test results for test group 2 have shown that test participants in this test group understood and made much better use of DASA than group 1. The reason is because they already have much more computer knowledge than group 1, some have even tried to program a little. Although they have a minor programming experiences, most of the test participants still faced the same barriers like those in group 1, selection, coordination and use barriers, but to a less degree. Most of the test participants managed to overcome the mentioned barriers by searching for help within DASA such as using command information or simply try out a script (Section 6.6.4). In other words, they keep continuing instead of giving up. Two features in DASA were specially helpful for the test participants search for help. First, all test participants expressed that the disclosure box helped them learn about the possible commands and the syntax for using them (Section 6.6.4). Later, the disclosures were also used as a reminder for when they were to use the commands in a script. The feature of inserting language structures through forms were also very helpful as it made it easier for the test participants to learn and try out the different aspects of the scripting language. Both of these features did not help group 1 as much as originally hoped, however. The reason still being the general lack of programming knowledge and training.

One test participant in group 2 did not overcome the coordination barrier when it came to using a loop structure (Section 6.6.7). The test participant was confused about how to use the incrementing variable in a for loop. Several other test participants hit the understanding barrier when they thought they knew how to use some script structure, but their script did not do as they expected (Section 6.6.6). In continuation hereof, they faced the information barrier, since most of the test participants did not know how to fix their script; One did the same mistake in a later assignment(Section 6.6.9). In general, this test group lacked the debugging skills which test participants in group 3 are possessing.

The sub-conclusion for group 2 is that the test participants were capable of learning to use if statement and loop structures, but sometimes faced the understanding barrier and then the information barrier in the continuation of the first barrier. DASA seemed to help this test group to overcome the selection, coordination, and use barriers which test group 1 also faced, but did not overcome. However, some test participants from group 2 still had problems with the coordination barrier (Section 6.6.6), so test participants from this group had different ways of learning things. This indicated that DASA did not manage to help all test participants from this group, but was capable of helping most of the test participants to learn a new scripting language and start programming.

During the interview, one participant from group 2 mentioned that he could not see the point of programming as all the applications he used were able to do what he required of them. This is an example of the sociological barrier, see Section 2.3.1.

Test Group 3

As for test group 3, we have not much to discuss about their test results. For most of the assignments, they went straight to the scripting box and solved the tasks eagerly with programming (Section 6.6.6). Again the feature of inserting language structures through forms proved helpful when they needed to learn the syntax of the scripting language. We expected this on beforehand.

A Problem with Iterative Task Recognition

The usability test did not give enough data about the iterative task recognition for us to make any definite conclusions. The main reason for this were that only a few of the test participants actually used it. The test, however, showed that it is important to make the end-user more aware of the features in an application. When a new script were generated it were shown together with a green colored status message at the bottom of the window. However, this simply was not enough for most of the test participants to notice it. Those who did, had trouble comprehending what exactly it was. Group 3 did understand the generated script, but would rather create their own.

6.8.2 Results Discussion Related to the Domain

Most of the observations we have done during our testing rounds regarding the drawing domain, have to do with the drawing area of the application or the usage of some of the commands. However, also a few observations about how the scripting language was used are interesting.

All test participants recognized the three shapes, rectangle, ellipse and line, and most had no problem at all drawing them using a click-and-drag approach. This approach is a common approach in other drawing applications too, including Microsoft Paint which is included in the Windows operating system. One difference between DASA and Paint, however, is that once a shape has been drawn in Paint, the shape cannot be changed[4], while DASA offers the possibility to manipulate with drawn shapes. For most of our test partic-

[4]The newest version of Paint which is included in Windows 7, have added the possibility to manipulate the size of the last drawn shape.

ipants, this possibility was a surprise as many actually expected to be able to manipulate their drawn shapes through menus or by double clicking. The eight handles shown when a shape is selected, did not seem to be unfamiliar either. An interesting observation with the selection, however, is that most test participants selected and moved shapes by clicking on their periphery, which must be because they expected that they could click through the inner area of a shape. The few that did select by clicking inside the shape also expected there to be a fill-color, which is usually the case for other drawing applications. As with many other aspects of our application several test participants intuitively used `ctrl+click` to add shapes to the current selection. This is a trait learned from the Windows operating system together with the use of the `ctrl+a`, `z`, `x`, `c`, and `v` keyboard shortcuts which were also used frequently.

Even though all recognized the three shapes, some was confused about the lack of differences between an ellipse and a circle, and between a rectangle and a square. Mathematically, there is a difference between these, but in DASA they are treated as the same, which did lead to some scripts not doing what the participants intended (Section 6.5.9). Therefore, it is important to take the correct meaning of geometric terms into account when dealing with the drawing domain.

The use of a coordinate system in a drawing tool is not uncommon, but only few applications actually make use of it for other than displaying the mouse position. It seemed, however, that only G1 had slight problems grasping this. One reason could be that G2 and G3 had all tried to program before, and the exact same coordinate system is used for application development to specify the position of buttons, input boxes etc. Some test participants did question where the (0,0) coordinate was in DASA, but fairly quickly realized that it is in the upper left corner.

When the test participants understood the drawing area had a coordinate system, all values of two numbers divided by a comma were recognized as a coordinate set in the drawing. They all also recognized the use of *X* and *Y* for this. This even made some test participants to not pay attention to what was written elsewhere in a window, when they saw two input boxes with the labels *X* and *Y*. Whenever a position needed to be specified in DASA, it was possible to click on the drawing to easily use that position, but during the entire test only one participant discovered this even though the text "*(click on the drawing)*" was written right above the coordinate input boxes.

Since DASA disclosed an English name for a color, whenever possible, all the test participants used this, and thereby could guess the name of a color, they had not used

before (Section 6.6.3, Section 6.6.9 and others). The few, from G2 and all from G3, who actually did see another way to illustrate a color (the red,green,blue approach, e.g. `255,37,144`), did not seem to be confused about this. G3 had without a doubt seen this during their studies, while those from G2 have seen this in relation to their work with HTML. Expressing colors using a hash symbol with values in hexadecimal (e.g. `#FF2590`) was not seen during the test as it was not disclosed, and the test participants might think it was not possible.

The disclosure box works well in a drawing domain, as long as the commands are expressive enough. Most of the names we chose for the commands and their parameters were positively received. One observation, however, show that when using a scripting language and a command line, the test participants will rather write `topleft` and `bottomright` than `upperleft` and `lowerright`, probably because it sounded more *exact*.

Generating a script to finish iterative tasks is certainly possible in a drawing domain. The domain, however, does contain a lot of noise and some effort is required from the user to actually get a usable script. The user has to know how the generation of a script works in order to draw in a way where it would produce the correct script, e.g. ending an iteration by unselecting a shape to avoid manipulating the wrong shape in the next iteration (Section 6.6.3). The script which is generated, however, can be very confusing, and maybe even scary for G1 and G2. Hiding this script and just make previews of what will happen when it is executed, like it is done in Eager, could be a better approach for our target groups.

6.8 Summary

In this chapter, we have described the purpose and goals of our usability testing. The primary goal was to achieve some test results to use as a base to prove or disprove the hypotheses. In addition, we would also like to find some insights on how end-users are dealing problems with programming. These goals set some requirements for our choice of testing methods, and we find usability testing suitable for our test. We structured the test phase in iterations whereof the first two iterations were to help finding bugs and critical issues in DASA.

Furthermore, we described the preparations of the usability test which included the questions to categorize test participants, assignments and a questionnaire. We also described some expectations for the three different test groups of our usability test. For the test results, we discussed and related problems and successes with the area of end-user programming and the domain. The general results are that part of DASA was too difficult for test group 1 and worked for most test participants from test group 2. For test group 3, it seemed like DASA was too easy to use.

Chapter 7

Epilogue

In this chapter we will first conclude on the whole project, based primarily on the hypotheses and the results from the test. Next we will evaluate our own work during this project period, followed by a description of some of the ideas we have for possible future work that could be done with the product. Finally we will put our tool, DASA, into perspective and look at how it can be used in the field of end-users, beyond the drawing domain.

7.1 Conclusion

To conclude on this project, we return to the hypotheses from Section 4.1. We conclude on each of the hypothesis individually and base this on the results from the usability testing.

Hypothesis 1: We can teach end-users, with basic computer knowledge, the fundamental principles behind programming by using the concept of learning by observation and an improved environment with several kinds of feedback.
During our usability test, we saw several end-users who quickly learned how to perform actions using a command line or a script and express both conditions and loops using SDCASL. The commands were learned by looking at the disclosed actions in the disclosure box. Some further commands were guessed based on this, others were discovered using the interface of the application which provided the needed information. The conditions, loops and other structures of SDCASL were primarily learned by using the interface as well, where special forms, used to easily insert language structures into a script, teached these structures through learning by observations.

We originally intended our prototype and language to teach the fundamentals of programming to end-users with no programming knowledge, but we did not see much success for this group. DASA, and maybe SDCASL, is still to complicated for them. We had much more success with those end-users who have had minor experiences with programming, partially because they had already been introduced to the fundamental principles of programming such as conditions and loops. Because of these two observations, we believe that a longer test period with an improve test setup will prove that learning by observation approaches, such as self-disclosing, and an improved working environment will help end-users to learn programming.

Figure 7.1 shows our intended target group, see Section 4.2, and the group which DASA seems to have actually been a success to.

Figure 7.1: An illustration of which part of the users our application actually ended up helping. Our original plan is shown in light gray.

Hypothesis 2: A task with redundant actions is completed more efficiently by an end-user who uses a script constructed by an automatic recognition of these actions.
A major problem with the test regarding this hypothesis is that only a few participants actually noticed and used the recognition of iterative tasks. From this, we can conclude, it is important to make end-users more aware of all the features an application contains which can help in their work. This could be done both in the interface itself, but also by a better introduction to it through a manual or a "tip-of-the-day" message. The last two options could not be part of this test and as with the first hypothesis a longer test period in an improved test setup could help.

The test participants who did see the generated script were divided into two groups: Those who successfully used it and understood how it worked, and those who did not understand it, and therefore chose not to use it. This shows that the generated script was to complicated for one who had just learned SDCASL, or maybe even to program. Because of the complication with the generated script, we can conclude that an application with such a feature, needs to make the user more aware of this feature.

7.2 Project Evaluation

In this section, we do a critical evaluation of our own work, both the overall process and with more focus on the testing phase.

7.2.1 Project Period

As with many previous projects we had a slow start—also called "student disease"—because of confusion about what to do in the project, and the problem statement was not clarified. In an attempt to overcome the slow start, we created a schedule for the whole semester which did help us. The schedule, however, were flawed in the sense that some activities were estimated to take less time than what they actually did. This let to many changes during the project period. There were a lot of pressure to finish the prototype in time before the usability testing, because it was postponed few times. Even with the pressure, we did manage to finish a decent prototype and ran two test rounds with a total of 15 participants, which we feel was a great success.

7.2.2 Test

In the short run and regardless of the results, our usability testing was a success, but during the test period we faced some issues which had some impact on our test results.

Test Participants

Firstly, the test participants. Recall that we had divided test participants into three groups with different programming knowledge level and had two test rounds. We originally planned to have an equal number of test participants in each test group, more precisely two test participants in each group for each test round. It all ended with an unbalanced distribution of the test participants, one missing in group 1 due to a cancellation and more than planned for group 2 and 3. This unbalancing was because of a mistake when we invited the test participants. In several cases, it turned out that the test participants did not match the test group, we had beforehand categorized them to be in. To avoid this mistake, we should make a short interview with the test participants before inviting them.

Test Monitor

During the usability testing, we changed the test monitor. It can be discussed whether this was right to do or not. In order to keep the same setup for the usability testing, the test monitor must be the same person for all test participants along with assignments, environment etc. If this should be a *real* usability testing, the test monitor must be a person with no relation to the testing product group in order to keep the neutrality between the testing product and the test participants. However, there are two downsides with such a test monitor in relation to DASA. Firstly, DASA is a prototype and can have bugs and errors. A person with no relation to our product has no idea how to help the test participants if a critical error occurs, and the application halts. Secondly, the test participants in our case were often family members and friends, so in the attempt to make these participants as comfortable as possible, we decided to switch the test monitor when needed. So an advantage of switching the test monitor in our case was that the test monitor knew the test participant, which meant that it was easier for him to know what the test person was thinking and could easier help the test participant in the right way.

A Too Narrowed Usability Testing Setup

DASA is a prototype for a concept that is supposed to be used over a longer period of time and gradually teach the end-users programming while assisting in their daily work. Our usability testing was limited to one hour for each test participant and used an environment which the test participants normally would not use. This setup simply cannot show for sure whether the concept works or not. It can give some small insights and ideas to some improvements, but not conclusive evidence of success. Ideally, DASA should be distributed to test participants who should work with it on a daily basis for a month. Afterwards they should be given a real programming assignment with e.g. a general purpose programming language to test whether they learned the fundamentals of programming. Another way is to implement the concept in a larger drawing application which features plugins, such as Paint.NET[1], and use the already existing community to find test candidates.

[1] Homepage for Paint.NET: http://www.getpaint.net/

7.3 Future Work

DASA is only a prototype and can be improved in several ways. First, there still exists bugs which of course needs to be removed in future versions. Second, we found several possible improvements during both the first and second test round which could be added such as auto completion and a better scripting enviroment. These will improve DASA both as an application, but also as a tool to teach end-users how to program. The concept behind DASA could also be used outside the application. A future work would be to implement the important parts of DASA into other applications. These applications should then be used by end-users for their daily work. Therefore, using a well-known application and add our concept as a plugin would be a preferred approach. This will allow us to properly test the concept in practice over a longer period of time.

7.4 Putting DASA Into Perspective

DASA is currently specialized in the drawing domain, but most of the main features can be inserted into another domain with none or very little changes. The scripting language, SDCASL, is designed to be domain independent. Only one construct, foreach, and one feature, the special variables, make use of domain specific rules, but the idea behind both of them can be used for most other domains. The commands used in the language all follow the same basic domain independent rules and, therefore, all of them can be changed or removed, and new commands can be added for any other domain. The same is the case for the types available. A color type might not be needed for one domain while another domain needs a color expressed with values between 0 and 1. The iterative task recognition algorithm, ITRA, only requires domain specific rules for the noise reduction as knowledge about the commands is required in order to specify what noise is. The rest of the algorithm in its current state is domain independent as it uses the commands and types for the comparison and calculations. It can, however, be improved if the comparison approach is changed to be domain specific. The current implementation of procedures is very domain dependent as it only works with three kinds of arguments: Positions, colors and penwidth. The idea behind the procedures can, however, still be moved to a different domain. One of the biggest successes for this project, the disclosure box, can be moved to any other domain without changes. It uses the rules set for commands and types and simply disclose these. What is needed for a new domain, is to find when these commands are completed and should be disclosed which is very different in e.g. a text-editing domain than in a drawing domain.

Appendix A

Bibliography

[BAM06] Margaret M. Burnett Brad A. Myers, Andrew J. Ko. Invited research overview: End-user programming. 2006.

[Bur09] Cook C. Rothermel G. Burnett, M. End-user software engineering. 2009.

[CHK+93] Allan Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, , and Alan Turransky. *Watch What I Do*. The MIT Press, 1993. [Online; `http://acypher.com/wwid/index.html`].

[Cyp93] Allan Cypher. Eager: Programming repetitive tasks by demonstration. In Allan Cypher, editor, *Watch What I Do*, page Chapter 9. The MIT Press, 1993.

[DB09] Ahn Tuan Nguyen Dao and Peter Heino Bøg. Specialization - programming technologies: End-user programming, 2009. Report for our previous semester. [Online; `https://services.cs.aau.dk/public/tools/library/details.php?id=1263152583`].

[DE95] Chris DiGiano and Mike Eisenberg. Self-disclosing design tools: a gentle introduction to end-user programming. In *DIS '95: Proceedings of the 1st conference on Designing interactive systems*, pages 189–197, New York, NY, USA, 1995. ACM.

[DiG96] Christopher John DiGiano. *Self-Disclosing Design Tools: An Incremental Approach Toward End-User Programming*. PhD thesis, The University of Colorado at Boulder, 1996.

[Har04] W. Harrison. From the editor: The dangers of end-user programming. *Software, IEEE*, 21(4):5–7, July-Aug. 2004.

[HS07] Jenny Preece Helen Sharp, Yvonne Rogers. *Interaction Design: Beyond Human-Computer Interaction (2 edition)*. Wiley, England, 2007.

[KJR00] Margaret M. Burnett Justin Schonfeld T. R. G. Green Gregg Rothermel Karen J. Rothermel, Curtis R. Cook. Wysiwyt testing in the spreadsheet paradigm: an empirical evaluation. 2000.

[KMA04] Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 199–206, Washington, DC, USA, 2004. IEEE Computer Society.

[KW09] Abraham R. Beckwith L. Burnett M. Erwig M. Lawrence J. Lieberman H. Myers B. Rosson M. Rothermel G. Scaffidi C. Shaw M. Ko, A. and S. Weidenbeck. The state of the art in end-user software engineering. 2009.

[LWDW03] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Mach. Learn.*, 53(1-2):111–156, 2003.

[Mit82] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[MKB06] Brad Myers, Andrew Ko, and Margaret Burnett. Invited research overview: End-user programming, 2006. [Online; accessed 10-September-2009, `http://www.cs.cmu.edu/~bam/papers/EUPchi2006overviewColor.pdf`].

[Nor88] D. A. Norman. *The Design of Everyday Things*. Doubleday, New York, NY, USA, 1988.

[nov]

[Ous98] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer Magazine*, March 1998. [Online; accessed 14-December-2009, `http://home.pacbell.net/ouster/scripting.html`].

[Pay00] Gordon W. Paynter. *Automating iterative tasks with programming by demonstration*. PhD thesis, The University of Waikato, Hamilton, New Zealand, 2000.

[Pot93] Richard Potter. *Just-in-Time Programming*. The MIT Press, 1993. Chapter 27 in [CHK+93].

[Qui93] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[RA09] Martin Erwig Robin Abraham, Margaret Burnett. Spreadsheet programming. 2009.

[SCT00a] David Canfield Smith, Allan Cypher, and Larry Tesler. Novice programming comes of age. In Henry Lieberman, editor, *Your Wish Is My Command*, page Chapter 1. Morgan Kaufmann, 2000.

[SCT00b] David Canfield Smith, Allen Cypher, and Larry Tesler. From the editor: The dangers of end-user programming. 2000.

Appendix B

Thoughts

From the experiences obtained during this project, we got some further thoughts about end-user programming. We believe that programming language should be a part of the primary school as a course equal to learning a second (foreign) natural language. In the following, we will clarify our thoughts about this idea.

Learning a programming language is just like learning a new natural language. The difference is that the programming language is an artificial language which makes it unnatural to human beings. One might believe that an end-user with good skills in speaking languages should be well prepared for learning a programming language, but that is just not the case. If we look at high level programming language such as C, C#, Java etc., they are using English words and phrases which should be recognizable for end-users. Basically, these high level programming languages are a very restricted form of the speaking language, English— they have no ambiguity. So the problems might be in the restrictions of the programming languages which many end-users cannot comprehend, because they are used to express themselves in a speaking language which has fluent rules. By this we mean that the speaking languages evolves over time, and the same rule might be changed later on when enough people accepts it. In addition, some words can be self-defined by some local people and have different meanings from other local people, but that does not mean, the definitions are wrong, it is just fluent.

If we look at the programming language as a restriction of a speaking language, we can relate it to mathematics. Mathematics is based on some final definitions and logical compositions, just like a programming language. It is also, in some sense, an artificial language. Our point here is not what equalities there is between mathematics and a programming language, but rather why most people in the world know mathematics—not

necessarily at a scientific level, but at the basic level. Mathematics is a need whether we like it or not. It is the fundamental building blocks for our own mental development such as learning to think logically. The question is if we are not facing the same need as mathematics yet?—although the computer age, compared to the age of mathematics, is still very young. Nevertheless, we have experienced ourselves how basic computer usage course was introduced late in the primary schools, but was later shifted to be introduced earlier in the primary schools after our own graduations. People realized that computer knowledge is a need even when at this time, computers were not so widespread as today. The computer course became a course of equal importance as music or art courses.

Today, computers are very widespread, and we find them everywhere—from cell phones to kitchen equipments to bathroom facilities etc.— and we need to communicate with these machines without realizing it. Of course, there are interfaces to help end-users communicate with the computers, but in restricted form and often not exactly what the end-users need, so they are obliged to compromise their actual needs. Computers are here to stay, and they get more advanced and powerful each day. We need to learn how to *really* communicate with these machines. Programming language is the *remedy*, as we have experienced, it is difficult for end-users to learn. Therefore, we suggest that programming language should be a course in the primary schools. An existing or perhaps a new common end-user friendly programming language can be used for the course. In this way, end-users are already prepared to think programmatic from their young ages, just like when they start learning basic mathematics or basic language grammar.

Appendix C

Scripting Language Syntax

In this appendix we describe the entire syntax of our scripting language, SDCASL, using the Backus-Naur Form (BNF). It consists of rules, which state that the left side (a *non-terminal* which is a name enclosed in '<' and '>') can be replaced by (::=) the right side (any non-terminals and terminals). Any key or expression that is not found on the left side is called a *terminal*, hence any that can be found is known as non-terminals. There can be an unspecified amount of whitespaces on both sides of any terminal and non-terminal, and therefore only whitespaces inside an expression (what is enclosed by ") is important. The line character (|) is used as a choice between either side of it, meaning one the left side can be replaced by several sequences of terminals and nonterminals each divided by a line. The terminal <EOL> stands for the end of a line (or more correctly an end-of-line character). The terminal ϵ (epsilon) is used to represent and empty sequence, or no terminals and non-terminals.

Script

A script written in the script box of DASA, and as the content of a procedure, starts with the rule <script>. Listing C.1 shows the first basic elements of such a script.

```
1 <script>     ::= <statement> <statementlist>
2 <statementlist> ::= <EOL> <statement> <statementlist>
3                   | ϵ
4 <statement>  ::= <command>
5                   | <structure>
6                   | <assignment>
7                   | <procedure>
8                   | <comment>
9                   | ϵ
```

Listing C.1: The basic syntax for a script.

The rules that are not specified in the listing will be specified later in this appendix.

Command Line

A script written in the command line only consists of one line. Instead of the <script> rule it starts with the <commandline> rule and can be seen in Listing C.2.

```
<commandline> ::= <command>
                | <assignment>
                | <procedure>
```

Listing C.2: The syntax for the command line.

Structures

Structures are the basic language elements and the definition of these can be found in Listing C.3.

```
<structure>  ::= <for> | <if> | <foreach> | <forall>
<for>        ::= "for" <variablename> "=" <expression> "to" <expression>
                     "do" <EOL> <statements> <EOL>
                     "end"
<if>         ::= "if" <condition> ("then"|"do") <EOL> <statements> <EOL>
                     "else" <EOL> <statements> <EOL>
                     "end"
               | "if" <condition> ("then"|"do") <EOL> <statements> <EOL>
                     "end"
<foreach>    ::= "foreach" <objecttype> "do" <EOL> <statements> <EOL> "end"
<forall>     ::= "forall" "do" <EOL> <statements> <EOL> "end"
<objecttype> ::= "all" | "selected" | "rectangle" | "ellipse" | "line"
<assignment> ::= <variable> "=" <variablevalue>
<variablevalue> ::= (a legal type)
<procedure>  ::= <procedurename> <argumentlist>
<variable>   ::= <variablename>
               | <specialvariable>
<variablename> ::= [A-Za-z0-9]
<procedurename> ::= [A-Za-z0-9]
<comment>    ::= "#" (the comment string)
<specialvariable> ::= (a known special variable name)
```

Listing C.3: The syntax for the language structures.

Command

The syntax for any command that we have defined in DASA, such as `rectangle` and `set`, are specified in Listing C.4.

```
<command>      ::= <commandname> <argumentlist>
<commandname>  ::= (a known command name or synonym)
<argumentlist> ::= <argument> <argumentlist>
<argument>     ::= <argumentname> "=" <variablevalue>
<argumentname> ::= (a known argument name or synonym)
```

Listing C.4: The syntax for one command.

Condition

A condition is used in an if-structure and is evaluated to either `true` or `false`. Its definition is shown in Listing C.5.

```
<condition>     ::= <expression> <booloperator> <expression> <conditionlist>
<conditionlist> ::= <conditionoperator> <expression> <booloperator>
                        <expression> <conditionlist>
                  | ϵ
<conditionoperator> ::= "&" | "|" | "&&" | "||" | "and" | "or"
<booloperator>  ::= "=" | "==" | "!=" | "<>" | "<" | ">"  | "<=" | ">="
```

Listing C.5: The syntax for the if-conditions.

Expression

An expression is defined in Listing C.6 and is used at several places in the syntax.

```
<expression>  ::= <negative> <expressionpart> <expressionlist>
<expressionlist> ::= <operator> <negative> <expressionpart> <expressionlist>
                   | ϵ
<expressionpart> ::= <variable> | <random> | <number>
<negative>    ::= "-" | ϵ
<random>      ::= "random" <number>
<number>      ::= (an integer or a double)
<operator>    ::= "+" | "-" | "*" | "/"
```

Listing C.6: The syntax for expressions.

Appendix D

Scripts

This appendix contains scripts used in the thesis. These scripts are all written in SDCASL used in DASA. For further explanation of the syntax and semantics of SDCASL refer to Appendix C and our previous report [DB09].

Attribute Differences

The following script is used to illustrate how performing the same actions can generate two very different results from the algorithm depending on what parameter set is used. An illustration of this can be seen in Figure 5.10.

```
# parameter set: from and to
for x=0 to 18 do
  line from=100,200 to=x*-3+200,x*-18+200
end

# parameter set: from, length and angle
for x=0 to 18 do
  line from=100,200 length=100 angle=x*-10
end
```

Listing D.1: Drawing a circle of lines using two different parameter sets.

Appendix E

Test Material

E.1 Introduction Questions

This is the introduction questions used to categorize the test participants in the appropriate testing groups.

- What is your job function?
- Do you use a computer on a daily basis?
- Do you know what programming is?
- Do you know what a programming language is?
- Have you ever tried to create a program or solve a task by programming?
- Do you use spreadsheets e.g. Excel?
- Are you familiar with creation of mathematical expressions in spreadsheets?
- Did you know, mathematical expressions in a spreadsheet is actually programming?
- Do you want to learn how to program?
- Have you used a drawing program e.g. Photoshop or AutoCAD?

E.2 Assignments

Following are the texts from the assignments used for the test in the original Danish version. These are the assignments as they looked for test round 2, where assignment #4 had been changed. All the figures can be seen in the thesis, Section 6.4.5.

Opgave 1 - Basale Figurer

Dette er den første opgave i en serie af 8 opgaver. I denne opgave skal du lære at benytte tre basale tegneværktøjer, så du kan gøre dig bekendt med tegneprogrammet. Når du tegner, er det vigtigt, at du lægger mærke til `Disclosures`-boksen. Boksen findes i bunden af programmet.

Opgaven består i at tegne: en rektangel, en cirkel og en streg.
Hvor stor hver figur skal være, er helt op til dig.

Opgave 2 - Farvelægning

Dette er opgave 2, som omhandler farvelægning af figurer. Opgaven er en fortsættelse af opgave 1. Det er igen vigtigt, at du lægger mærke til `Disclosures`-boksen, som du kan finde i bunden af programmet.

Du skal farve figurernes streger med følgende farver:

- Rektanglet skal være rød.
- Cirklen skal være blå.
- Stregen skal være grøn.

Opgave 3 - Brug af Kommandoer

Dette er opgave 3, og du skal i denne opgave tegne med præcision, så vi vil introducere dig for yderligere et værktøj, kommandolinjen. Det er igen vigtigt, at du lægger mærke til `Disclosures`-boksen.

Gør følgende:

- Tegn, vha. kommandolinjen, en cirkel med radius 250 og centerpunktet skal være i punktet 300,300. Du må ikke bruge musen!

- Brug musen til at flytte cirklen lidt til højre.
- Flyt cirklen tilbage på dens tidligere position; Positionen skal være præcist den samme. Du må IKKE bruge fortryd.
- Farv cirklen blå vha. en kommando.
- Sæt cirklens tykkelse til 5 pixel vha. en kommando.

Opgave 4 - Lav en Procedure

Dette er opgave 4. Du skal i denne opgave introduceres for `procedurer`, som er en gruppering af kommandoer. Du starter med at tegne et sæt af flere figurer i samme farve. Sættet danner et billede af en cykel. Derefter skal billedet tegnes med andre farver og positioner.

Gør følgende:

- Tegn et billede af en sort cykel (se eksemplet for neden). Du må gerne bruge musen.
- Lav en procedure ud fra kommandoer, som er blevet vist i disclosures-boksen. Husk farven og positionen skal kunne ændres!
  Hjælp: se om der er noget, du kan bruge ved at højreklikke i `disclosures`*-boksen.*
- Kald proceduren 5 gange med forskellige farver og positioner.

Opgave 5 - Brug af Script

Dette er opgave 5 og er den sidste introduktionsopgave i testen. Denne opgave består i at ændrer på et eksisterende billede ved hjælp af et script. Du starter med at indlæse et eksisterende billede, som vi har lavet. Boksen i øverste højre hjørne bliver din primære skriveboks.

Billedet indeholder røde og grønne firkanter, der er placeret hhv. over og under den sorte streg. Der er desuden en masse streger og figurer for at besværliggøre opgaven.

Du skal gøre følgende:

- Alle røde firkanter skal farves blå.
- Alle grønne firkanter skal farves rød.
- Alle firkanterne skal bytte plads således, at de blå flyttes ned under den sorte streg, og de røde flyttes oven over stregen.
  Hjælp: Farven på firkanterne kan bruges som en betingelse.

Opgave 6 - Tegn Cirkler

Dette er opgave 6 og er en selvstændig opgave. Du bestemmer selv, hvordan du løser opgaven ud fra beskrivelsen og kriterier for opgaven. Du skal i denne opgave lægge mærke til boksen nederst i højre hjørne.

Gør følgende:

- Du skal nu tegne 30 cirkler af stigende størrelser. Du må gerne bruge musen.
- Start med en lille cirkel og tegn derefter yderligere cirkler af større radius.

Opgave 7 - Den Vilde Skov

Dette er opgave 7. I denne opgave skal du bruge de værktøjer, du har afprøvet undervejs i de forrige opgaver. Det er dog helt op til dig, hvordan du vælger at løse opgaven.

Du skal nu tegne et billede efter følgende beskrivelser og kriterier.

- Billedet illustrerer en skov med 30 træer, et hus midt i skoven, samt en sol.
- Hvert træ laves vha. et rektangel og en cirkel, der hhv. er stammen og kronen på træet. De skal placeres således, at de danner et billede af en skov.
- Huset laves vha. et rektangel og to streger, der hhv. er selve huset og taget. Huset skal have en dør, samt to vinduer. Disse laves med rektangler.
- Billedet skal have en sol, som laves vha. en cirkel.

Opgave 8 - Farvelægning af Den Vilde Skov

Dette er opgave 8 og den sidste opgave i testen. Opgaven her er en fortsættelse af opgave 7. I denne opgave skal du igen bruge de værktøjer, du har afprøvet undervejs i de forrige opgaver. Det er dog helt op til dig, hvordan du vælger at løse opgaven.

Du skal nu farvelægge billedet med den vilde skov fra forrige opgave efter følgende beskrivelser og kriterier.

- Husets tag skal være rød (de to streger).
- Alle trækroner skal være grønne.
- Alle træstammer skal være brune.
- Solen skal være gul, også dens stråler.

E.3 Questionnaire

This is the original Danish version of the questionnaire given to each test person from the second test round, after they finished the assignments.

Del 1

1. Hvad er den nederste venstre boks til?

(a) At tegne
(b) Scripting
(c) Udførte kommandoer
(d) Auto-genererede script
(e) At vise oprettede procedurer
(f) Ved ikke

2. Hvad er den øverste højre boks til?

(a) At tegne

(b) Scripting

(c) Udførte kommandoer

(d) Auto-genererede script

(e) At vise oprettede procedurer

(f) Ved ikke

3. Hvad er den nederste højre boks til?

(a) At tegne

(b) Scripting

(c) Udførte kommandoer

(d) Auto-genererede script

(e) At vise oprettede procedurer

(f) Ved ikke

4. Hvad vises der i "Dislosures"-boksen forneden?

(a) Tilfældigt tekst

(b) Kommandoer

(c) Programmets statusbeskeder

(d) Ved ikke

5. Hvad er en "procedure"?

(a) En kommando

(b) Et programmeringssprog

(c) En gruppering af handlinger

(d) Ved ikke

6. Hvad er et "script"?

(a) Et talesprog

(b) En række instruktioner til computeren

(c) Et computersprog

(d) Ved ikke

7. Hvad betyder `set color = red`?

(a) At farve en figur rød

(b) At tegne en firkant

(c) At lave en procedure

(d) Ved ikke

8. Hvad gør kommandoen `rec color=red w=100 h=50 pen=5`?

(a) Tegner en cirkel med farven rød

(b) Tegner et rektangel med 100 i bredde og 50 i højde

(c) Kommandoen virker ikke

(d) Ved ikke

9. Hvad er en if-sætning?

(a) Et loop

(b) En kommandor

(c) En betingelse

(d) Ved ikke

10. Hvad er et "loop/løkke"?

(a) En metode til at gentage handlinger flere gange

(b) En kommando

(c) Et script

(d) Ved ikke

11. Hvis der er 100 firkanter af forskellige størrelser, og du vil kun farve dem med en hvis bredde. Hvad gør du?

(a) Jeg farver de udvalgte firkanter med musen

(b) Jeg kan benytte et for-loop med en betingelse for firkantens bredde

(c) Jeg udfører en kommando på de udvalgte firkanter

(d) Ved ikke

Del 2

1. Hvor nyttigt syntes du, disclosures er?

Ubrugeligt - Unyttigt - Nyttigt - Meget nyttigt - Ved ikke

2. Hvor nyttigt syntes du, procedurer er?

Ubrugeligt - Unyttigt - Nyttigt - Meget nyttigt - Ved ikke

3. Hvor nyttigt syntes du, scripting er?

Ubrugeligt - Unyttigt - Nyttigt - Meget nyttigt - Ved ikke

4. Hvor nyttigt syntes du, auto-genereringen er?

Ubrugeligt - Unyttigt - Nyttigt - Meget nyttigt - Ved ikke

5. Hvor nyttigt syntes du, vores værktøj er til at hjælpe dig gennem opgaverne?

Ubrugeligt - Unyttigt - Nyttigt - Meget nyttigt - Ved ikke

6. Hvad er dit generelle indtryk af hele applikationen?

Ubrugeligt - Unyttigt - Nyttigt - Meget nyttigt - Ved ikke

7. Kan du se andre former for applikationer, hvor disse funktionaliteter kunne være nyttige?

(a) Nej

(b) Ja, hvilke?

(c) Ved ikke

Appendix F

CD-ROM

The CD-ROM that comes with this thesis contains the following:

- This thesis in PDF format.
- Figures from this thesis in their original format.
- Source code and binaries for the product, DASA.
- The report for the previous semester in PDF format (SW9).
- Source code for the previous semester product (SW9).
- Binaries for DASA used during both test rounds.
- Example scripts written in SDCASL.
- Videos of our solutions to the test assignments.
- Raw test data.

Appendix G

Danish Synopsis

Following is the danish translation of the synopsis for this project.

Temaet for dette projekt er end-user programmering og vi fokusere på hvordan vi kan lære programmering til end-users der aldrig har programmeret før. Vi udvikler et værktøj (prototype) som en udvidelse til en tegne applikation sammen med et domæne specifikt script sprog. Prototypen bruger principperne ved self-disclosing i konceptet Læring ved Observation. Målet er at lærer end-users at programmere ved brug af det designede script sprog mens de bruger tegne applikationen. Ydermere udvikler vi en algoritme til at genkende iterative opgaver således at et script kan blive genereret til at færdiggøre opgaven. Prototypen og script sproget er udvidelser af det arbejde vi foretog i vores tidligere projekt, SW9.

Vi slutter med en brugbarheds test der beviser dele af vores hypoteser, såsom at self-disclosing fungere som et lærings koncept. Men, vi beviste ikke dette helt for den gruppe af end-users vi oprindeligt havde udtænkt, men mere for end-user programmøre. Overordnet er projektet og testen en succes og vi finder flere interessante konklusioner omkring emnet end-user programmering.