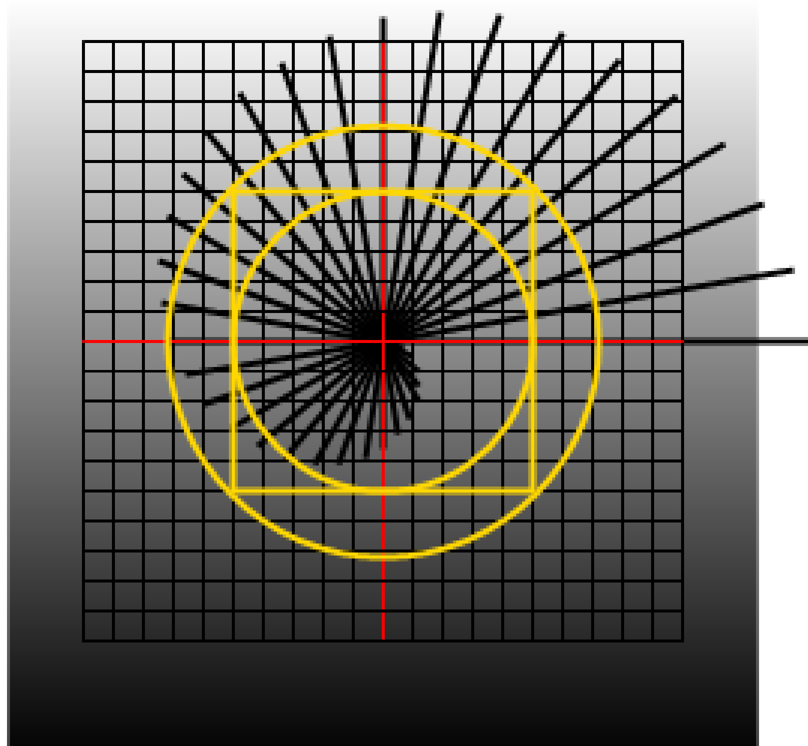
Specialization: Programming Technologies

End-user Programming

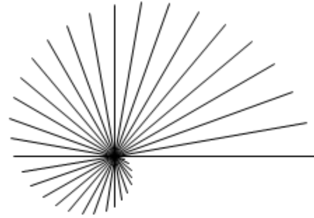


Aalborg University:

Software Engineering, SW9, Fall 2009

Anh Tuan Nguyen Dao & Peter Heino Bøg

December 18th, 2009



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

Title:
End-User Programming

Theme:
Specialization
Programming Technologies

Project Unit:
Software Engineering
9th semester, Fall 2009

Project Group:
d509a

Participants:
Anh Tuan Nguyen Dao
Peter Heino Bøg

Supervisor:
Kurt Nørmark

Report Count: 4

Page Count: 53

Appendix: A-J

Finished: December 18th 2009

Synopsis:

This report is the first part of two and documents the thesis in the area of end-user programming. The purpose of this report is to give an introduction to end-user programming area and addressing a problem in this research area. There are two groups of end-users with different goals. The first group wants to create the software itself, and the second group wants to use programming as a tool. The problem is then how we can help end-users to start programming.

Our solution for the end-user problem is a tool which uses the concepts of learning by observation and programming by demonstration, and it is aimed at the second group of end-users. The main goal of our tool is to help end-users to learn to program while they use their application.

In this part of the thesis, we have implemented a platform which will be the cornerstone for the next part. This includes a simple and expandable scripting language.

The contents of this report is freely available, but publication (with reference source) may only happen after agreement with the authors.

Preface

This report documents a project made by the Software Engineering group d509a at the Department of Computer Science at Aalborg University. It is made on the 9th semester and the group is part of the Programming Technology department. The theme of this semester was “End-User Programming”. The project period started at September 3rd 2009 and ended December 18th 2009.

The reader is expected to have a basic knowledge in C# and computer science in general.

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

To ease readability and focus on the important parts, some code examples may have been modified in this report and, thus, may differ slightly from the actual code.

References to literature and sources in this report are written in square brackets, such as [DE95], and references to figures and tables in the appendix are written with a letter and number, such as Listing G.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 J.

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

Anh Tuan Nguyen Dao

Peter Heino Bøg

Contents

1 Introduction **7**
1.1 Definition of End-user 8
1.2 The Experiment Application 9
1.3 Report Structure 10

2 Analysis **11**
2.1 The Gap Between the Human Brain and the Computer 11
2.2 User Actions 13
2.3 End-user Programming 14
2.4 Interactive and Programmatic Access 18
2.5 Self-Disclosing 19
2.6 The Difference Between Genders 21
2.7 Dangers of End-user Programming 22
2.8 Related Applications and Systems 23
2.9 Summary 27

3 Problem Statement **29**
3.1 Motivation 29
3.2 Vision 31
3.3 Delimitation 31
3.4 Questions 32

4 Experiments **33**
4.1 Preliminary Work 33
4.2 Experiment 1: Disclosing Commands 34
4.3 Experiment 2: Sloppy Command Line 37
4.4 Experiment 3: Keyword Arguments 42
4.5 Experiment 4: Scripting Language 47
4.6 Summary 50

5 Epilogue **51**
5.1 Conclusion 51
5.2 Evaluation 52
5.3 Future Work 53

A Bibliography i

B DrawTools iii

C AutoCAD Commands v

D SVG Commands vii

E Types ix

F Commands xi

F.1 Commands for Drawing Shapes xi

F.2 Commands for Manipulating Shapes xiv

F.3 Commands for Application Handling xvi

G Sloppy Parsing Algorithm xix

H Implementation of Script Parsing and Interpretation xxiii

I Scripts for Repetitive Task Examples xxvii

J CD-ROM xxxi

Chapter 1

Introduction

In the modern time, everyone has a personal computer at home. Such a machine is very powerful for processing information and should make users' lives easier both at home for entertainment and at their everyday job. Even when normal users work with computers everyday, the real power in a computer is unrealized. Most users today interact with the computer through software applications developed by other users. We call the users of software for *end-users* and the users who develop applications for *programmers*. The limitations of the interaction is a myopic view of computers, like Alice looking at the garden in Wonderland through a keyhole [SCT00a]. The true power of a computer is to do automated repetitions very fast, but if applications does not provide tools to such automated task solving, the end-users are limited to do the task one step at time, repeatedly.

In general, end-users are not programmers and can be e.g. accounting managers, teachers, waitress, secretaries, photographers, architects etc. Many of them encounter repetitive tasks in their everyday job, e.g. a photographer whos job is to take photos and edit them. Imagine the photographer has thousand of photos, editing each of them with the same photo effects would be a tedious task. Unless a professional programmer had developed such editing tool which automatically applies the changes to each photos, the photographer must do it manually one photo at time.

Typically, end-users are not interested in learning a program language, because it is difficult [SCT00a]. It takes years to learn a foreign language, and a programming language is an artificial language which seems to be unnatural for non-programmers, hence, making it very difficult for end-users to learn it. Instead, they are more interested in solving their tasks fast without spending too much time on learning how to solve them. The difficulty lies in the differences in the representations of a problem between the human brain and the computer, also called the Grand Canyon Gab [SCT00a] between human and computer. There are two ways to bridge this gap: (1) move the end-users closer to the system or (2) move the system closer to the end-users [SCT00a].

The history of end-user programming has a long tail, but was unstructured until recently. End-user programming has always been there, but is unnoticed by most people, since programming in the early ages of computer science was reserved for scientists. However, we have seen that the development of computer interaction has gone from simple instruction cards to programming languages and later to a graphical user interface (GUI). The number of end-users has also grown along with the development of computer interaction. The earliest application with end-user programming in mind and which we have seen is the tool Eager from 1991 by Allen Cypher and uses the programming by demonstra-

tion (PBD) paradigm. However, the idea of such a tool was already introduced in 1989[1]. Since Eager, many other PBD applications have arisen, Stagecast Creator (1999) also by Allen Cypher, ToonTalk (1999) and Scratch (2007), but they are all aimed for children. In this project we will not focus on children but we can still use these applications as an inspiration.

In 2003 a number of universities in the United States along with IBM started a consortium, End-Users Shaping Effective Software (EUSES) Consortium, which is a start of a structuring of the end-user programming research area. Within in the start year there were only two publications. At the time of writing, there are over 250 publications[2], and the number of publication each year is increasing. The goal of EUSES Consortium is to develop and investigate end-user software engineering technologies for enabling End Users to Shape Effective Software. The motivation lies in 55 million end-user programmers in the United States as of 2005 compared to the 2.75 million professional programmers.

Recall the two ways to bridge the Grand Canyon Gap, we will in this project introduce a tool which uses the second way to learn end-users to use the first way. The tool uses PBD and the concept of learning by observation. The whole project is divided into two parts. In the first part, which is what this report is about, we will make a platform which is integrated into a drawing application, DrawTools. Therefore, we will do experiments beside analysis in order to implement our platform into DrawTools. In the next part, we will make research on repetitive task pattern in order to automatically create scripts from end-user interactions.

For the remaining of this chapter, we will define the term end-user and their different goals. Understanding their goals is important, since our design of tool will be affected by the target group. Furthermore, we will give an introduction to the application, we used in our experiments and finally the structure of the report.

1.1 Definition of End-user

Brad Myers, Andrew Ko, and Margaret Burnett [MKB06] give some definitions for different levels of programmers. The first is the **professional programmer** which is someone whose primary job function is to write or maintain software. These programmers typically have significant training in programming, e.g. with a bachelor in computer science or higher. The second level is **novice programmer** which basically is a professional programmer in training. A novice programmer will end up with the same theory knowledge as the professional programmer already have. Finally, there are **end-user programmers**, they do write programs, but it is not their primary job function and usually they have not had any significant training in the area. Instead, end-user programmers might be self-taught programmers, or **dilettantes** as they are called by Warren Harrison [Har04]. Seen from another perspective, the professional programmers are paid to create software or maintaining software while end-user programmers are programming to support their goal in some expertise domain.

Notice that there is a difference between an *end-user* and an *end-user programmer*,

[1] We have tracked it down to this year by looking at the publication list of Allan Cypher at `http://acypher.com/Publications/`

[2] See the full list of publications at `http://eusesconsortium.org/pubs/publications.php`.

since the end-user is the consumer of a software application without any programming skill whereas the end-user programmer has the programming skill. In this report, we will use the term end-user for users who know how to use a computer and want to learn to program, and the term end-user programmer for those users who can program, but lack knowledge in software engineering theory.

Furthermore, we can categorize programmers (shown in Figure 1.1) as a parent to three sub groups, professional, novice and end-user. It is important to clarify that end-users have different goals with programming which is shown as two sub groups to end-user group. One sub group wants to program in order to create a program. The other group wants to program only to solve a problem.

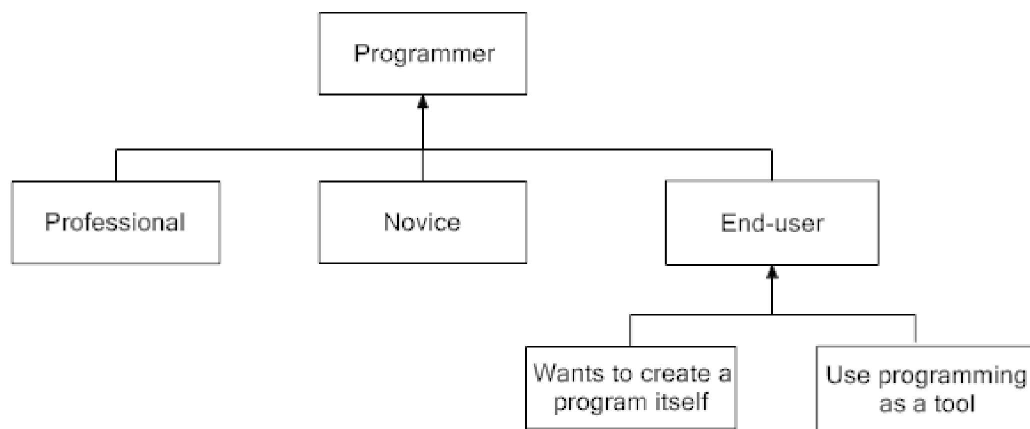


Figure 1.1: An illustration of how programmers can be divided into professionals, novice and end-user. End-user programmers can be divided further into their goals with programming.

1.2 The Experiment Application

In this project, we do not only read papers and make analysis, but we also do experiments to get a better understanding of how to create our tool. These experiments should end with a platform which will be used in the next project when we will create the actually tool. To make it as easy for us as possible, the experiments will all be based on an existing, simple and open source application, DrawTools. A screenshot of DrawTools is shown in Figure 1.2.

DrawTools is a drawing application created by “Alex Fr” for a tutorial and released on *The Code Project*.[3] It is written in C# using Visual Studio. This application has been chosen, because it is open source and is well-written in C#. In addition, the DrawTools application is very simple both in user interface and the tools for drawing. The simplicity of DrawTools lets us focus more on the core idea of disclosing end-user interactions. Furthermore, a drawing software program has been chosen, because of the extensive mouse activities in such a program compared to e.g. a text editor. For the interested reader, a class diagram of DrawTools is to be found in Appendix B.

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

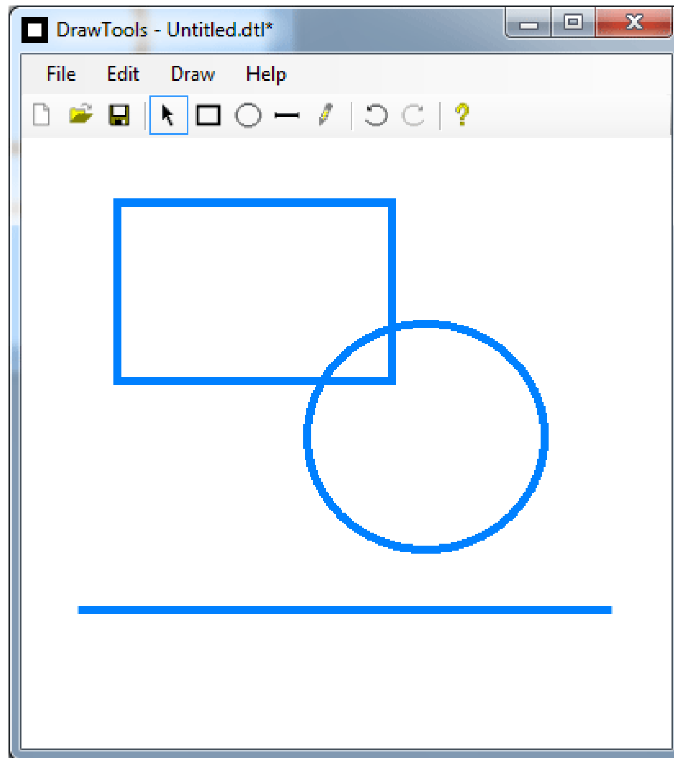


Figure 1.2: A screenshot of the DrawTools application.

1.3 Report Structure

Here is an overview of how the rest of this report is structured:

Chapter 2, Analysis, will analyze the area of end-user programming.

Chapter 3, Problem Statement, will describe the problem we would like to solve during this project period in greater details which include delimitations and questions.

Chapter 4, Experiments, will describe several experiments we made. Each experiment will describe the design and implementation phases and end with a result.

Chapter 5, Epilogue, will contain the final discussion about the area and our solution. It will also include a description of what we might need to do on the next semester.

Finally we have an appendix which includes the bibliography and further details about our implementations from the experiments for the interested ones.

Chapter 2

Analysis

In this chapter we describe our analysis of the area of end-user programming. First we describe the gap between the human brain and the computer. This is one of the main issues of why the area of end-user programming is not straight forward. Next we explain the terms of user actions and different levels of events. This is used in later sections and chapters. Then we give an overview of end-user programmers and the different goals end-user programmers have. In addition, we will give a list of end-user programming approaches.

There are different ways to communicate with an application, and we describe this as interactive and programmatic access to the application. We analyze this because our experiments will add a programmatic access to an application which only features an interactive access. Our tool is based on the idea of self-disclosing which we will analyze and describe some properties and guidelines on how this should be added to an application.

In general, the majority of professional programmers is male, but end-users are software consumer with different jobs, hence, the balance between the genders seem to be equal. We will here take a look at how the gender of an end-user might influence the way they work with an application. In addition, we will also describe some of the dangers of end-user programming. Finally, we will introduce to applications which are inspirations to our experiments.

2.1 The Gap Between the Human Brain and the Computer

According to the consortium EUSES, there is be 55 million end-user programmers in USA at year 2005, and how large is the number if we count the end-users. Clearly, there must be many more end-users than end-user programmers, since they are defined as software application consumers and non-programmers. The big vision is then to get some end-users, if not all, to become end-user programmers, but this seems to be very difficult, otherwise end-user programming research area would not exist today. In the following, we will look at a phrase called Grand Canyon Gap as Don Norman introduced in his book from 1986 [SCT00a]. He described the difference in how a human brain represents a problem and how a computer understands and accepts the same problem. To describe the representation of a problem, there are two approaches; Sloman's and Bruner's approach, developed by Aaron Sloman and Jerome Bruner, respectively. Understanding these approaches can give us an idea of why it is difficult for an end-user to learn a programming language.

Sloman's Approach

Aaron Sloman divided representations of a problem into two general types in his paper from 1971 [Slo71]: analogical and Fregean representations. The Fregean representation is using predicate calculus statements, hence, the name Fregean, named after the inventor of predicate calculus, Gottlob Frege [Slo71]

In the first type, the analogical, the representation is self-descriptive. It means that a person can get information e.g. about relations between objects without any domain specific knowledge. A map is a good example of a analogical representation, since it contains a lot of information about relations between streets, rivers, cities etc. Figure 2.1 shows a map of Aalborg and its surrounded areas. As we see in the example, Aalborg University is south of Nørresundby and east of Aalborg.

Figure 2.1: An example of an analogical representation; A map is very self-descriptive.

The second representation, the Fregean, is using "function-signs" and "argument-signs" [Slo71] which together represents the relationship between objects. Take the map example again, the representation of a map in a Fregean way could be done as in Listing 2.1. Each symbol (u, a, n) is defined as a place or city, and the functions (east, south) describes the relations between the symbols.

```
1 u: "Aalborg Univerty"
2 a: "Aalborg"
3 n: "Nørresundby"
4 east(u, a)
5 south(u, n)
```

Listing 2.1: An example of how a map can be represented in a Fregean way.

Bruner's Approach

The psychologist, Jerome Bruner, gave three ways to how we can represent any domain of knowledge [BR96]:

1. **'Enactive' representation**: *"By a set of actions for achieving a certain result."* This representation means for example that you cannot learn to ride a bike by reading a book, you have to do it yourself.

2. **'Iconic' representation**: *"By a set of summary images or graphics that stands for a concept without defining it fully."*
   This means that you learn what a car is by seeing pictures of cars.

3. **'Symbolic' representation**: *"By a set of symbolic or logical propositions drawn from symbolic system that is governed by rules or laws for forming and transforming the propositions."*
   A speed limit sign is a symbolic representation which the appearance has been decided through rules and laws.

The first two are analogical representations, and the third is the Fregean representation [SCT00a].

The Grand Canyon Gap

Human beings have a skill to understand enactive and iconic (or analogical) representations very easily, because to mimic and to recognize is the part of their nature. The symbolic or the Freagean representation requires, however, experiences in our daily lives, hence, children at younger ages have difficulties with understanding such kind of representations. Although, we learn to understand Fregean represenations throughout our lives, it is still difficult to learn to program, because going from an analogical to a Fregean representation requires experience in a rational way of thinking. At one side the human brain mostly uses the analogical representation, but at the other side the computer will only accept the Fregean representation—there is a gap between a human brain and a computer. For an end-user, this gap is as wide as Grand Canyon, according to Don Norman.

There are two ways to bridge the gap: (1) move the end-user closer to the system, or (2) move the system closer to the end-user [SCT00a]. The first way is known as general purpose language or system programming language which move the programmers closer to the system, meaning that we let the programmers control the computer through the programming language. The second way is to create a tool or technique to let the end-users control their computers through an analogical representation.

2.2 User Actions

In the next section, and others to come, we will talk about user actions and different levels of events. Therefore we need to introduce these here.

When a user is interacting with a computer he performs many input **events**, e.g. through the use of a mouse and keyboard, and an abstraction of these events can be made. Alan Cypher[Cyp93a] divides them into three levels: **low-**, **mid-** and **high-level**. The low-level events are the very basic inputs such as moving the mouse, clicking the mouse and pressing a key on the keyboard. When we combine several of these low-level events we get a mid-level event, or a **user action**. These could include *clicking on button at (x,y)*, *closing the application* or *scrolling one page down*, but they can all be generalized to fit any application. The Eager application described in [Cyp93a] used an environment called HyperCard, which again used Apple Event which were available on Mac OS. The HyperCard environment sends out much more complex events such as *Copy words 2 through 3 ("Trial info") of line 1 of background field 1 of card 2 of stack "Cali:Eager Demo:Mail*

Messages". These are high-level events and they are each an abstraction over several user actions, here e.g. *put focus on a textbox*, *marking some of the text* and *issuing the copy-command*. These high-level events are depending on the application in use. Figure 2.2 illustrates how these three levels are related.

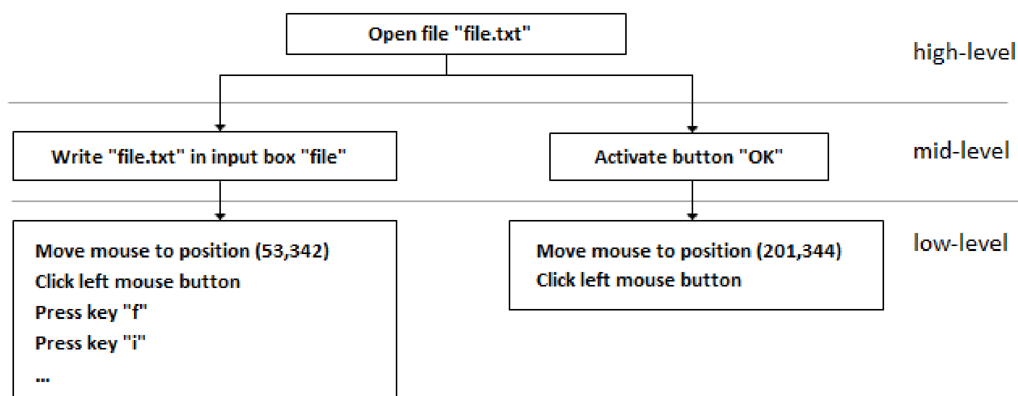


Figure 2.2: An illustration of how a high-level event such as opening a file called "file.txt" could be divided into two mid-level events. Each of these mid-level events are again divided into several low-level events.

2.3 End-user Programming

End-user programming research is about to empower end-users with programming skill to solve repetitive activities or obtain their goal with help of their personal computers. Here programming is not meant as usual heavy application programming which professional programmers do (also called semantic programming), but rather as techniques for end-users to customize an application or the computer to solve their tasks faster, easier or perhabs giving better results.

Modern software applications are huge and aim for large number of different end-users, thus, application designers cannot fulfill all end-users' need. The result is that either end-users have to learn a programming language like C or must live with the limited interaction with their computers through these applications. The latter situation is a waste of computer power, and the first situation is difficult to realize. In the following, we describe each approach, starting with the simplest one.

Preferences Programming

Preferences are pre-defined alternatives provided by the application designer. They are used to accommodate the needs 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 one application. End-users are offered the option to show numbers in different currency format or a cell can be interpreted as a number or text which defines the alignment of the content. In general, preferences gives only a limited number of alternatives and is not generalized enough to be considered as traditional programming.

Figure 2.3 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 2.3 also shows that there is several other pages like this one with many more preferences. 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.

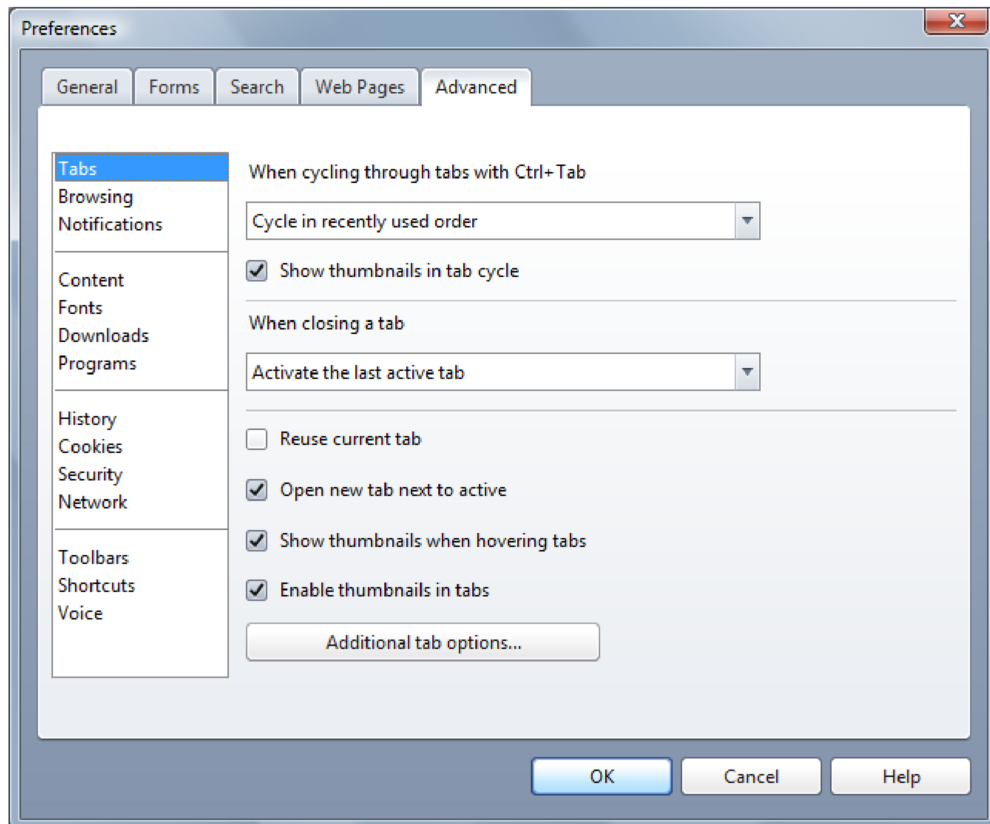


Figure 2.3: The preferences dialog available in the Opera Browser.

Programming by Demonstration

Programming by demonstration, also known as *programming by example*, is actually macro recorders which instead of recording low-level events, produce 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*. By doing this, the end-user is teaching the application what to do by demonstrating it. This approach is often used in the area of robotics and small game development. PBD is found in many large software application, such as Microsoft Word. In this application, this approach help solving repetitive tasks such as large and complex text formatting.

A variant of PBD is macro recorders. Macro recorders are used to record users inputs for later to repeat them. These recorded inputs are, however, low-level events (Section 2.2)

which makes it hard to reproduce the exact same effect, if for instance the window of the application has been moved.

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 users create rules by demonstrating the simulation the rules, and by combining the rules the users can create small games. Figure 2.4 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 in the game and what should happen when the pirate hits a chest. The simulation also supports use of variables which gives users opportunities to create more complex rules. Other similar applications as Stagecast Creator is ToonTalk and Scrath which also are aimed for children.

Figure 2.4: A picture of a pirate game in Stagecast Creator.

Spreadsheet Programming

Spreedsheets are one of the most used systems in world, both businesses and individuals use spreedshets to create simple calculations or even large and complex financial models [ABE09]. 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, and in essence the end-users do firt-order functional programming [ABE09]. Spreadsheet programming does not require any programming skills, since end-users create functions as mathematical expression which is known by far the most people of the world. In addition, spreadsheets also constantly give feedback to the end-users as they make progress even when the program is not complete or contain errors which means the end-users can freely keep making progress without getting interrupted. However, missed error corrections can

problems as we shall see in Section 2.7.

Microsoft Excel is an example of a very popular spreadsheet application, shown in Figure 2.5[1] This example shows how rich the Microsoft Excel is with functionalities for creating charts and models.

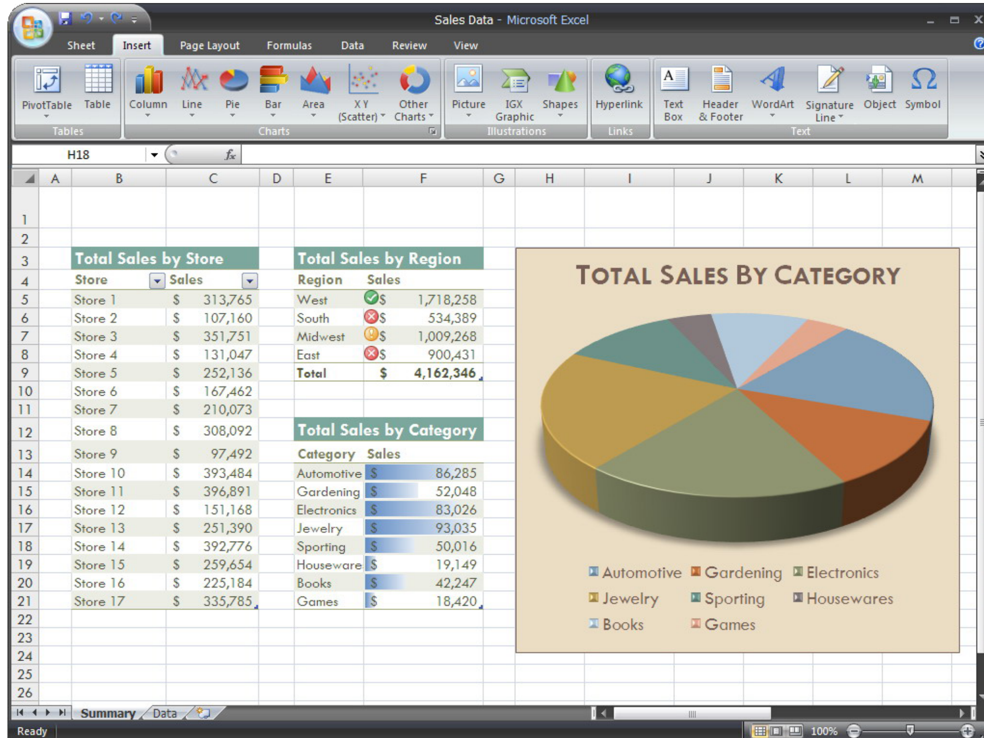


Figure 2.5: A picture of Microsoft Excel which contains a lot of functionalities.

Scripting Programming

Scripting 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[2], Lua[3] and Visual Basic for Application (VBA)[4]. Typically, these language 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. As mentioned earlier a macro recording in Microsoft Word results in a VBA script. Such a script can be altered to include loops and conditions.

John K. Ousterhout[Ous98] furthermore describe scripting languages as "glue languages". This is because they are also often used to glue together components that might

[1]This picture is found at `http://pubpages.unh.edu/~pit2/excel.htm`.
[2]Tcl website: `http://www.tcl.tk`
[3]Lua website: `http://www.lua.org`
[4]VBA website: `http://msdn.microsoft.com/en-us/isv/bb190538.aspx`

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.

2.4 Interactive and Programmatic Access

All of the end-user programming approaches described in the previous section interacted with the API of an application. We say, however, that there are two types of access to such an API: Interactive and programmatic. Interactive is actually calling the same methods in the application that the programmatic does, hence, the API.

Interactive Access

Interactive access to an application can be accomplished in many different ways. The mouse is the most common approach in todays computers, but the mouse can be used to different things. Many applications features menus, toolbars and dialogs as the default way of interaction where the mouse is used as the physical tool by moving and clicking. Some applications make use of mouse gestures. A mouse gesture is where moving the mouse and using its buttons in a special way have the same meaning as a command. The first mouse gesture were for the move-command introduced by Apple. It is what we today call *dragging* or *drag-and-drop*. Other programs such as the Opera Browser recognizes a special moving pattern for commands, e.g. down-and-right to close a tab. This type of interactive access have also been widely explored in games, such as in Black and White. A fairly new feature that have only recently been made acceptable is voice-control. More and more applications make it possible to do commands simply by speaking into a microphone, and again the Opera Browser is an example of such and application. This way of interaction is, however, still a bit flawed, but one might speculate whether this will be the preferred form for interactive access to applications in the future.

Another example of interaction without the traditional mouse is through motion detection using a camera. There exists a gesture recognizer called “Eye Toy” for PlayStation 2 from 2003 which is simply a camera plugged into the PlayStation and acts like a controller. The camera detects motions from the player and transform that into an action in the game. A disadvantage of this kind of interaction is that the user needs to be in front of the camera all the time, and some movements cannot be detected. A newer version have been released for PlayStation 3 and a similar, though more advanced, tool, currently known as “Project Natal”, have been announced for the XBox 360.

A newer way of interaction is touch screen and touch devices which gives opportunity to make new combinations of interactions e.g. multi-touch. This kind of interaction will likely be a standard, if not already, in the future for some type of equipment. However, when it comes to precision and text writing, mouse and keyboard are far better and faster.

What is common for all of these interaction approaches is that they all have the same purpose: To allow access to the applications API.

Programmatic Access

Access to an applications API can, however, also be done through a more programmatic approach, and such an access to applications also come in several different flavors. A simple one is the command-line, where users simply write a command in one line using some arguments and then invoke it. A good example of a type of applications that uses a command-line approach are computer games where the user might have access to a console to access more advanced options than what is normally available in an options pane. Such a console is specially helpful during the debugging phase for the developers of the game, but also for more advanced end-users who create their own levels or modifications to the game.

The next step is through scripting languages which is compiled and interpreted within the application and therefore are often sand-boxed. These scripting languages are often fairly simple and domain-specific.

Some applications allow a bit more advanced programmatic access through a plugin system. A plugin is often written in a more general purpose language and it uses the applications API for access. A popular application with this approach is the Firefox browser where many features are only available through third-party plugins. Plugins can sometimes be executed outside of the applications control, but it is always required to follow a protocol, such as having some specific set of methods in a main class.

If an application offers what you could call an open API it can be used by other applications, thereby avoiding the original application completely, except that it needs to follow the protocol for communication specified in the API.

2.5 Self-Disclosing

As children we all learned from how our parents or others did things and replicated it, and this concept is called **observational learning** or **learning by observation** (LBO). In the area of computer science this concept can describe how an end-user learns to use part of an application by using it and observing how it behaves. A more concrete example is when learning HTML by using a What-You-See-Is-What-You-Get (WYSIWYG) editor and see e.g. what the tag for making a link is by creating one using the tools.

DiGiano, Chris and Eisenberg[DE95] describes how a user learns the commands used in the AutoCAD application by first using the toolbars and then observing how the commands are expressed in the output field. Originally AutoCAD were created as a command line software, but eventually it adopted an interactive GUI where the same commands could be executed using toolbars and the mouse. AutoCAD has an output field where it displays information about these commands even when they are executed using the mouse, and it is this approach that through observational learning teaches the user about how the same commands can be written in the command line. Self-disclosing is the term for when you expose some information about yourself and in relation to end-user programming it is when you perform an action in an application and this application tells you more precisely what you have done. This is what AutoCAD in general terms does as it discloses the commands that the end-user is actually invoking.

DiGiano and Eisenberg[DE95] describes three properties which characterize self-disclosing programmable applications which enables LBO, where AutoCAD fulfills all. They are formalized as follows (italic text) and we have given each a headline (bold text):

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.

2. **Groups of disclosed expressions**: *The system will indicate groups of disclosed expressions connected with a single operation.*
   In AutoCAD you need to select two corners of the rectangle and these actions are group together in the command window by changing the start of the command line from "Command:" to a text describing the current parameter of the activated command. AutoCAD, however, do not display the actual argument value in the output field if the mouse have been used.

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.*
   E.g. writing the commands for creating a rectangle with the same input would create the exact same rectangle as done with the mouse.

These three properties will greatly help teaching end-users the different commands that might exists in an application. They work as in-context tutorials that teaches while the end-user is doing the actual work, thereby removing often time consuming and out-of-context tutorials such as the classic "Hello World". However, they do not show more advanced structures of a build-in scripting language, such as loops and conditional branching. For this it is required for the end-user to read a manual or documentation. Therefore we need to tune these properties. DiGiano and Eisenberg[DE95] do this by developing their own application and this resulted in six guidelines for the pedagogical use of self-disclosure, which are formalized as follows (italic text):

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).

During the development of our experiments we need to take the three properties and these six guidelines into consideration.

2.6 The Difference Between Genders

The amount of women in the programming industry is increasing, but is still very sparse. A good example of this is the amount of female students at the department of computer science at Aalborg University. Here only 5.1% of the 255 students are women[5]. Beckwith, Laura and Burnett[BB04] speculates that this trend might be because of ignorance in gender differences within computer science, which have made it a heavily man-dominated area. So to attract more female end-user programmers, we need to consider the differences in genders when creating our system.

There exists a research field called Gender HCI which is a subfield of Human-Computer Interaction (HCI). Research in this field emphasizes on the difference between how male and female interact with a computer. It started in 2004 at Oregon State University, but the fields project page now exists on the EUSES website[6].

The difference between male and female in regards to end-user programming can be split up into three main areas: Confidence, support and motivation.[BB04]

In the area of confidence, Beckwith, Laura and Burnett[BB04] look at self-confidence and the lack thereof, overconfidence and perceived risk. They found that females tend to have a higher lack of self-confidence when working with technologies than males do, even when it is not justified, meaning females have less self-efficacy. This also means that males often are overconfident which might result in more buggy programs. Whenever a person is about to do a task, this person weight the perceived costs, pay-offs, risks, and benefits of this task. Females have a tendency to put more focus on the costs and risks than males do. Because of this and the low self-efficacy, females are less likely to start using a new and unknown feature, even though it might be more efficient, as it might be "too risky".

The area of support is about how people learn, e.g. new features. Each person have a different learning style, but Beckwith, Laura and Burnett[BB04] did not find any specific differences between the two genders. They did, however, find a difference in the style of which males and females solve problems. Studies in this have mostly been done through computer games and here it showed that females like games where they can explore the world and problem freely and in collaboration with others while males tend to like games with linear problem solving and competitions. To solve a problem we all need information about it, and here there is also a difference between how male and female process this information. Males tend to like a small information load for simple tasks. In computer science females have higher tendency to read through long tooltips before actually using the feature.

[5]We have reached this number by manually counting the students for every study group listed on the computer science intranet. This includes international master students, but not first year students (basis year) as these only exists in a different system.

[6]Gender HCI project page: `http://eusesconsortium.org/gender/gender.php`

Finally there is a difference between how males and females get motivated to work with technologies. Females are generally motivated by how technology can help other people while males enjoy technology for its own sake. At the same time females tend to like systems that have a perceived ease of use while males like systems with perceived usefulness.

In many cases it is impossible to create a system which benefits both genders, e.g. males overconfidence and females high risk perception contradict each other. Some of the above mentioned statements are still only speculations in regards to computer science and end-user programming, therefore much research is still needed in the research field of Gender HCI.

2.7 Dangers of End-user Programming

In Chapter 1 we described how the amount of end-user programmers are very large compared to professional programmers. This also means that more and more programs are written by non-professionals who have not had any or only limited education in security, verifiability, reusability and efficiency. This is a problem as these programs have a much higher chance to contain errors which can cause damage.

Many end-user programmers program in spreadsheets applications such as Excel. Harrison [Har04] mentions a particually case from the 1980s where a contractor used the spreadsheet program Lotus 1-2-3 to prepare bids. This end-user had, however, made a mistake in the macros which resulted in a bid that was too low. He won the contract, but lost money on the project and thereafter tried to sue Lotus. The main reason for this error were the lack of testing, an important step in software development that many end-users do not do or simply do not know about.

Harrison [Har04] also mentions the increasing number of websites created by end-users who have just learned a bit of HTML and server-side scripting languages such as PHP and Perl. He speculates how many smaller e-businesses have failed because of lost orders or payments originated from bad code written by end-user programmers. In many cases these kind of end-users seldom know what they don't know, and they often have unrealistic ideas of their own actual ability[Har04].

A different kind of danger of having end-users program for an application is the possibility that they might exploit this opportunity. This is specially the case in games where it is possible to create addons that might help the end-user play the game. If this feature is not done probably the end-user might use it to simply cheat. A good example of a game with great addon possibilities are World of Warcraft. Using XML and LUA it is possible to create addons for the GUI which can be used to help the gamer and even improve the game experience. Blizzard, the creators of World of Warcraft, have, however, done a great many things to this system in an attempt to avoid cheating. At first it was possible to create a bot-like system that would simply play for you, but the system were changed so some actions needed to be started by a user action, such as a mouse click. These problems however usually only exists if the end-user have experience with programming.

The first mentioned problems with end-users about errors does also exists in a game such as World of Warcraft. It is possible for an inexperienced programmer to make errors in an addon which will eventually break the whole GUI system. Therefore, Blizzard were required to also handle these addons in a kind of sand-box mode where errors, e.g. missing

variables or null-pointer exceptions, can be caught and handled such that only the addon in question breaks.

2.8 Related Applications and Systems

In this section we will describe other applications or systems which is related to this project. Some use a command-like approach for drawing and these are AutoCAD and SVG. This knowledge will be used in experiment 2, Section 4.3, and 3, Section 4.4, when we create our own commands. Another drawing system that have the possibility to use scripting languages is Photoshop and knowledge about this will be used in experiment 4, Section 4.5, for creating our own scripting language. Finally we will describe Eager, an application for recognizing patterns in user actions and thereby provide a way to easily complete repetitive tasks.

AutoCAD

AutoCAD has been developed by Autodesk since 1982 and it was the first CAD software for the PC. CAD stands for Computer-Aided Design and is a term for designing objects (real or virtual) using a computer. Originally AutoCAD only used a command line, but later an interactive GUI was added. It also comes with its own dialect of Lisp (AutoLISP) which is a good example of a powerful, but complex end-user programming language. Drawing in AutoCAD can be done using either the GUI with the mouse, the command line or a combination of both, e.g. selecting the first point in a line with the mouse and writing the second using the command line. Several elements helps in this process like offsetting, cutting lines, or snapping to points, intersections, tangents etc.

Drawing in AutoCAD is done through a kind of dialog between the user and the application. To give an example of this we will go through how to draw a circle. Listing 2.2 shows what AutoCAD has disclosed for this dialog.

```
1 Command: c
2 CIRCLE Specify center point for circle or [3P/2P/Ttr (tan tan radius)]:
    50,50
3 Specify radius of circle or [Diameter]: d
4 Specify diameter of circle: 100
```

Listing 2.2: Example of the output field in AutoCAD 2000 after drawing a circle. Everything after the colons (:) on each line is user written.

First we tell AutoCAD what we want to do. Here we have written a "c", which is a shortcut for the `circle` command. AutoCAD then asks us to specify the center point of the circle or to select a different way of drawing the circle. These extra possibilities are:

2P Specifies the circle by picking 2 points on its diameter.

3P Specifies the circle by picking 3 points through which the circle will pass.

TTR Specifies the circle by picking two lines, arcs or circles for the circle to be tangent to, and entering the dimension of the radius.

We choose the default way by writing the center point, (50,50). We could also have selected this point with the mouse. Next AutoCAD wants to know the radius of the circle.

Again it gives us the possibility to do it in a different way, choosing the diameter, which we try by writing “d”. The final interaction here is then to specify the diameter which we set to 100. Now the circle will be drawn.

Appendix C contains a list of several AutoCAD commands and how this dialog is done for each of them. This knowledge will used for when we design our disclosure class in experiment 1, Section 4.2, and our own commands in experiment 2, Section 4.3.

SVG

SVG stands for Scalable Vector Graphic and is an XML file format for describing 2D vector graphics. It is an open standard that have been developed by World Wide Web Consortium (W3C) since 1999. It is possible to create animated vector graphics in SVG.

SVG uses XML tags to describe shapes of different kinds. These commands have a keyword and several attributes (or arguments). As an example we will here describe the tag for drawing a circle:

```
<circle cx="600" cy="200" r="100" />
```

As in AutoCAD a circle takes a center point, which is here done with the attributes `cx` and `cy`, but unless AutoCAD there is no other way to specify a circle in SVG. We cannot specify the diameter of the circle either and instead we need to specify the radius, which is here done with the attribute `r`. In keep with the keyword argument approach, see Section 4.4.1, it is only required to specify the radius as the default center point is (0,0).

Appendix D contains a list of SVG commands related to the commands we need to create for the DrawTools application in experiment 2, Section 4.3. We will also use this knowledge for the keyword arguments in experiment 3, Section 4.4.

Adobe Photoshop

Adobe Photoshop, hereafter referred to as just Photoshop, was original created by Thomas Knoll, a PhD student at the University of Michigan, in 1987 and was named Display. Together with his brother, John Knoll, he expanded the application and renamed it to Photoshop. Adobe bought the license for the application in in 1988 and Photoshop 1.0 was released in 1990 on Mac exclusively. It is now being released on PC as well and have grown into “an industry standard for graphics professionals”[7].[Sto09]

Photoshop offers the use of three scripting languages: AppleScript, JavaScript and VBScript. All have an extended standard library which adds methods and objects that can be used in the Photoshop environment. These scripts can be called manually or through application events such as when a new document is created. All three scripting languages are complex in nature and not possible to use for an end-user with no experience in programming.

Listing 2.3 is an example of a small script for Photoshop written in JavaScript. It rotates the current selected layer 45 degrees. It is worth noticing here that it needs to have two checks to avoid errors: Is there any documents open and is the current layer the background layer. The variable `app` is a reference to the Photoshop application itself. The first two lines makes it possible to run the script by double clicking on the script file in Windows Explorer or Macintosh Finder.

```
1 #target photoshop
```

[7] http://money.cnn.com/2007/03/01/technology/adobe/.

```
2  app.bringToFront();
3  if (app.documents.length > 0)
4  {
5      if (app.activeDocument.activeLayer.isBackgroundLayer == false)
6      {
7          docRef = app.activeDocument;
8          layerRef = docRef.layers[0];
9          layerRef.rotate(45.0);
10     }
11     else
12     {
13         alert("Operation cannot be performed on background layer");
14     }
15 }
16 else
17 {
18     alert("You must have at least one open document to run this
           script!");
19 }
```

Listing 2.3: Script to rotate a layer in Photoshop 45 degrees written in JavaScript.

In Photoshop all drawings are divided into several layers. It is possible to add “blending options” to each of the layers, and these options include adding a shadow which is one of the tasks mentioned in Section 3.1.1. It is also possible to select objects and easily move, scale or copy them. For drawing a gradient there is a special tool available that have preferences to make all kinds of gradients. There is, however, no command-like interface or a script editor native in Photoshop.

Eager

Eager is an old tool written in LISP which uses the programming by example approach by monitoring user activities in an HyperCard environment. Every action is monitored and when Eager detects an repetitive pattern in the user actions, it will start predicting the next action of the user. If the actions matches what Eager has predicted, it will create a program which will be executed and complete the task automatically. The goal of Eager is to help users to solve repetitive tasks without doing textual programming. An example of how Eager works, taken from the chapter about Eager in Watch What I Do by Alan Cypher[Cyp93a], is shown in Figure 2.6.

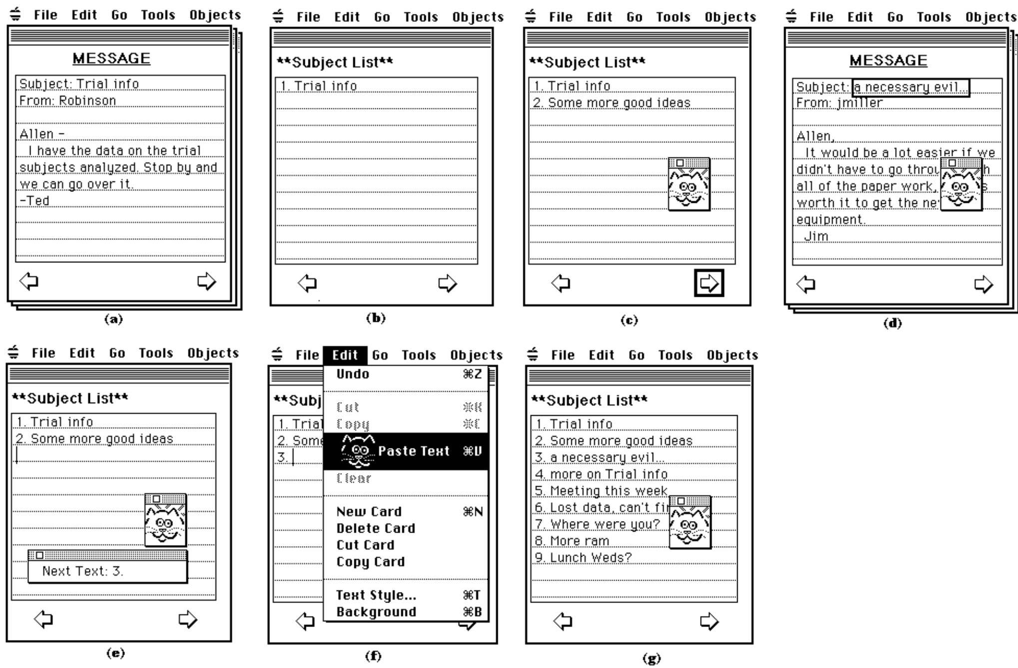


Figure 2.6: As an example, a user has a stack of messages each with different subjects (a), and the goal is to create a list with all subjects (b). When the user copies the subject from a message and paste it into the new list the second time (c), Eager will detect this repetitive pattern and start it's 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.

2.9 Summary

In this chapter, we have introduced many aspect in the end-user programming area. We introduced the Grand Canyon Gab which describes the difficulties between the way a human brain represents a problem, and the way a computer understands and accepts a problem. There are two ways to bridge this gab, either we move the end-users closer to the system, or the system closer to the end-users.

We have also looked at user actions which give a basic knowledge of what happens when we interact with a computer. Furthermore, we described different programming approaches for an end-users, starting with preferences programming which is not considered as traditional programming. For PBD there exists many tools, Stagecast Creator, ToonTalk and Scrath are some examples, and they are all aimed to learn children to program by demonstrating rules. Unlike PBD, spreadsheet programming is in essence a first-order functional programming, and let end-users create formulas and business models. Finally, scripting programming is the most difficult one compared to the other end-user programming approaches, and it is also the closest programming approach to a system programming.

Although, there are many programming approaches, we have seen that there are in general two ways to communicate with an application, interactive and programmatic.

By human nature, we are good to learn and replicate things which basically is LBO. AutoCAD is using this idea to teach an end-user the internal command system which can be used in AutoCAD's command line.

Even though humans are good to learn, male and female have their own way to learn things. This knowledge is useful when we create a tool which should be used by both genders. We have, however, also found that it is very difficult to prepare an application for both genders as some of their differences are contradicting.

We have seen that end-user programmers lack knowledge of software engineering and because of that, they tend to make software errors which can cause huge damage. A sandbox mode can be boxed around the part where end-user programmers execute their programs and thereby isolate the damage caused by errors.

Finally, we introduced AutoCAD, SVG, Adobe Photoshop and Eager which are all inspirations to our experiments.

Chapter 3

Problem Statement

From the analysis, we have looked into a wide area of end-user programming and found a problem which we find interesting. In this chapter, we will describe the problem and how we will solve it. We start with a motivation to clarify the problem, and it follows by a vision of our idea to solve the problem. Clearly, we need to put our focus somewhere, and this is described in our delimition section. Finally, we have one main question and some more specific questions for this project periode. These questions will help us keep focused and on the right track.

3.1 Motivation

From the analysis we have found that end-users have difficulties with learning a new language, regarding to the two different way of representing a problem (Section 2.1). However, there are huge number of end-user programmers according to the EUSES consortium, but even more end-users which clearly require some attentions, and the number is still increasing. This means that we need to put our focus on how to help these end-users start programming or learn them to think in a Fregean way, either by tools or techniques. At the writing moment, there exists many tools which are based on PBD (e.g. Stagecast Creator, ToonTalk, Scratch). Although, they have different difficulties and are aimed for children at different ages, there is still a gap from this kind of rule-based style to a semantic programming style.

If we look at adult end-users, they have many different professions, but none of them has programming as their primary job. Typically, they want to program a piece of code immediately to solve a tedious repetitive task for their everyday job. Using an application like Stagecast Creator to start learning to program takes time and does not give programming skill immediately, hence, we have two different groups of end-users with different goals; to create a larger piece of software, or to solve a specific task immediately and fast by programming. Stagecast Creator fits the first group, but the second group needs another kind of tool.

In this project, we will create a tool which uses LBO, inspired by AutoCAD, and PBD, inspired by Eager. AutoCAD discloses commands from the user interactions while Eager detects repetitive tasks and automatically generates a script for that task. Our tool will, therefore, disclose user interactions, not only commands, but also disclose repetitive tasks in a scripting language.

The point of the disclosing is to let end-users learn to program by observing their interactions in a textual form while using the software application. The textual form should give the end-users the idea of how the same interaction can be done by writing a command or combine the commands into a large serial of interactions. This approach will therefore let end-users learn to program within their already known application environments, hence, motivates them to program. PBD is used indirectly, because our tool monitors interactions from the end-users. In other words, end-users are programming by demonstrating while interacting with their application. Our goal is, therefore, to create such a tool which uses both LBO and PBD approaches.

3.1.1 Examples of Repetitive Tasks

We have here created a list of concrete examples of repetitive tasks which might exists in a simple drawing application. A motivation is to overcome these tasks by programming:

- Drawing the same shape many times.
  E.g. a 360 lines in a circle or many rectangles inside each other.
- Drawing a gradient.
  Fading from one color to another color.
- Drawing the same thing several times with different colors.
  E.g. switching between red and green colored lines.
- Drawing a specific figure several times.
  E.g. a star, working with a script that can draw this figure would help.
- Scale everything or part of a drawing.
  E.g. make it smaller so it fits inside something else.
- Drawing a chessboard.
  Or grid.
- Adding a shadow.
  Or drawing the exact same thing in a different color just moved a bit.
- Drawing very precise.
  E.g. drawing tangents to a circle in a non-orthogonal way (not on the default axis).
- Random locations of several objects.
  E.g. drawing a night sky with stars at random locations in random sizes.

Many larger drawing applications, such as Adobe Photoshop, includes functions that deals with some or all of these problems, but more simple and easy to use applications, such as Microsoft Paint, do not.

3.2 Vision

Our vision is to create a tool which discloses all end-users interactions, or events, for the whole computer—all running applications and the operating system. It is inspired by the application Eager, see Section 2.8. An abstraction of the events will be made and related events will be grouped together as user actions. These end-user interactions can be used and edited as expressions in a scripting language created for the purpose. The end-user interactions will be disclosed while the end-user is doing their interactively tasks, thereby, teaching the scripting language by using the LBO approach. The scripting language will include loop structures, conditional branching, abstractions and variables. An advanced feature in this tool is detection of repetitive tasks. Our tool should monitor the end-user interactions and if a repetitive task is found, the tool should automatically create a script which includes loops and abstraction such that the repetitive task can be automated, and at the same time give the end-user an idea of how such a script should be written. At the end this tool should help end-users work faster in any application and in any combination of applications and last, but not least, learn to program.

3.3 Delimitation

In this project, we have chosen to minimize the global user interactivity with the computer to a single software application, the drawing software application DrawTools. This application has been chosen, because it is open source and is well-written in C#. In addition, the DrawTools is very simple both in user interface and the tools for drawing. The simplicity of DrawTools lets us focus more on the core issue of this project—that is, disclose user interactions. A drawing software program has been chosen, because of the extensive mouse activities in such a program compared to e.g. a text editor.

To disclose the user interaction with the drawing application, we will design a small scripting language. The scripting language should be expressive enough to cover the very simple tools available in DrawTools. The main purpose with the script language is to disclose user interaction with understandable commands for further use—in this case, let the user learn to program by observing the disclosing.

In addition to the scripting language, we will implement conditional branching in the form of if-condition and support loop-structures in order to give the user the opportunity to easily create repeatedly commands.

3.4 Questions

For this project, we have one major question to answer. This question is a problem which we from the analysis have been able to narrow down to:

> *"How do we motivate end-users to program?"*

To help answer this question in a more specific area, we have created further questions which we would like to answer during this project periode. Refer to the Chapter 5 for a discussion on this.

How do we create a command line and scripting environment for a drawing application?
We need to figure out which commands that should be available and what arguments they need and if they should have more than one set of arguments. Here we also need to fulfill the six guidelines for the pedagogical use of self-disclosure described in Section 2.5. In addition, this question is to help us understand how we can create such an enviroment for an existing application.

How do we disclose the end-user actions in a meaningful and easy way?
As part of the LBO approach, we need to disclose the end-user actions in a meaningful way which at the same time works as an easy accessible history of actions. Therefore we need to figure out how this can be done in an simple way which at the same time does not require to many changes to the original application.

How should the command line and scripting language be designed?
This language needs to have the possibilities of loop-structures and conditional branching together with a command-like approach. We also need to consider the fact that the language needs to be easy to use and understand and at the same time describe the end-user activities—meaning fulfill the three properties of self-disclosure described in Section 2.5.

Chapter 4

Experiments

To answer the questions stated in Section 3.4, we have done a number of experiments involving the design and implementation of various parts of an application that involves PBD and LBO. We will describe each of these experiments in this chapter, in chronological order, starting with the core idea of disclosing end-user interactions. Each experiment is structured with a purpose, design and implementation, usage, and result. The first section of this chapter, however, is about some preliminary work involving Windows hooks, which turned out to be too complex to start with. This led to the use of a simple application as the base for the experiments. The first part will also include an explanation of applications and systems that is related to our experiments. At the end of this chapter we will discuss what we have learned from the experiments and what we can use for future projects.

4.1 Preliminary Work

In this section we will describe some work we did before we decided to do experiments.

In Windows it is possible to capture a users low-level and mid-level events sent to any application by using what is known as hooks[1]. A hook works like an observer pattern. One application hooks itself to an event and gets told about this event before the original receiver does, that way it is possible to listen and interpret events and even stop them. One kind of hook listens to events after the original receiver have handled it, so it is possible to know what its response was.

For our project we would like to listen to all events and keep a kind of statistics database of the most recent. This will allow us to indirectly create statements through PBD, but also to look for repetition.

It is possible to use all kinds of hooks in C#, but only for the application itself. The only global hooks are those for mouse and keyboard which is the low-level events. This makes it possible to create a macro recorder, but not an application that uses the PBD paradigm. The reason for this limitation is the way the .NET framework works.

So to create a PBD application that listens to all kinds of events for any application, we cannot use C# or any other programming language which uses .NET. C++ is a here a choice instead. While we worked with the Windows API in C++ we realized that we need to do much more than just collecting events using hooks. Because of this we have decided

[1] About hooks on MSDN: `http://msdn.microsoft.com/en-us/library/ms632589(VS.85).aspx`

to do several experiments in a simpler environment that will bring us closer to our goal, one step at the time, starting by only looking at events for one application. We hope that through this approach we will at the end have everything ready and make the switch to using hooks.

4.2 Experiment 1: Disclosing Commands

In this experiment, we will disclose end-user actions in the drawing application. The purpose is to examine and understand how to disclose end-user interactions in an existing software application, in this case a drawing tool. In addition, this experiment also gives us a deeper understanding of the method calls in the application which is needed for the next experiment about commands. Notice that we will only strive to implement the most necessary code at this point in order to gain as much progress as possible.

4.2.1 Design

It is a good idea to keep all disclosing of a shape close to the shape itself. Code-wise that means having the disclosing code in the draw-classes. All of these draw-classes inherit from the abstract class `DrawObject`, which is shown in Figure 4.1. This means that we can create a class for disclosing interactions and add it to the `DrawObject`, thus, all draw-classes will contain our disclosure class. By doing this, we can, whenever e.g. a draw method is called, call a method to disclose the action from our disclosure class to send a message to a text box. Furthermore, it will ease future implementations or changes to the disclosure class, since it will only be done at one place.

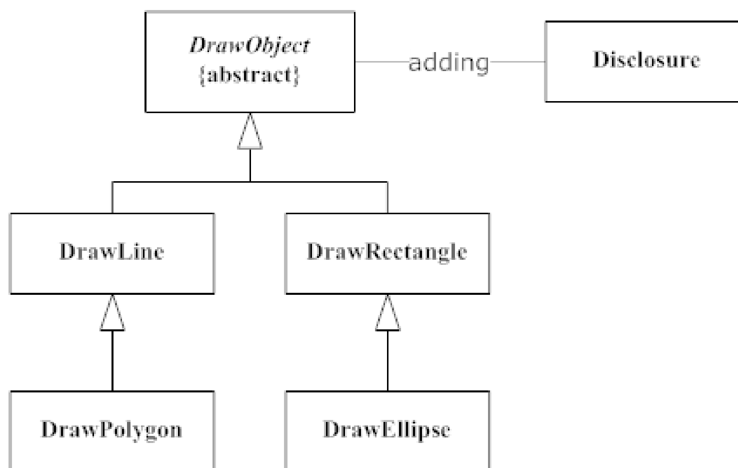


Figure 4.1: A class diagram for DrawTools with our *Disclosure* class added to the *DrawObject* class.

Each draw-object looks alike and the creation and manipulation of these can be expressed very similarly. To increase readability, however, we will make the disclosure class contain several styles also.

The first important methods are `SetStartPoint()` and `SetEndPoint()`. These are used to specify the start and end points of a shape. For a line that is each end, and for a rect-

angle it is two opposite corners. These will be used to calculate the width and height of the shape. They will also be used for the actual disclosure done with the `Output()` method when a shape is created. When a shape is moved it will instead call the `OutputMove()` method that will express a distance and direction based on the start and end points. Finally the method `OutputSize()` is called when a shape is resized. In both cases the actual moving and sizing is not an instant process. Instead it starts when the mouse is pressed and ends when the mouse is released again, but is updated every time the mouse is moved. Therefore we have added the methods `RegMove()` and `RegSize()` which is called whenever the mouse is moved such that the whole movement or resizing can be disclosed.

4.2.2 Usage

In order to test the result of our implementation of a disclosure class into the draw application, we simply interact with the application. As we interact with the application, we expect to see some predefined messages printed out in our text box. As mentioned, we only implemented the most necessary code, therefore, we expect only to see messages from interactions with some tools. Figure 4.2 shows a screenshot of the new DrawTools application where a few shapes have been drawn.

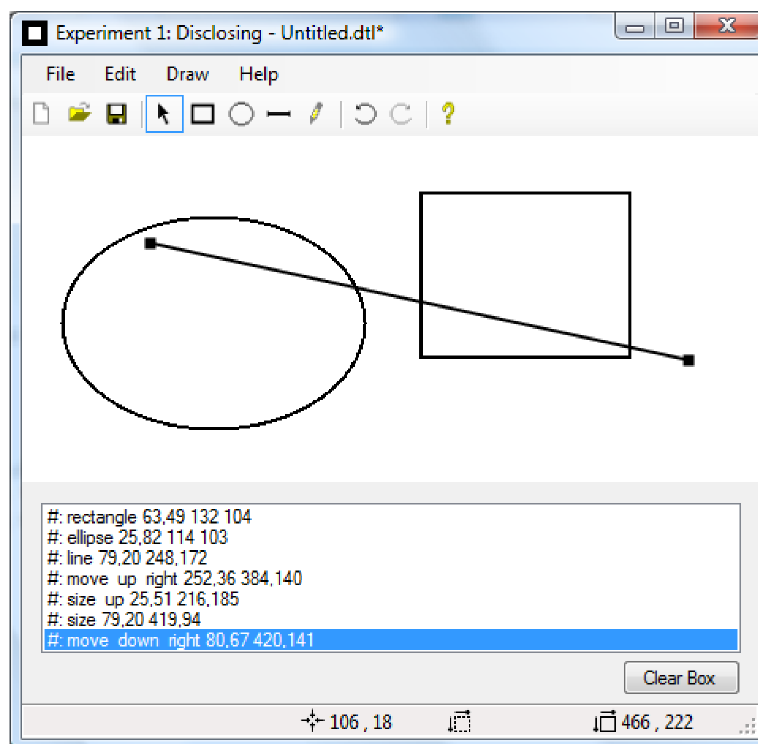


Figure 4.2: DrawTools after experiment 1, where a few shapes have been drawn. Notice the disclosing of the shapes creation.

4.2.3 Results

We succeeded in creating a disclosure class which can be called whenever it is needed to disclose an interaction with the application. The next step is then to add a command line where it is possible to write the same that is disclosed to get the same result, and this class

is made such that it is easy to change the disclosing so it fits the commands we will create in the next experiment.

The command line will first be implemented in the next experiment, therefore our experiment application only fulfills a few of the properties and guidelines for self-disclosing described in Section 2.5. The disclosure and command line in AutoCAD works as a dialog between the end-user and the application. In our application there is only one action for every disclosure, e.g. drawing a rectangle is only disclosed when the mouse button has been released and the whole drawing is done. Therefore there is no grouping of disclosures, but there is a grouping of the arguments for a command.

4.3 Experiment 2: Sloppy Command Line

With the knowledge of method calls in the source code, we will in this experiment make a small command line interaction with the application. The goal is to let end-users be able to type in the commands instead of using the mouse. Furthermore, we will adjust the disclosing from the previous experiment to match the commands designed in this experiment. This experiment will also set the base for a later scripting language.

In addition, we will experiment with sloppiness in the command line. Sloppiness is described by Miller, Chou, Bernstein, Little, Van Kleek, Karger, and schraefel[MCB+08] and in this context sloppy means that both the arguments and the command keyword itself can come in any order without the use of argument keywords. We believe this will make it easier for end-users to learn to use the command line, since they do not need to remember all the arguments for a command and their order. In other words, we think it is more natural for end-users if they can decide the order of arguments and which to use by themselves.

4.3.1 Design

To create a command line we need a parser and an interpreter. The interpreter will be the same no matter how the command line will work, it just needs a command and some arguments. In this experiment we have decided that the command line should be sloppy. We have chosen that the whole string written into the command line will be divided at each space. Every part of the string can be the command, and every part can be any argument to this command. Therefore we need to analyze what each part could be and how these are most likely to be connected. Miller et al.[MCB+08] takes the approach of defining a set of types (of the arguments) and a set of commands. Each command will then have a list of types for each possible parameter set. In the following sections we will first describe the available types, then the commands and their relations to these types, and finally the parsing algorithm itself.

Types

All types can be recognized either through a pattern or a simple textual equality, or both. These types are not all distinguishable, e.g. several of them consists of only an integer value. A good example is the type for the pen width which is either an integer or a textual string like “thin” or “thick”. A complete list of types can be found in Appendix E.

Commands

The commands will be based on the available tools in DrawTools. Each command needs one or more unique keywords such that it can be distinguished from other commands. These keywords should “make sense” to an end-user and should also include the most obvious choices and shortcuts for when the end-user have learned the command. If a command have more than one possible parameter set it needs to use types that can be distinguished. One note here is that several types can consist of a simple integer, therefore these types are not distinguishable.

We can categorize these commands into three groups: Drawing shapes, manipulating shapes and application handling. In the following we will give an example of a command from each of these groups. The complete list of commands can be found in Appendix F.

Commands for Drawing Shapes
These commands are used for adding new shapes to the drawing. All of these commands should have the option of specifying the color and width of the border. If unspecified it should use the currently selected color and width. Therefore all parameter sets will also include a `Color` and a `PenWidth` type which in all cases are optional.

These commands are based on the equivalent commands from AutoCAD, see Appendix C, and SVG, see Appendix D. We have also looked at the mathematically definitions to the shapes.

As an example of a drawing shape command we will look at the rectangle command. The specification of a rectangle is that it has four sides that are parallel in pairs, and all corners have an angle of 90°. The parallel sides also have the same length. So for drawing a rectangle we only need to specify its position and its width and height. A common approach to this, which both AutoCAD and SVG uses, is to specify the upper left corner (closest to (0,0)) and either the width and height or the lower right corner (furthest away from (0,0)). Therefore the rectangle command should have two parameter sets.

Table 4.1: The two parameter sets for the rectangle command.

	Parameter	**Type**
Set 1	Upper left corner	Position
	Lower right corner	Position
Set 2	Position	Position
	Width	Integer
	Height	Integer

These two parameter sets are distinguishable both by the count of parameters and the fact that the first have two `Position` types while the second have only one and two `Integer` types instead. In DrawTools there is no requirement that the first corner is the upper left, therefore there is no need to make constrains about the order of the `Position` parameters. The order of width and height, however, do matter and we have decided that width is the first, as this approach seems the most common.

The most obvious choice for a keyword of the rectangle command is `rectangle` and `drawrectangle`. However, it can also include several shortcuts: `r`, `rec` and `rect`. The use of `r` for the rectangle is because it is the most used command that starts with an “r”.

Commands for Manipulating Shapes
These commands is used to manipulated already created shapes in the draw area. All of these requires that one or more shapes have been selected.

These commands are `move`, `resize` and `delete` where delete is the simplest of these commands as it does not take any arguments. The effects of these commands should be obvious.

Commands for Application Handling
These commands include different menu options available in DrawTools. A good example of this is a command for closing the application. Such a command is available in almost any application that uses a command line approach. It will use the close method already

available in DrawTools which means it will first ask whether or not you want to save the current drawing if it has been changed. This is an example of a command that does not take any arguments.

The exit command have often come in several forms in applications that have used a command-like approach, e.g. a Unix shell or games with a console. Therefore we also need to have several keywords for the exit command. These are: `exit`, `quit`, `close` and `q`.

Parsing Algorithm

Even though Miller et al.[MCB+08] describes their sloppy command line they never actually mention how it is designed and implemented. Therefore we have to do this our self. In this section we will only make a brief description of it, while the complete design and implementation can be found in Appendix G.

The algorithm used to parse our sloppy command line can be divided into two parts: A lexical analysis and a syntactical analysis.

Lexical Analysis

As described earlier we have decided that the string from the command line will consists of several parts divided by a space. Therefore the first part of the lexical analysis is simply to split the string. Hereafter it will go through each part and first find possible command keywords and thereafter possible types. Listing 4.1 shows how the syntax is expressed in BNF.

```
1 <command>  ::= <keyword>
2               | <part> <parts>
3 <parts>    ::= " " <part> <parts>
4               | ϵ
5 <part>     ::= <keyword>
6               | <argument>
7 <keyword>  ::= (string with only letters and digits)
8 <argument> ::= (string without any space character)
```

Listing 4.1: The syntax for the sloppy command line expressed in BNF. In this syntax there cannot be any whitespace on either side of the <key>'s. There is one constrain that says that at least one <keyword> needs to be present in the <command>.

To find the possible command keywords and possible types the algorithm will simply search through the libraries of commands and types and ask each of them if they could fit at any of the <part>'s. There is no requirement for the order of command keywords and arguments, but at least one of the <part>'s must be a command keyword, else it would not make sense.

Syntactical Analysis

After the lexical analysis we do a syntactical analysis. This part of the algorithm is quite complex as it contains loops nested in six levels. A performance analysis of this, however, shows that it is not that expensive as several of the loops will only be run a few times, some even only once. A testing even showed that with our current amount of commands and types it is still faster than the lower bound for timing set by either the operating system (Windows 7 in our case) or the .NET framework (we believe this bound to be around 2.5 milliseconds). Such speed makes it possible to actually run the parser for every keystroke thereby make it possible to give the end-user realtime feedback.

The syntactical analysis first finds every possible command that the lexical analysis found. It then checks all parameter sets for each of these commands. Each parameter set have several parameters that is then compared to each of the possible arguments found in the lexical analysis. Whenever the algorithm is done with one parameter set it will calculate a score based on how many arguments that fitted the parameter set, how many that did not fit, and finally how many that are still missing. At the end of each command these scores are compared and the best possible parameter set is selected. This ends with the calculation of a score for each command and finally the best command is selected and returned.

Implementation
The parser and the interpreter is implemented as two classes. Each currently have one method that parse and interpret a command line respectively. Both classes uses the singleton pattern to make them easier accessible, and specially for the interpreter this is great help in later experiments. The interpreter's only function at this point is to call and `Action()` method on the command, and this lets each command handle their own interpretation.

4.3.2 Usage

Now that we have created the sloppy command line we want to test it. We have two main concerns: Does it work, and is it understandable? We will do the test our self as the application is not ready for "real" end-user programmers. Figure 4.3 is a screenshot of the application after we have run several commands which all resulted in a shape to be drawn and a command to be disclosed.

The commands we ran where:

```
1 rectangle 20,20 100 150
2 c 100,100 75
3 75,75 ellipse 300,200 #0f0
4 300,275 red thin 75,225 line
5 52 0,0,255 32 l 2 43,37
```

We intentionally used shorter synonyms and wrote the arguments in a random-like order. All of them gave the result we expected, but it is worth noticing that several of them are not particularly readable. Specially the last command, Line 5, can be very confusing. This expression actually creates a line with length 52 and an angle of 32 degrees starting from the position (43,37) using a blue color and with a width of 2 pixels.

4.3.3 Results

The result of this experiment was as intended, our command line worked. However, after a discussion about the sloppiness of the command line, we find that this programming style is still too difficult and lack of readability for an end-user. As seen in Section 4.3.2 the example of a command is not well self-describing; it is difficult to see what each arguments are used for. Also, we find it difficult to create such sloppiness command line, because we need to check each arguments to see which argument is e.g. start point of a draw object, and the sloppiness restricts the flexibility of the commands. Therefore, we will in the next experiment improve the sloppiness command line with keyword arguments which will ease the implementation and improve the readability.

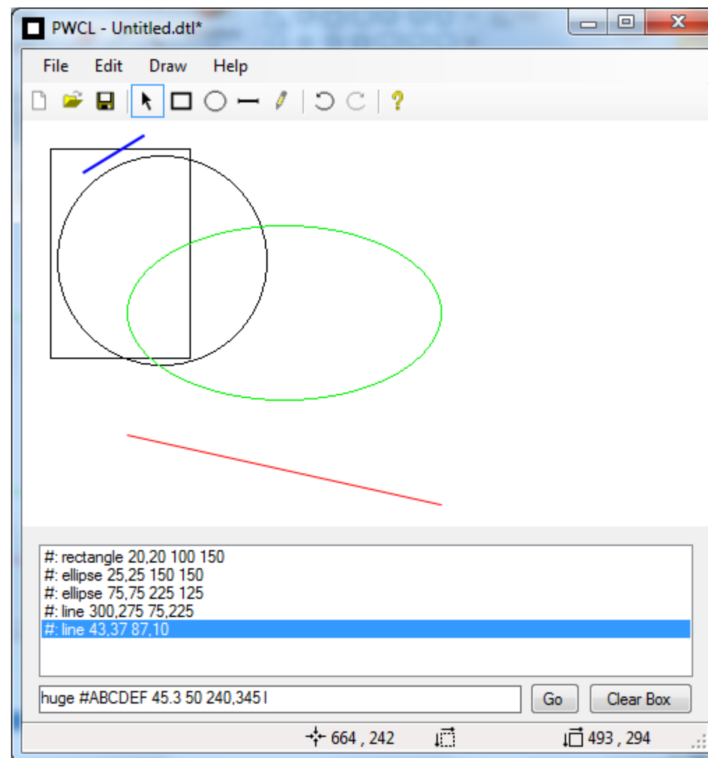


Figure 4.3: Screenshot of the application after experiment 2. Five commands have been written in the command line which all resulted in the drawing of some shapes in the drawing area.

Now that we have added the command line we have successfully fulfilled more properties and guidelines for self-disclosing, Section 2.5. Now all properties are fulfilled including "identical result from programming" which states that if the user had entered the same as was disclosed by using the mouse the result would have been the same. Because of the already existing undo/redo functionality in DrawTools we have also fulfilled the "experimentation" guideline about the possibility of experimenting with the commands. Furthermore it is possible to copy a disclosed command to the command line to easier change the arguments. As mentioned in Section 4.2.3 any command equals one action, which means that even though it is possible to use either the mouse or the command line to draw shapes, it is not possible to use them together, therefore the "combination of access" guideline is only partial fulfilled.

4.4 Experiment 3: Keyword Arguments

In the previous experiment, we found it difficult to implement sloppiness regarding to check what each arguments means and how it should be used as arguments to method calls. We also found that sloppiness is not an easy solution for end-users. In this experiment, we will improve the structure of arguments by using keyword arguments like in any XML-based file format, e.g. SVG, and programming languages such as Common Lisp. This will ease the implementation and improve the readability.

First we will give a discussion about different argument approaches and why we ended up choosing keyword arguments. Thereafter we will describe the design and implementation of the parser algorithm and other parts of the application that needed to be changed to use the new argument approach.

4.4.1 Argument Approaches

When we realize that sloppy arguments is not the best choice for end-users we need to look for alternatives. In this section we discuss different kinds of argument approaches and finally pick one we will use instead.

The by far most commonly used way for expressing arguments of a procedure is *ordered arguments*. However, there exists other approaches such as *keyword arguments* and *variable-length arguments*. Here we will describe these three approaches. For a description of the sloppy argument approach see Section 4.3.

Ordered Arguments

Ordered arguments are the most widely used approach and it means that each argument needs to come in the order specified by the method description, the formal parameters. In some languages, such as C++, it is possible to have default values for these parameters which give the programmer the opportunity to leave out some of the arguments. In these cases it is often only possible to have default values and leave out arguments from the end of the list and not in a random fashion. Some languages also allow several methods with the same name where the count or type of the arguments then decides which method will actually be called. This means that it is possible to have several methods with the same name, but different signatures. In a language such as C# this approach is used instead of default values.

Ordered Arguments is not easy to use for end-users as it requires the order of arguments to be exact. At the same time the arguments do not have great readability as it is not possible to see what the value is used for just by looking at the code. Most modern editors, however, fixes this problem by giving a good feedback on the method that the user is currently writing.

Keyword Arguments

In languages where the keyword argument, also known as *named parameters*, is used the order or count of arguments does not have any significance. Instead all arguments are prefixed with the name of the argument. This makes it possible to write the arguments in any order. An advantage of this approach is that it is easy to see what is going on at the method call without knowing the method definition, thus, increasing the readability.

A disadvantage of this approach is if the name of the parameter changes, then all calls to this method needs to be changed as well even though the order did not change. The same problem exists with ordered arguments if the order of parameters change, but this happens less frequently. Most modern editors, however, comes with a refactoring tool that will make such an action easy to do.

Variable-length Arguments

Variable-length Arguments is a term for when a procedure takes a variable amount of arguments. Such a procedure is also known as a *variadic function*. One of the most common uses of this is the `printf()` method in C. It takes in one string and then a variable amount of other variables, which can even be of different types. In most languages the same effect can be achieved by passing an array or list of values instead, though these will often have to be of the same type. Variable-length arguments is a handy feature in some cases, but is in general not easy to understand for end-users as there is not any obvious documentation of what arguments might be needed.

Choice of Argument Approach

From this discussion on different approaches to arguments, we have decided to use the keyword arguments instead of our current sloppy arguments. We feel this is the best choice for end-users and at the same time not difficult to design and implement.

4.4.2 Design

To make the change from sloppy arguments to keyword arguments we will first define the new syntax. There exists several programming languages that uses keyword arguments, and also several different ways to use it. The next version of C#[2] will have keyword arguments and it will use a colon (`:`) placed between the keyword and the value. Ada uses an arrow (=>) to separate keyword and value and then separates each argument with a comma (`,`)[3]. Another language that uses a comma to separate the arguments is Python, but instead of an arrow Python just uses an equal sign (=) between keyword and value[4]. All XML-bases formats[5], such as SVG, also uses the equal sign as the separator, but to separate the arguments there is only a whitespace, or the value is surrounded by double or single-quotes. We feel that the way it is done in XML is the most intuitive for end-users. We define that any keyword will only be a single word, meaning that they will not contain any whitespace. It is the same in almost any other language which features keyword arguments, but we use this fact to define the separator between each argument. The final word before an equal sign will always be a keyword. Therefore everything after an equal sign, except the last word before the next, is the value. The final syntax can be seen in Listing 4.2.

[2] C# 4.0: `http://code.msdn.microsoft.com/Project/Download/FileDownload.aspx?ProjectName=csharpfuture&DownloadId=3550`

[3] Ada: `http://www.adaic.org/standards/05rm/html/RM-6-4.html`

[4] Python: `http://docs.python.org/3.1/reference/expressions.html#calls`

[5] XML: `http://www.w3.org/TR/REC-xml/#sec-starttags`

```
1 <command>       ::= <commandname> <argumentlist>
2 <commandname>   ::= (string without any whitespace)
3 <argumentlist> ::= <keyword> "=" <value> <argumentlist>
4                   | ϵ
5 <keyword>       ::= (string without any whitespace or equal characters)
6 <value>         ::= (string without any equal characters)
```

Listing 4.2: The syntax for the keyword argument command line written in BNF. There can be an unspecified amount of whitespaces on both sides of any <key>'s.

This syntax of course requires us to change the parsing algorithm. In fact we can create an even simpler parser than with the sloppy arguments. This new algorithm have one loop for the lexical analysis and only loops nested in three levels for the syntactical analysis. To make this new algorithm work with the old types and command classes we only need to change the parameters for each command such that they include a list of keyword synonyms and a default value. We want several keyword synonyms for each parameter to increase the usability. Here we need to make sure that all parameters have distinguishable synonyms, just like the commands. However, this is only required within each parameter set. Some parameters exists in many parameter sets and commands, e.g. color and penwidth, so to help the end-user these should always have the same synonyms in any parameter set.

Listing 4.3 shows how the parameter for the start position of a line is defined. In this case the list of synonyms are quite long which increases the chance of end-users using a correct one from the beginning.

```
1 TypeObject tempParameter = new TypePosition();
2 tempParameter.AddSynonyms(new string[] {
3   "startposition", "start", "startpos", "startpoint", "s", "sp", "from"
4   });
5 tempParameter.SetDefaultValue(new Point(0,0));
```

Listing 4.3: Definition of the start position parameter for the line command.

These parameter keywords and default values are pretty straight forward to design and implement and we will not describe this further here. A complete list of parameter keywords for each command can be found in the the command list in Appendix F.

Finally our `Disclosure` class needs to be updated so it uses the new keyword syntax for the outputs.

Expressions

As a preparation for the scripting language in the next experiment we wanted to expand the current command line to allow arithmetical expressions. This is to allow the values of the arguments that uses integers or doubles to be expressed using calculations like addition and multiplication. Listing 4.4 shows the syntax for expressions. This syntax includes the use of variables which, however, will only be possible to use in the next experiment. The `<value>` from Listing 4.2 cannot always be translated directly to the `<expression>` in Listing 4.4. The reason is that the value for e.g. a position will instead contain two `<expression>`'s divided by a comma (`,`), such that each coordinate can be calculated. This allows us to have different domain specific types within the language and still give the end-user the possibility to calculate the values, e.g. both the x and y in a position.

```
1  <expression> ::= <expression> <operator> <expression>
2                 | <random> <integer>
3                 | <random> <double>
4                 | <random> <variable>
5  <operator>   ::= "+" | "-" | "*" | "/"
6  <variable>   ::= string with letters and digits
7  <integer>    ::= string with only digits
8  <double>     ::= string with digits and one dot
9  <random>     ::= "random "
10                | ϵ
```

Listing 4.4: The syntax for an expression written in BNF.

One note is that the operators do not follow the usual calculations rules of multiplication and division before addition and subtraction, but simply does the calculations from left to right. Most end-users are not used to think in this way, but this is purely an issue of making it easy for our self. This is therefore one element that should be changed in the future.

The `random` keyword is used to get a random number. The number written after this keyword will be used as the max value and a random number between 0 and the max value will be found, both included. To get a random number between two numbers a small calculation is needed, e.g. `10+random 20` will give a random number between 10 and 30. It is important that there is at least one whitespace between the `random` keyword and the number, else it will be interpreted as part of a variable.

4.4.3 Usage

In this section we will illustrate how the new keyword arguments actually looks like. We will first show how the same commands used in the sloppy argument test, see Section 4.3.2, looks like when using keyword arguments instead. Notice that the arguments can still be written in random order and that many arguments can have very short keywords. We have tried using a shorter synonym where these were used in the test of the sloppy arguments.

```
1 rectangle start=20,20 width=100 height=150
2 c c=100,100 r=75
3 ellipse start=75,75 end=300,200 color=#0f0
4 line from=300,275 color=red penwidth=thin to=75,225
5 l l=52 color=0,0,255 a=32 pen=2 s=43,37
```

Here the last command, Line 5, is much easier to read, though still quite complex. The reason is that it uses all the shortest synonyms available. If we write it as expressive as possible, and in a more logical order, it will be even more readable and look like this:

```
line startposition=43,37 length=52 angle=32 color=blue penwidth=2
```

We have also made a small test of the expressions. Here we try combining the expressions with different value types and with the use of the `random` keyword:

```
rectangle start=10*5,40+5 width=10+random 100 height=100/2
```

The drawing of the rectangle is then disclosed and it was as expected:

```
rectangle startpoint=50,45 width=31 height=50
```

4.4.4 Results

In this experiment we changed the command line from sloppy arguments to keyword arguments and we believe it now have much better readability. Based on an analysis of different approaches to arguments we feel this is the best solution for end-users who have little or no programming experience. It is also easier to make a lexical and syntactic analysis when using keyword arguments which means that it will be easier to give good feedback later on. Now that the command line is done we can move on to creating our own scripting language in the next experiment. This language will use the command line structure we have developed in the last two experiments.

In regards to the properties and guidelines for self-disclosing, there have not been any changes when we made the shift from sloppy to keyword arguments.

4.5 Experiment 4: Scripting Language

In order to let end-users solve repetitive tasks by programming, we will in this experiment introduce a scripting language with for-loop construction and if-condition. The idea is to give end-users the possibilities to iterate commands available from the previous experiments.

We need to make the disclosing such that it makes sense, and is easy to understand for an end-user and still work as a scripting language. If we were to use an existing scripting language we would most likely need to make an abstraction of this — a kind of intermediate language — so the user do not see and use the script, but our understandable abstraction. We would then need to have a deep understanding of the existing scripting language and make an interpreter that translates our abstraction and then either send it to the script interpreter or interpret the script its self. Therefore we believe it is easier to create our own simple scripting language from scratch and make a parser and interpreter for it.

4.5.1 Design

The syntax for our loop and if structure is shown in Listing 4.5. We strive to create a for-loop which looks like mainstreamed system programming languages, but still is easy to read for an end-user. Instead of using curly brackets ({ }) to encapsulate blocks, as it is done in many C-like programming languages, we believe it is more readable to use **do** and **end** in the for-loop construction. The same principal is used with the if-condition, where we have chosen to use **then**, **else** and **end** instead of curly brackets. Since we force a newline after each command we also need one after **do**. This will at the same time force a bit of structure on the code so it has better readability. We considered using indention as a scope rule as well, like it is done in Python, but we decided against it as it increases the possible errors made by the end-user.

```
1  <script>     ::= <statements>
2  <statements> ::= <statement> <EOL> <statements>
3                 | ϵ
4  <statement>  ::= <command>
5                 | <for>
6                 | <if>
7                 | <assignment>
8  <for>        ::= "for" <variable> "=" <expression> "to" <expression>
9                        "do" <EOL> <statements> <EOL>
10                       "end"
11 <if>         ::= "if" <condition> "then" <EOL> <statements> <EOL>
12                       "else" <EOL> <statements> <EOL>
13                       "end"
14                | "if" <condition> "then" <EOL> <statements> <EOL>
15                       "end"
16 <assignment> ::= <variable> "=" <expression>
```

Listing 4.5: The syntax for a for-loop and if-condition written in BNF. The <command> and <expression> keywords can be seen in the keyword experiment Section 4.4. There can be an unspecified amount of whitespaces on both sides of any <key>'s.

A `<condition>` is at least one boolean expression. We have decided to include both single equal (=) and double equal (==) as the "is-equal-to". The reason is that we believe

most end-users see a single equal as the most logical way of expressing it. In many system programming languages the single equal is only used for assignment, but we remove this possibility from within a condition for simplicity. Between the boolean expressions we use both the approach from C (`&&` and `||`) and from Pascal (`and` and `or`), and the normal bitwise operators (single `&` and `|`).

Parser and Interpreter

Because we want to expand from command lines to a scripting language with for-loop, if-condition and assignment, we do not want to create a whole new parser. Instead, we will try to reuse the already existing parser for command lines from previous experiments, and only make some minor changes if that should be needed.

The idea is to create a new parser for our new language to wrap around the already existing command line parser. This means when the scanner in our parser detects a command, it use the already existing command line parser, thereby, reusing the existing parser and minimizing too much extra work. The idea is illustrated in Figure 4.4.

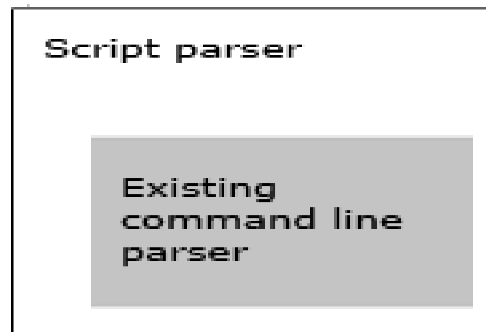


Figure 4.4: An illustration of the idea of a wrapper parser around the existing command line parser.

As with our parser, our interpreter should also use the already existing interpreter for command lines. All the interpretations of for-loops, if-conditions and assignments will be transformed into equivalent C# code and executed.

4.5.2 Usage

Now that we have a scripting language which gives us the ability to use loops and conditions we can solve some of the examples of repetitive tasks listed in Section 3.1.1. Figure 4.5 is an illustration of the script and the outcome for drawing a grid, which is one of the tasks. The script used can also be seen in Listing I.5. This script consists of two loops for drawing first the vertical lines and then the horizontal lines. Both loops have a condition such that the center line is drawn in a different color than the rest. At the start of the script there are a few assignments which can be used for drawing the grid with a different count of cells and in a different size. The scripts for most of the other tasks can be seen in Appendix I.

All of these scripts work as intended. The current version of the application is, however, not very good at giving feedback when an error occurs. Some errors will report a specific line number, some will just give notice that an error exists somewhere, and finally

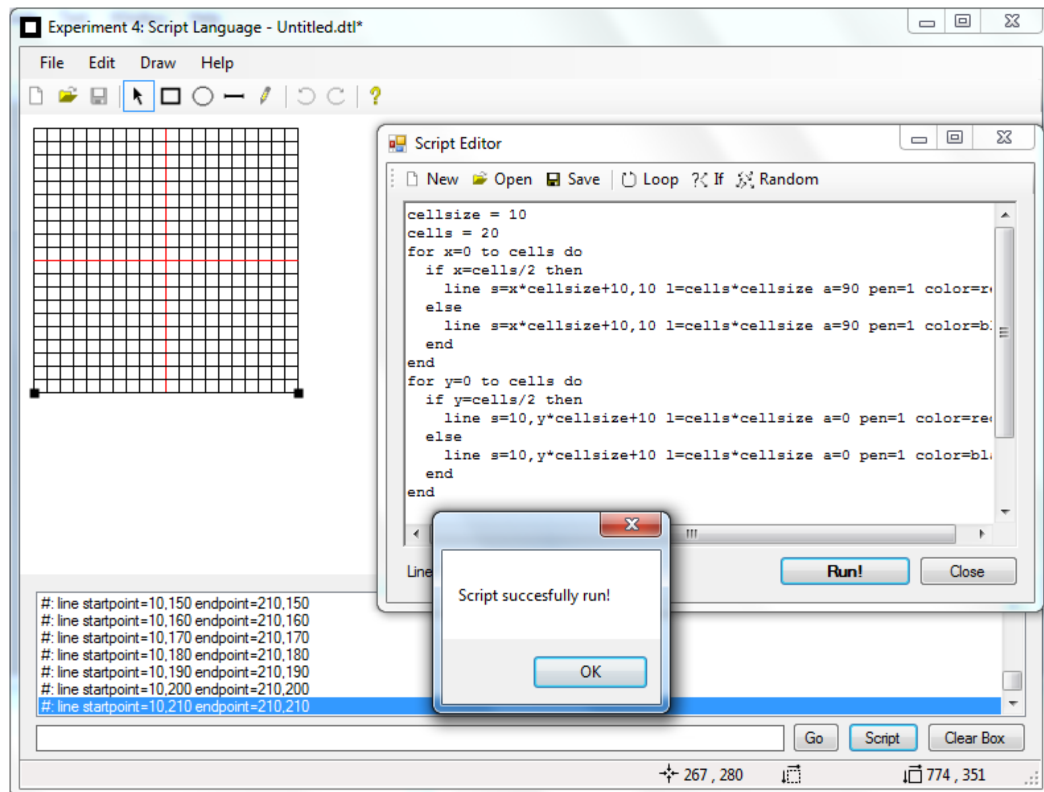


Figure 4.5: The drawing of a grid using a script. Notice that every line drawn is also disclosed in the output box.

some errors will never be reported and is only visible because nothing happens or even worse as a crash. This is of course not ideal and something we will need to change in the future.

On Figure 4.5 it is also possible to see the simple script editor. Notice that there is a toolbar with "New", "Open" and "Save" buttons used for managing the script as files. The toolbar also have buttons for easily adding loop-structures, conditions and the random keyword. These will all open a small dialog where further options can be set, like the minimum and maximum value of the random function.

4.5.3 Results

We succeeded in creating a scripting language and a parser and interpreter for it. This language is simple and it will be easy to expand it because of the way we have created commands and types. It is therefore suitable to use as a base for the language we need to create in our future project. The language include loop and condition structures and have the possibility to use variables in arithmetics expressions.

Only one guideline for self-disclosing, Section 2.5, from the last experiment have been improved by adding the scripting functionality. It is the "experimentation" guideline about the possibilities for experimentation which we believe have been improved greatly by adding the possibility of writing more than one line at once and using loops, conditions and variables.

4.6 Summary

During our experiments we managed to expand the DrawTools application with several things. First, we added a disclosure class which discloses many of the actions that is possible to do, including the drawing of different kind of shapes. Next, we added a command line and a parser and interpreter for this which made it possible to write the commands that were disclosed, thereby drawing shapes through a programmatic access to the applications API instead of through an interactive access by using the mouse to click on toolbars and the drawing area. Originally we made this command line sloppy, which means that command and arguments could be written in any order without any identification. It worked but it had a low readability and we believe it to be to complicated for an end-user to use. Therefore we changed it such that the command will be written first and then all of its arguments will be written in a random order afterwards but this time with an identifier, also called keyword arguments. This increased the readability greatly. Finally we created a small scripting language which used English verbs, such as *do* and *end* instead of symbols, such as curly brackets, normally used in several programming languages. Our scripting language have great readability and is fairly easy to use. We added a script editor to the application wherein the scripting language could be written. Several features in this editor will help an end-user to learn the language. The language itself have loops and condition structures and the possibility to use variables and arithmetic expressions. Many arguments for the commands uses domain specific types, such as a position. This language is easy to extend with new structures, commands and types.

Our application fulfills all of the properties for self-disclosing described in Section 2.5. It does, however, not fulfill all of the guidelines. The only guideline not fulfilled is the "scaffolding" guideline, which is about how the application will gradually learn the end-user the languages and commands be gradually increasing the complexity of the disclosed commands. All commands in all arguments have several synonyms, e.g. shortcuts, and we could make it such that the system would start with the most expressive keyword and end with the shortest. We, however, believe that it will be confusing for an end-user if the same action suddenly discloses what appears to be a different command than they are used to.

Chapter 5

Epilogue

In this chapter, we will conclude our project by answering the questions stated in the problem statement, Section 3.4. We will also give an evaluation of our project and see what we could have done better or what we should keep doing. Finally, we will end with a future work section wherein we will describe our thoughts for the later work with this project.

5.1 Conclusion

From the analysis we have found a problem with end-users who want to start programming. One main issue is because of the gab between how a human brain represents a problem and how the computer understands and accepts the same problem. In addition, there are different goals with programming, seen from an end-user perspective. One group wants to create the software itself, the other group wants to use programming as a tool to solve e.g. repetitive tasks. Our tool is aimed at the second group, and will help end-users to start programming in their usual application environment. It is inspired by two applications: AutoCAD and Eager. We use the idea of learning by observation based on AutoCAD, and programing by demonstration and automated repetitive task detection based on Eager.

In this project period, we have created a platform which is a part of our final tool. Our platform is implemented in a drawing tool as functionalities to disclose end-user actions, command lines and a very small and domain specific scripting language. In our problem statement we defined a few questions which we tried to answer by developing this platform. In the following paragraphs we will try to answer these questions:

How do we create a command line and scripting environment for a drawing application?

Any command line and scripting environment needs a parser and an interpreter. Through our experiments we have developed and expanded these. They are combined by the fact that anything written in the command line can be used as a statement in the scripting language. To better suit end-users, however, we need to improve both the command line and the scripting environment such that they both give much better feedback and help the end-user while he programs.

How do we disclose the end-user actions in a meaningful and easy way?
We did it by creating a disclosure class that were added to the base drawing shape class. This disclosure class is updated every time the shape is updated and will print out a meaningful text for the action that have just occurred. The text will work in the command line and scripting language, which is why the meaningfulness of what is disclosed depends on how scripting language is designed.

How should this command line and scripting language be designed?
As mentioned, the command line is designed such that any command can be used in the scripting language as well. Each command have several synonyms for make it more likely that any end-user will write a correct name. The arguments for the commands use the keyword argument approach which we believe to be the approach with the highest readability. This will both make it easier for an end-user to read and write his own code, but also make it much easier to understand what is disclosed. The language is also designed such that as few errors as possible can arise. This includes that several symbols is used for the same thing and whitespace mean as little as possible. Furthermore the language uses English verbs where possible instead of symbols.

The scripting language should be expandable as we need it as a base for the next part of our vision and we have managed to design it such that new commands and structures can be added with ease.

The main question was: *How do we motivate end-users to program?* Our idea for this is by adding our tool into their used application environment, and thereby learning them to program while they do their everyday work. The point is not to require end-users to change their environment in order for them to learn programming languages, because it is time consuming and difficult.

5.2 Evaluation

It has always been difficult to find a good idea when we start in a new research area, and it costs a lot of time to get started. However, in this project, we did some preliminary work wherein we tried to test some of our ideas through prototypes. Our early tests of ideas helped us a lot to understand the end-user programming research area and where to make further researches. It turns out that starting with creating or at least trying to create our ideas early in the project gave us far more useful ideas than just reading papers and making analysis. There is, however, a disadvantage about starting with prototypes, because we could have wasted too much time on an irrelevant prototype, since we do not have enough knowledge of the research area.

Regarding to the analysis, we lacked a bit structuring in our approach. What we did, was just searching for papers and take out some of the interesting without having a plan for what or which areas within end-user programming we were looking for. However, this lack of structuring is caused by the lack of knowledge in this research area, but even when we felt it was a bit unstructured, we managed to get a wide understanding of the area. To do it more structured, we should have made a list of questions which we would have tried to answer by searching for the right papers.

For the implementation phase of the project, we divided the implementations into ex-

periments instead of the usual design-implementation-documentation phase. This way, we could focus on some key features in each experiment instead of thinking too much ahead. However, a downside of this approach is that it may require too much refactoring in the next experiment, because there is no upfront designs. In our case, we had to change from sloppiness arguments to keyword arguments which required some changes in our code.

5.3 Future Work

During this project we have only fulfilled part of our vision from our problem statement, Section 3.2. In the next semester we will continue on this project to get as close to the vision as possible. This require us to do much work in several areas.

First we need to work with the Windows hooks again to catch events happening for any application. This will most likely give as a large amount of events we then need to combine and structure into some more meaningful user actions. We need to expand our scripting language such that it can express these user actions. Hopefully it will only require a few more types and several new commands. It might, however, also be a good idea to expand the scripting language with abstractions and a more comprehensive standard library, e.g. with mathematical functions such as the trigonometric functions sine and cosine that would have been useful when working with circles and angles in a drawing application. The possibility for abstractions will also help the community to grow the language[ACF+07] as they can create their own libraries. The current application is not very helpful when the end-user is making errors in e.g. syntax. We need to improve this in the next project such that it will give a good and constructive feedback saying exactly where the error is, what it is and how to fix it. The application should also give feedback while the code is being written. This could include syntax coloring and an explanation of the current command, such as what arguments are needed and what values each argument can be.

To get even closer to our vision we need to analyze the user actions and look for repetitions. These will probably not be easy to spot as there might be events for several different applications at the same time and not just the one the end-user is actually using. When we catch a repetition the application needs to inform the end-user of these and provide a script for completing their tasks.

Appendix A

Bibliography

[ABE09] Robin Abraham, Margaret Burnett, and Martin Erwig. Spreadsheet programming. 2009.

[ACF+07] Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, and Guy L. Steele Jr. Project fortress. *Linux Magazine*, September 2007. [Online; accessed 14-December-2009, `http://research.sun.com/projects/plrg/Publications/linuxMagazine.pdf`].

[BB04] Laura Beckwith and Margaret Burnett. Gender: An important factor in end-user programming environments? In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 107–114, Washington, DC, USA, 2004. IEEE Computer Society.

[BR96] Cathy Brand and Cyndi Rader. How does a visual simulation program support students creating science models? In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 110, New York, NY, USA, 1996. ACM.

[Cyp93a] Allan Cypher. Eager: Programming repetitive tasks by demonstration. In Allan Cypher, editor, *Watch What I Do*, page Chapter 9. The MIT Press, 1993.

[Cyp93b] Allan Cypher, editor. *Watch What I Do*. The MIT Press, 1993. [Online; `http://acypher.com/wwid/index.html`].

[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.

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

[KAB+09] A. Ko, R. Abraham, L. Beckwith, M. Burnett, M. Erwig, J. Lawrence, H. Lieberman, B. Myers, M. Rosson, G. Rothermel, C. Scaffidi, M. Shaw, and S. Weidenbeck. The state of the art in end-user software engineering. 2009.

[MCB+08] Robert C. Miller, Victoria H. Chou, Michael Bernstein, Greg Little, Max Van Kleek, David Karger, and mc schraefel. Inky: a sloppy command line

for the web with rich visual feedback. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 131–140, New York, NY, USA, 2008. ACM.

[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].

[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].

[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.

[Slo71] Aaron Sloman. The computer revolution in philosophy: Philosophy science and models of mind. In *Proceedings of the 2nd International Joint Conj3rence on Artificial Intelligence*, pages 270–278. North-Holland Publishing Company, 1971.

[Sto09] Derrick Story. From darkroom to desktop - how photoshop came to light, 2009. [Online; accessed 3-November-2009, http://www.storyphoto.com/multimedia/multimedia_photoshop.html].

Appendix B

DrawTools

Figure 1.2 shows a screenshot of the original DrawTools application before our experiments. It is worth noticing here that every interaction is done using the mouse, e.g. by using the toolbar. Figure B.1 shows a class diagram of the DrawTools application. The `DrawArea` class is where the drawing takes place, and it can be seen as the large white background of the application. The drawing is done by placing shapes on this draw area, and these shapes are the `DrawObject` classes. To draw objects you use a `Tool`. These tools are selected e.g. by clicking the buttons in the toolbar. When such a tool is selected every mouse action done on the draw area will go through this tool and at the end the appropriate `DrawObject` is created and added. The `Command` classes are used for the undo/redo feature. Every time a new shape is drawn it also creates a `CommandAdd` object and adds this to the command history.

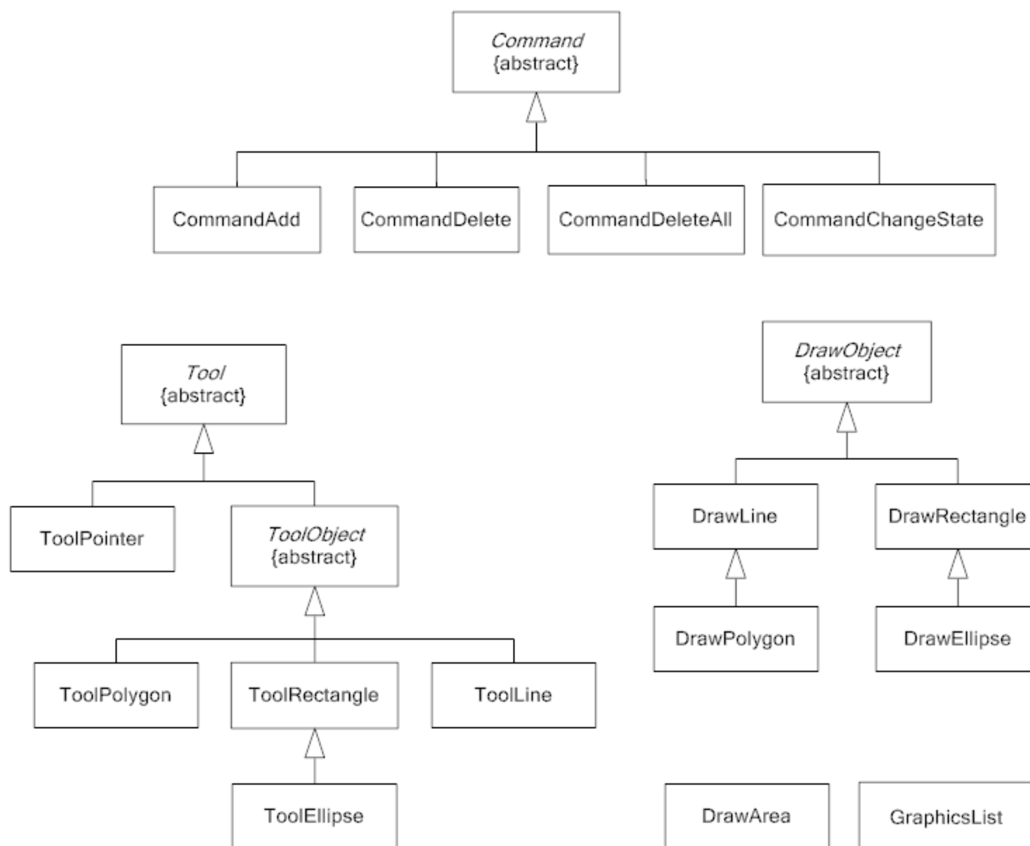


Figure B.1: Class diagram over the original DrawTools. Notice here that they are divided into three main groups: Commands, Tools and DrawObjects.

Appendix C

AutoCAD Commands

Here is how AutoCAD handles drawing of different simple shapes.[1]

Rectangle

```
rectangle, rectang or rec
```

A rectangle is always drawn using the points of two diagonally opposite corners. These are selected one after the other.

Circle

```
circle or c
```

The `circle` command will give you five possible options:

(default) Specifies the center position and the dimension of the radius.

D Specifies the center position and the diameter dimension.

2P Specifies the circle by picking 2 points on its diameter.

3P Specifies the circle by picking 3 points through which the circle will pass.

TTR Specifies the circle by picking two lines, arcs or circles for the circle to be tangent to, and entering the dimension of the radius.

Ellipse

```
ellipse
```

The `ellipse` command in general requires the length of the two axis, but can give you three possible options:

(default) Specifies the end points of the first axis and one end point of the second axis or an eccentricity rotation.

C Allows specification of center point of the ellipse rather than first axis endpoint.

[1]This list is partly based on `http://academics.triton.edu/faculty/fheitzman/commands.html` and partly on own experience.

Line

```
line or l
```

The `line` command in general requires a From point and a To point. The line command will automatically start the drawing of a polygon by continuing the next line from where the previous ended. Following are options after the first point have been specified:

(default) Sets the To point and starts a new line at the same location.

C Closes the polygon back to first “From Point”. Requires at least two lines to have been drawn.

U Undoes the last line segment.

Polygon

```
polygon or pol
```

The `polygon` command requires a count of sides (at least 3) and will draw a polygon with equal side lengths. E.g. four sides will create a square.

E Specifies size and rotation of the polygon by picking endpoints of one edge, like a line.

C Circumscribes the polygon around a circle with a center point and a specified radius.

I Inscribes the polygon within a circle with a center point and a specified radius.

Polyline

```
pline or pl
```

A Polyline is a combination of several elements (lines and arches) with a wide range of options. To use the CAD drawing outside the computer, like in a cutting machine, it is often required to draw everything using the `pline` command.

General Attributes

Both the color and the thickness of a the lines in a shape is specified by the current layer it is drawn in. A different color and thickness can, however, be specified before and after the creation of the shape, but not as part of the command itself.

Appendix D

SVG Commands

Here is how SVG handles the drawing of some different simple shapes.[1]

Rectangle

```
<rect x="1" y="1" width="1198" height="398" />
```

A rectangle takes a starting point (`x` and `y`), a `width`, and a `height`. Only the `width` and `height` are required as the default starting point is (0,0).

Circle

```
<circle cx="600" cy="200" r="100" />
```

A circle takes a center point (`cx` and `cy`) and a radius (`r`). Only the radius is required as the default center point is (0,0).

Ellipse

```
<ellipse cx="900" cy="200" rx="250" ry="100" />
```

Ellipse takes a center point (`cx` and `cy`) and two radius', one for the x-axis (`rx`) and one for the y-axis (`ry`). Only the two radius' are required as the default center point is (0,0).

Line

```
<line x1="100" y1="300" x2="300" y2="100" />
```

Line takes a starting point (`x1` and `y1`) and an ending point (`x2` and `y2`). No attribute is required, but since both points default value is (0,0) it would not make much sense to omit both.

Polyline

```
<polyline points="50,375 150,375 150,325 250,325 250,375 350,375
    350,250 450,250 450,375" />
```

[1] This list is based on the SVG specification: `http://www.w3.org/TR/SVG11/shapes.html`.

Polyline takes a list of `points`. The points are separated by a space and they consists of an x-value and a y-value separated by a ",". This attribute is required. A line will be drawn between each neighboring pair of points.

Polygon

```
<polygon points="350,75 379,161 469,161 397,215 423,301 350,250 277,301
    303,215 231,161 321,161" />
```

As with the polyline, the polygon also takes a list of `points` that work the same way. Main difference is that the first and last point in the list will also be connected by a line and the shape can have a "fill" color. The example draws a star.

General Attributes

All shapes have attributes which specifies the stroke (border) width and color, and the fill color (line and polyline do not have the fill as they do not have any "inside"). They all also have a `transform` attribute which makes it possible to specify several things including a rotation of the shape.

Appendix E

Types

All types can be recognized either through a pattern or a simple textual equality, or both. For experiment 2, sloppy arguments, there is no space allowed inside any type so here these are only used to clarify each part of the patterns. From experiment 3, keyword arguments, space is allowed and integers and floating point values are replaced with expressions instead which can include variables and algebraic operators.

Integer

An integer is simply a string that can be parsed by the C# method `Int32.TryParse()`.

Colorname

A colorname is any name of a color that exists in the C# enum `Color` and can be parsed by the `Color.FromName()` method.

Angle

An angle can be specified both as and integer and as a floating point value. To make sure that no localization can confuse a floating point value with a position we define it as two integers divided by a ".".

```
Angle ::= Integer . Integer
        | Integer
```

Color

A color can be described in many different ways. On a computer RGB is usually used. RGB is short for Red-Green-Blue and is three integers, each from 0 to 255, describing the intensity of each of these colors. In HTML, and SVG, these integers are usually expressed in hexadecimal, which combined is 6 digits long. In HTML it is also possible to express a color using only 3 digit hexadecimal. Here each number only goes from 0 to 15.

```
Color ::= # 3-digit-hexvalue
        | # 6-digit-hexvalue
        | Integer , Integer , Integer
        | Colorname
```

PenWidth

The pen width type is used to express the width of the border of any drawing object. This width is usually expressed in pixels, meaning an integer. We add some textual strings to make it more readable.

```
PenWidth ::= Integer
           | ``thin'' | ``thick'' | ``small''
           | ``medium'' | ``big'' | ``huge''
```

Position

A position is expressed as a coordinate (x and y) describing the distance from the upper left corner of the drawing.

```
Position ::= Integer , Integer
```

Appendix F

Commands

The following is a complete list of all commands available through our experiments. Most of them are available from experiment 2 (sloppy command line) and will all include a list of parameter keywords which is only used from experiment 3 (keyword arguments). As in the report we divide these up into three groups: Drawing shapes, manipulating shapes and application handling.

F.1 Commands for Drawing Shapes

Rectangle

```
rectangle, drawrectangle, rect, rec, r
```

The specification of a rectangle is that it has four sides that are parallel in pairs, and all corners have an angle of 90°. The parallel sides therefore have the same length. So for drawing a rectangle we only need to specify its position and its width and height. A common approach to this is to specify the upper left corner (closest to (0,0)) and either the width and height or the lower right corner (furthest away from (0,0)). Therefor the rectangle command should have two parameter sets.

Parameter	Type	Keywords
Upper left corner	Position	`first`, `firstpoint`, `upperleft`, `startpoint`, `startposition`, `start`, `s`, `sp`
Lower right corner	Position	`second`, `secondpoint`, `lowerright`, `endpoint`, `endposition`, `end`, `e`, `ep`
Position	Position	`first`, `firstpoint`, `upperleft`, `startpoint`, `startposition`, `start`, `s`, `sp`, `from`
Width	Integer	`width`, `w`
Height	Integer	`height`, `h`

These two parameter sets are distinguishable both by the count of parameters and the fact that the first have two `Position` types while the second have only one and two `Integer` types instead. In DrawTools there is no requirement that the first corner is the

upper left, therefore there is no need to make constrains about the order of the `Position` parameters. The order of width and height, however, do matter and we have decided that width is the first, as this approach seems the most common.

Circle

```
circle, drawcircle, c
```

The mathematical function for a circle consists of a position of the center and a radius, so this must be available as a parameter set. In DrawTools the circle is drawn using the same approach as a rectangle with the two diagonally opposite corners, but this approach will only draw a circle if the rectangle is a square, therefore this will instead be part of the ellipse command.

Parameter	Type	Keywords
Center	Position	`center`, `c`, `centerposition`, `centerpos`, `centrum`, `origin`
Radius	Integer	`radius`, `r`

Some might want to draw the circle by specifying the diameter instead of the radius, but this parameter set would have the exact same types as with the radius which makes them indistinguishable (for the sloppy command line).

Ellipse

```
ellipse, drawellipse, e
```

Mathematically an ellipse consists of either a center point and two radius' or two fixed positions and a constant distance to both of them. In DrawTools an ellipse is drawn as a rectangle which means it can also have the possibility of using two diagonally opposite corners.

Parameter	Type	Keywords
Center	Position	`center`, `c`, `centerposition`, `centerpos`, `centrum`, `origin`
Radius X	Integer	`radiusx`, `radius1`, `rx`, `r1`
Radius Y	Integer	`radiusy`, `radius2`, `ry`, `r2`
First fixed point	Position	-
Second fixed point	Position	-
Combined distance	Integer	-
Upper left corner	Position	`first`, `firstpoint`, `upperleft`, `startpoint`, `startposition`, `start`, `s`, `sp`
Lower right corner	Position	`second`, `secondpoint`, `lowerright`, `endpoint`, `endposition`, `end`, `e`, `ep`

Because of the way an ellipse is drawn in DrawTools the fixed position approach also requires that both positions have the same x-coordinate or the same y-coordinate. This adds a constrain to the parameters for this parameter set, which can either be resolved by returning and error notice or e.g. by finding an average x-coordinate to use instead.

Line

```
line, drawline, l
```

A simple line is from one point to another. However, in mathematical terms it is also from one point and then an angle (or more correctly the tangent of an angle). The mathematical line, however, is unlimited which is not possible in DrawTools. Therefore a length of the line is needed.

Parameter	Type	Keywords	
First point	Position	`first`, `firstpoint`, `upperleft`, `startpoint`, `startposition`, `start`, `s`, `sp`, `from`	
Second point	Position	`second`, `secondpoint`, `lowerright`, `endpoint`, `endposition`, `end`, `e`, `ep`, `to`	
Position	Position	`first`, `firstpoint`, `upperleft`, `startpoint`, `startposition`, `start`, `s`, `sp`, `from`	
Length	Integer	`length`, `l`	
Angle	Angle	`angle`, `a`	(optional)

F.2 Commands for Manipulating Shapes

Move

```
move, m
```

The move command is used to move the selected shapes to a new location. The current location of all shapes is that of their upper left corner. This command is then used by specifying where this corner should be located instead. It is done by setting any moving direction to a distance. It is also possible to move just one handle, or point, which will result in a resizing for rectangles and ellipses. This command was changed drastically when we changed the command line to use keyword arguments instead.

Parameter	Type	Keywords	
Up	Integer	`up`, `u`	(optional, not sloppy)
Down	Integer	`down`, `d`	(optional, not sloppy)
Left	Integer	`left`, `l`	(optional, not sloppy)
Right	Integer	`right`, `r`	(optional, not sloppy)
Point	Integer	`point`, `handle`, `p`, `h`	(optional, not sloppy)

Resize

```
resize, size
```

Because of the way shapes are implemented in DrawTools we can create the command so there is no difference in the way they are resized. However, we have decided to also add ways that are unique for each type of shape.

Parameter	Type	Keywords	
New width	Integer	`width`, `w`	
New height	Integer	`height`, `h`	
New radius X	Integer	`radiusx`, `radius1`, `rx`, `r1`	(not sloppy, only ellipse)
New radius Y	Integer	`radiusy`, `radius2`, `ry`, `r2`	(not sloppy, only ellipse)
New radius	Integer	`radius`, `r`	(not sloppy, only circle)
New diameter	Integer	`diameter`, `d`	(not sloppy, only circle)
New length	Integer	`length`, `l`	(only line)
New angle	Angle	`angle`, `a`	(only line)
New length	Integer	`length`, `l`	(not sloppy, only line)

All shapes can be resized either by specifying a new width and height. The ellipse and circle is actually the same shape so both ellipse and circle can be resized using radius,

diameter or two radius'. A line can be resized by specifying a new length and angle. If only the length is specified it will keep its current angle.

Delete

```
delete, remove, del, rem, d
```

This command command is to remove the selected shapes from the drawing.

No parameters

Delete All

```
deleteall, removeall
```

This command command is to remove all shapes from the drawing.

No parameters

Select All

```
selectall
```

This command command is to select all currently existing shapes in the drawing.

No parameters

F.3 Commands for Application Handling

Color

```
color, setcolor, usecolor, defaultcolor
```

The color command is used to specify the default border color used by all the drawing commands.

It does, however, not have the shortcut `c` as that is used by the circle command.

Parameter	Type	Keywords
Color	Color	`color`, `value`

Exit

```
exit, quit, close, q
```

The exit command is used for closing the application. It will use the close method already available in DrawTools which means it will first ask whether or not you want to save the current drawing if it has been changed.

The exit command have always come in several forms in applications that have used a command-like approach, e.g. a Unix shell or games with a console. Therefore we also need to have several keywords for the exit command.

No parameters

New

```
new, newdrawing, newfile, n
```

The new command will create a new drawing, but just as with the exit command it uses the method in DrawTools that will ask if the old drawing should be saved.

No parameters

Open

```
open, opendrawing, openfile, o
```

The open command will display the open file dialog — though it will ask to save the old drawing first if needed.

No parameters

Redo

```
redo
```

The redo command will not repeat the last command but redo the last undone drawing command. Therefore the redo command will only have an effect if something have been undone and nothing have been drawn afterwards.

The redo command do not have the shortcut keyword `r` as it is already used by the rectangle command.

No parameters

Save

```
save, savedrawing, savefile, s
```

This command will save the current drawing. If the drawing have not been saved before, meaning it does not exists as a file, it will show the save-as dialog.

No parameters

Undo

```
undo, u
```

The undo command will undo the last drawing command. The undone command will be saved in a history, but if any other draw command is called afterwards this history will be cleared. As long as there is commands in this history the redo command can be called to redo them.

No parameters

Width

```
width, setwidth, usewidth, defaultwidth, w
```

With the width command it is possible to set the default PenWidth used by all the drawing commands.

Parameter	Type	Keywords
Pen width	PenWidth	`pen`, `width`, `penwidth`, `value`

Appendix G

Sloppy Parsing Algorithm

In this appendix we will describe how we have implemented the parser and interpreter for the sloppy command line. The basics are described in Section 4.3.1. Here we will focus on the implementation of the types, commands and the full design of the parsing algorithm.

Types

All types are implemented as classes which extends the class `TypeObject`. This base class has two abstract methods which each type class is required to override. These are `Recognizer()` and `GetValue()`. The `Recognizer()` is used to check if a string could be of this type. Each type class needs to check the string in their own way, e.g. with the use of `String.CompareTo()` or by checking if certain characters are present. At the end the method should return a new instance of the type class if the string fits, else `null`. Listing G.1 shows some of the `Recognizer()` method of the PenWidth type class, `TypePenWidth`. Notice that the new instance of the `TypePenWidth` class is constructed with the value of the string. This value is stored and retrieved again by the interpreter with the `GetValue()` method. This method returns an `Object` as this value could be anything, depending on the type class.

```
1  public override TypeObject Recognizer(string possibleType)
2  {
3      int realValue = 0;
4      if (int.TryParse(possibleType, out realValue))
5      {
6          return new TypePenWidth(realValue);
7      }
8      if (possibleType.CompareTo("thin") == 0)
9      {
10         return new TypePenWidth(1);
11     }
12     [...]
13     else if (possibleType.CompareTo("huge") == 0)
14     {
15         return new TypePenWidth(4);
16     }
17     return null;
18 }
```

Listing G.1: The Recognizer() method of the class TypePenWidth. Notice that it both recognizes normal integers as well as simple textual strings such as “thin” and “huge”.

And instance of all available type classes are saved in a library called `TypeLibrary`. This class uses the Singleton pattern and have two methods. A `Load()` which is called at the start of the application to create an instance of all these type classes, and a `Recognizer()` method which is used to simply call the `Recognizer()` method of all type classes. This is used by the parser to find all possible types for a string and will return a `List` as several types could fit.

Commands

The DrawTools application already have some classes that uses the name “Command”. Therefore we have decided to call our own commands for “Keywords” in the code instead.

Just like with the types all commands are implemented as classes that extends one class, `KeywordObject`. This base class, however, have three abstract methods: `Recognizer()`, `GetParameterSets()` and `Action()`. The `Recognizer()` is similar to the method of the type class with the same name. All commands are also stored in a library, this one called `KeywordLibrary` and it to works just as the `TypeLibrary` does.

The `GetParameterSets()` method of keyword classes is used to return a list of parameter sets that can be used with this command. Each set is a list of type objects. If a command does not take any arguments this method will return an empty list. To increase performance this list is created when the class is instantiated so it is only done once and not for every call to `GetParameterSets()`. Listing G.2 shows how this list is populated for the rectangle command, `KeywordRectangle`.

```
1  private void CreateParameterSets()
2  {
3      // Position Position
4      List<TypeObject> tempParameterSet = new List<TypeObject>();
5      tempParameterSet.Add(new TypePosition());
6      tempParameterSet.Add(new TypePosition());
7      tempParameterSet.Add(new TypeColor());
8      tempParameterSet.Add(new TypePenWidth());
9      this.parameterSets.Add(tempParameterSet);
10
11     // Position Integer Integer
12     tempParameterSet = new List<TypeObject>();
13     tempParameterSet.Add(new TypePosition());
14     tempParameterSet.Add(new TypeInteger());
15     tempParameterSet.Add(new TypeInteger());
16     tempParameterSet.Add(new TypeColor());
17     tempParameterSet.Add(new TypePenWidth());
18     this.parameterSets.Add(tempParameterSet);
19 }
```

Listing G.2: The population of the parameter sets list in the class KeywordRectangle. Here there are two parameter sets. One with two Positions, and one with one Position and two Integers.

The `Action()` method of the keyword classes is called by the interpreter when the command is executed. It takes a parameter set as argument. This parameter set will contain the found parameters, and their values, from the command line string. Each keyword class needs to go through this parameter set and extract all values, and hereafter use them. The `KeywordRectangle` class will at the end create an instance of `DrawRectangle` which is the DrawTools class for a rectangle shape that can be placed on the drawing area.

Parser and Interpreter

The parser have its own class which also uses the singleton pattern. In this experiment it only have one method, `ParseLine()`.

The interpreter also have its own class which again uses the singleton pattern. And just like with the parser it only have one method, `InterpretLine()`. This method takes the returned value from the parser as input. Hereafter it finds the command and calls the `Action()` method.

The parsing algorithm can be divided into two parts: A lexical analysis and a syntactical analysis, and these will be described in the following sections. We will primarily use pseudo code to describe it here.

Lexical Analysis

As described earlier we have decided that the string from the command line will consists of several parts divided by a space. Therefore the first part of the lexical analysis is simply to split the string. Hereafter it will go through each part and first find possible keywords and thereafter possible types.

```
1 parts = split ( commandlinestring , `` '' )
2 foreach part in parts do
3   part.keyword = possibleKeywords
4   part.types = possibleTypes
```

The algorithm for finding both possible keywords and possible types look alike so only the possible keywords will be described further. All keywords are stored in a library.

```
1 foreach keyword in library do
2   foreach synonym in keyword.synonyms do
3     if synonym == part then
4       add keyword to possibleKeywords
5       break
```

Syntactical Analysis

The syntactical analysis, shown in Listing G.3, is quite complex with nested loops in six levels. The general purpose of this part is to find all possible commands, all possible parameter sets for each of these commands and finally pick the command and parameter set that is the most likely or best suited.

```
1  foreach part in parts do
2    foreach keyword in part.keywords do
3      add keyword to possibleCommands
4      foreach parameterSet in keyword.parameterSets do
5        foreach parameter in parameterSet do
6          foreach part2 in parts except part do
7            foreach type in part2.types do
8              if parameter == type then
9                possibleCommands[keyword].parameterSets[parameter].value
                     = type.value
10               break(2)
11       calculate possibleCommands[keyword].parameterSets.score
12     calculate possibleCommands[keyword].score
13 return possibleCommand with highest score
```

Listing G.3: Pseudo code for the syntactical analysis section of the sloppy command line parsing algorithm.

Finding all possible commands requires to run through each of the part's found above (Line 1) and check if there were any possible keywords (Line 2).

Now we have each possible command. Next we need to go through each of the parameter sets that might be available in each command (Line 4) and check if the rest of the command line parts fit with this. This is done by first going through parameters in a possible parameter set (Line 5) and then go through each part of the command line again (Line 6). This time, however, we will go through the possible types (Line 7) instead of keywords. These types, or arguments, are then checked to see if they fit the parameter (Line 8) and if they do we save it.

After we have checked a parameter set we calculate a score (Line 11) on how good this parameter set fits the written command line. This score is based on how many arguments that fitted the parameter set, how many that did not fit, and finally how many that are still missing. When all parameter sets in a command have been checked and have their score calculated we can calculate a final score for the command (Line 12). This score will use only the best possible parameter set. After every command have been checked the algorithm will return the command with the highest score (Line 13), meaning the best possible command.

Appendix H

Implementation of Script Parsing and Interpretation

In this appendix, we will describe how our parser and interpreter for the scripting language have been implemented on top of our already existing command line parser and interpreter previous experiments.

Parser

In Listing H.1 and Listing H.2, we have shown how the parser have been implemented. We first split the script content into lines and loops through each line. For each line, we call a parser method for a line, the method is shown in Listing H.2. This method simply checks for the first word if it matches a known kind of word. E.g. if a `for` has been found, we call a parser method for the for-loop. The method will check if the syntax of the for-loop is correct and then return a for-loop token. The same principal is used for the if-condition.

```
1  public List<TokenObject> ParseScript(string script)
2  {
3    [...]
4    for (int lineIndex = 0; lineIndex < scriptLines.Length; lineIndex++)
5    {
6      string scriptLine = scriptLines[lineIndex];
7      if (scriptLine.Length > 0)
8      {
9        TokenObject token = this.ParseScriptLine(scriptLine);
10       if (token != null)
11       {
12         parsedLines.Add(token);
13       }
14       else
15       {
16         MessageBox.Show("Error at line " + lineIndex.ToString());
17       }
18     }
19   }
20   return parsedLines;
21 }
```

Listing H.1: The script content is passed to this script parsing method which splits the content into lines and passes them to the line parsing method.

If there is no match for a for-loop or if-condition, we say the script line could be a command. The script line is then passed further to our existing command line parser, thus, we call our script parser for the wrapper around the command line parser. However, if the command parser returns `null`, the script line may be an assignment instead. In this case, we call our assignment parser method.

If the parsing went well, we have a list of tokens which we give to the interpreter to construct loops, conditions, assignments and execute the right commands.

```
1  public TokenObject ParseScriptLine(string line)
2  {
3    [...]
4    string firstWord = this.GetFirstWord(line);
5
6    if (firstWord.Equals(Kind.ForLoop))
7    {
8      token = this.parseForLoop(line);
9    }
10   [...] // more else-if-conditions
11   else
12   {   // else, must be a command line
13     CommandStruct command = this.ParseLine(line);
14     if (command.keyword == null)
15     {
16       // must not have been a command after all, maybe an assignment
           then
17       token = this.parseAssignment(line);
18     }
19     else
20     {
21       token = new TokenCommand(command);
22     }
23   }
24   return token;
25 }
```

Listing H.2: For each line, we check the first word, if match any known kind, we call the corresponding parse method. If a command is found, we use the existing command line parser.

Interpreter

Our script interpreter is shown in Listing H.3. From the parser, the interpreter receives a list of tokens. These tokens will be interpreted depending on which kind it is. For-loop and if-condition are simply transformed into C# for-loop and if-condition, respectively, and commands are interpreted by the existing command line interpreter. All variables in the script exists in a C# `List`, and the interpretation of assignments will therefore just set the value of the correct item in the list with the `SetVariableValue()` method.

A for-loop will call `InterpretScript()` method recursively with the same starting index, thereby executing the body of the for-loop as many times as it should. An end-token will stop the method which means that every recursive call will automaticly return when they reach the end of the for-loop body.

```
public int InterpretScript(List<TokenObject> parsedLines, DrawArea
    drawArea, MainForm owner, int indexRoot)
{
  int tokenIndex = 0; // go through the token tree and interpret them
  for (tokenIndex = indexRoot; tokenIndex < parsedLines.Count &&
      tokenIndex >= 0; tokenIndex++)
  {
    TokenObject token = parsedLines[tokenIndex];
    if (token.GetType() == typeof(TokenLoop))
    {
      string var = ((TokenLoop)token).Var;
      TypeExpression from = ((TokenLoop)token).From;
      TypeExpression to = ((TokenLoop)token).To;
      int fromValue = from.Interpret();
      int toValue = to.Interpret();
      int returnedTokenIndex = 0;
      for (int _counterVariable = fromValue; _counterVariable <=
          toValue; _counterVariable++)
      {
        this.SetVariableValue(var, _counterVariable);

        // do other tokens
        returnedTokenIndex = this.InterpretScript(parsedLines,
            drawArea, owner, tokenIndex + 1);
        if (returnedTokenIndex < 0)
        {
          return returnedTokenIndex;
        }
      }
      tokenIndex = returnedTokenIndex;
    }
    [...] // more else-if-conditions
    else if (token.GetType() == typeof(TokenCommand))
    {
      // get the command struct or is it just the token?
      if (this.InterpretLine(((TokenCommand)token).Command, drawArea,
          owner))
      {
        // all good
      }
      else
      {
        // ERROR: Interpretation failed!
        return -(tokenIndex + 1);
      }
    }
  }
  return tokenIndex;
}
```

Listing H.3: A list of tokens is passed to the interpreter which construct the corresponding to C# structures and then execute commands with the existing command line interpreter.

Appendix I

Scripts for Repetitive Task Examples

These are the scripts for solving most of the repetitive tasks listed in Section 3.1.1.

Drawing the same shape many times

E.g. a 360 lines in a circle or many rectangles inside each other.

```
for x=1 to 36 do
  line start=140,140 length=4*x angle=10*x
end
```

Listing I.1: This script draws 36 lines with increasing length and at a clockwise increasing angle.

Drawing a gradient

Fading from one color to another color.

```
for x=0 to 255 do
  line s=10,10+x l=100 color=x,x,x
end
```

Listing I.2: This scrip will fade from black to white.

Drawing the same thing several times with different colors

E.g. switching between red and green colored lines.

```
for i=0 to 10 do
  y = i*4 + 10
  line s=10,y l=100 color=red pen=2
  line s=10,y+2 l=100 color=green pen=2
end
```

Listing I.3: This script will draw 20 lines which changes between red and green colors.

Drawing a specific figure several times.

E.g. a star, working with a script that can draw this figure would help.

```
1 line s=  5, 75 e=195, 75
2 line s=195, 75 e= 42,184
3 line s= 42,184 e=100,  5
4 line s=100,  5 e=158,184
5 line s=158,184 e=  5, 75
```

Listing I.4: This script will draw a star. This can then be used in a e.g. a loop to draw several of them, see Listing I.7.

Scale everything or part of a drawing.

E.g. make it smaller so it fits inside something else. Will require more keywords and functionality. scale (scale is partly implemented as size)

Drawing a chessboard.

Or grid.

```
1  cellsize = 10
2  cells = 20
3  for x=0 to cells do
4    if x=cells/2 then
5      line s=x*cellsize+10,10 l=cells*cellsize a=90 pen=1 color=red
6    else
7      line s=x*cellsize+10,10 l=cells*cellsize a=90 pen=1 color=black
8    end
9  end
10 for y=0 to cells do
11   if y=cells/2 then
12     line s=10,y*cellsize+10 l=cells*cellsize a=0 pen=1 color=red
13   else
14     line s=10,y*cellsize+10 l=cells*cellsize a=0 pen=1 color=black
15   end
16 end
```

Listing I.5: This script will draw a grid using vertical and horizontal lines. The center line will be drawn in red while the rest will be drawn in black. It is possible to change the amount of cells and the size of these by changing the assignments at the top.

Adding a shadow.

Or drawing the exact same thing in a different color just moved a bit. Will require more keywords and extra functionality. copy and changecolor

Drawing very precise.

E.g. drawing tangents to a circle in a non-orthogonal way (not on the default axis). Will require more keywords and extra functionality. line tangent to selected circle at angle

Random locations of several objects.

E.g. drawing a night sky with stars at random locations in random sizes.

```
boxes = 20
pos = 400
maxsize = 100
for i=1 to boxes do
  r s=random pos,random pos w=10+random maxsize h=10+random maxsize
end
```

Listing I.6: This script will draw rectangles of random sizes at random positions.

```
for i=1 to 10 do
  x=random 700
  y=random 350
  line s=x+5,y+75    e=x+195,y+75
  line s=x+195,y+75  e=x+42,y+184
  line s=x+42,y+184  e=x+100,y+5
  line s=x+100,y+5   e=x+158,y+184
  line s=x+158,y+184 e=x+5,y+75
end
```

Listing I.7: This script will draw several stars at random positions. The star is the one from Listing I.4, except that the random x and y have been added.

Appendix J

CD-ROM

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

- This report in PDF format.
- Figures from this report in their original format.
- Source code and binaries for:
  - DrawTools (original)
  - Experiment 1: Disclosing
  - Experiment 2: Sloppy Command Line
  - Experiment 3: Keyword Arguments
  - Experiment 4: Scripting Language