Blaze – Automating User Interaction in Graphical User Interfaces

Gabriel José Faustino Fonseca Barata

Dissertation for the Master’s degree in

Information Systems and Computer Engineering

Jury

Chairman: Doctor Joaquim Armando Pires Jorge
Adviser: Doctor Daniel Jorge Viegas Gonçalves
Members: Doctor Carlos Alberto Pacheco dos Anjos Duarte

November 2009
From the bottom of my heart, I would like to thank my parents, Ramiro and Maria do Rosário, for being so supportive and always being there for me. Thank you so very much for always encouraging me to go one step further.

A special thanks to my uncle José and my cousin Sofia, for being supportive and for always worrying about me and my future.

My most heartfelt thanks to Alexandre Ramos, Filipe Larsson, Paulo Carrilho and Sérgio Mendeiros, for being like brothers to me and even sacrifice yourselves for me and my work.

Thanks a lot to Daniela Loio for being there for me and supporting me in my long and tortuous testing sessions.

Thanks to Ana Rodrigues, Magda Carvalho and Sandra Gama, for being good friends and believing in my effort.

Thanks Tiago Guerreiro for being such a good co-adviser and supporting me in this research.

I am especially grateful to my adviser, Daniel Gonçalves, for advising and guiding me through this tortuous path that now comes to an end.

To all of these people aforementioned I am grateful for always believing in me.

Thanks to all users that accepted to collaborate with me and perform user tests. Each testing session lasted for two hours and I know it was quite a sacrifice. Thanks a lot for your time.

At last, but no least, I would like to thank the whole community for receiving Blaze so well and for the wonderful reviews. I hope we can work together in order to keep Blaze getting better and better.
RESUMO

Os utilizadores dos sistemas operativos actuais são constantemente confrontados com tarefas repetitivas, que surgem do uso diário. Apesar de actualmente existirem vários application launchers que tentam automatizar várias tarefas bem conhecidas, estes não estão à altura da unicidade das tarefas repetitivas que surgem do uso corrente. Muitas vezes, para tentarem maximizar o seu desempenho, os utilizadores tentam soluções alternativas, como o uso de linguagens de scripting ou mesmo editores de macros. Contudo, este tipo de solução está fora do alcance do utilizador comum, obrigando-os a completar a tarefa à mão.

Apresentamos um sistema inovador, denominado Blaze, que é capaz de automatizar as tarefas repetitivas do utilizador, não só numa única aplicação, mas ao nível do sistema operativo. A nossa abordagem baseia-se na monitorização contínua e passiva das actividades do utilizador e no estabelecimento de relações entre as acções capturadas. Um algoritmo de procura por prefixos comuns em árvores de sufixos é utilizado para detectar padrões repetitivos no historial de acções do utilizador, sem interromper o utilizador. Consequentemente, os padrões repetitivos detectados, em conjunto com as relações estabelecidas, permitem inferir o comportamento futuro do utilizador e compor acções que completem o resto da tarefa repetitiva. Estas acções podem ser guardadas em scripts que têm a capacidade de se adaptarem a novas situações, diferentes daquelas em que as acções foram inicialmente detectadas. Para além disso, o Blaze leva o conceito de application launcher para um novo nível, introduzindo um algoritmo de predição textual que consegue tolerar erros de escrita. Os testes efectuados mostram que o Blaze permite ao utilizador efectuar tarefas repetitivas 3 vezes mais rápido sem sacrificar significativamente o desempenho do computador.

PALAVRAS-CHAVE

Interfaces Adaptativas, Programação Implícita por Exemplo, Detecção de Padrões Repetitivos, Automatização de Tarefas, Sensibilidade ao Contexto do Utilizador.
ABSTRACT

Every day, operating system users are faced with repetitive tasks that should be automated. Although there are several application launchers currently available that can automate well known repetitive tasks, most of those tasks are unique and arise from everyday usage, making them unpredictable. In a situation where users face a repetitive task, they have only a few choices: if they are experienced users, they’ll probably be able to build a script to automate the task; if not, they will need to complete the remainder of the task by themselves or try to find a special-purpose application to do it and spend as much time learning how to use it as they would spend finishing it by hand.

We present Blaze, a system capable of automating user’s repetitive tasks in graphical user interfaces, operating system-wide. Blaze consists of an application launcher, which enhances its counterparts’ features, and is continuously monitoring and analyzing the user’s activities, without disturbing him. Thereby, Blaze identifies relationships and repetitions in the users’ actions and offers to accomplish the remaining of the task, in their place. To detect repetitions, Blaze resorts to a fast string search algorithm, based on common prefix search in a suffix-tree. Furthermore, Blaze is able to abstract a suggested automation and export it to a general-purpose script, which can later be replayed, even in a different situation from the one in which the repetition was detected. Our tests show that users can perform recurrent tasks 3 times faster using Blaze, without significantly sacrificing the computer’s performance.

KEYWORDS

Programming by Example, Adaptive Interfaces, Repetitive Pattern Detection, End-User Programming, User Context Awareness, Task Automation.
TABLE OF CONTENTS

1 Introduction .................................................................................................................. 1
  1.1 Contributions ........................................................................................................ 3
  1.2 Publications .......................................................................................................... 3
  1.2.1 Conferences .................................................................................................... 3
  1.2.2 Dissemination .................................................................................................. 4
  1.3 Document Structure ............................................................................................ 4

2 Related Work .............................................................................................................. 5
  2.1 Application Launcher Analysis ............................................................................. 5
    2.1.1 ActiAid ........................................................................................................... 5
    2.1.2 CAL ............................................................................................................... 6
    2.1.3 Colibri .......................................................................................................... 6
    2.1.4 Dash Command ........................................................................................... 6
    2.1.5 Enso Launcher ............................................................................................. 7
    2.1.6 Executor ....................................................................................................... 7
    2.1.7 Find and Run Robot ..................................................................................... 8
    2.1.8 Google Desktop ............................................................................................ 8
    2.1.9 Keybreeze .................................................................................................... 9
    2.1.10 Launchy ....................................................................................................... 9
    2.1.11 Shuriken ...................................................................................................... 10
    2.1.12 Skylight ...................................................................................................... 10
    2.1.13 Speed Launch ............................................................................................ 11
    2.1.14 WinSilver ................................................................................................... 11
    2.1.15 Gkrun ......................................................................................................... 12
    2.1.16 GNOME Do ............................................................................................... 13
    2.1.17 GNOME Launch Box ................................................................................. 13
    2.1.18 Katapult ...................................................................................................... 13
    2.1.19 KRunner ..................................................................................................... 14
    2.1.20 LaunchBar ................................................................................................... 14
    2.1.21 Butler .......................................................................................................... 14
    2.1.22 Quicksilver .................................................................................................. 15
    2.1.23 Todos .......................................................................................................... 15
    2.1.24 Ubiquity ...................................................................................................... 15

2.2 Application Launcher Evaluation and Discussion ................................................. 16
    2.2.1 Basic Application Launcher Features ......................................................... 17

IV
3.3 Blaze from an Architectural Point of View ........................................ 56
  3.3.1 System Abstraction ........................................................................ 56
  3.3.2 User Context .................................................................................. 56
  3.3.3 Indexer .......................................................................................... 57
  3.3.4 Scripting Engine ............................................................................ 57
  3.3.5 Interpreter ...................................................................................... 58
  3.3.6 Interface ......................................................................................... 58
4 Automating the User Experience .......................................................... 59
  4.1 The Observer ..................................................................................... 59
    4.1.1 Monitoring User Activity ................................................................. 59
    4.1.2 Generalizing ................................................................................ 62
  4.2 The Apprentice ................................................................................... 65
    4.2.1 Learning What to Automate ............................................................ 65
    4.2.2 Supported Repetition Types ............................................................ 67
  4.3 The Assistant ..................................................................................... 67
    4.3.1 Learning When to Suggest ............................................................... 67
    4.3.2 Exploring Alternatives ................................................................. 71
4.4 Major Problems ............................................................................................................. 72
5 Evaluation and Results .................................................................................................. 73
  5.1 User Tests ............................................................................................................... 73
    5.1.1 Procedure ......................................................................................................... 73
    5.1.2 Results ............................................................................................................ 74
    5.1.3 User Satisfaction ............................................................................................. 78
  5.2 Performance Tests .................................................................................................... 79
    5.2.1 Intensive Use Test ............................................................................................ 79
    5.2.2 Idle Test .......................................................................................................... 80
  5.3 Discussion ................................................................................................................ 81
6 Conclusions and Future Work ...................................................................................... 82
7 Bibliography .................................................................................................................. 84
Appendix A – User Test Guide .......................................................................................... 87
Appendix B – Blaze Reviews ............................................................................................. 94
LIST OF FIGURES

Figure 1 – a’ctivAid........................................................................................................5
Figure 2 – CAI................................................................................................................6
Figure 3 – Colibri.......................................................................................................... 6
Figure 4 – Dash Command............................................................................................ 6
Figure 5 – Enso Launcher............................................................................................ 7
Figure 6 – Executor.................................................................................................... 7
Figure 7 – Find and Run Robot 2 ............................................................................. 8
Figure 8 – Google Quick Search Box.......................................................................... 9
Figure 9 – Keybreeze................................................................................................. 9
Figure 10 – Launchy.................................................................................................... 9
Figure 11 – Shuriken..................................................................................................10
Figure 12 – Skylight...................................................................................................11
Figure 13 – Speed Launch – Draging an Item to the Bull’s Eye.................................11
Figure 14 – Speed Launch – Picking a Name to the New Command.........................11
Figure 15 – WinSilver...............................................................................................12
Figure 16 – Dmenu....................................................................................................12
Figure 17 – Gmrun....................................................................................................12
Figure 18 – GNOME Do...........................................................................................13
Figure 19 – GNOME Launch Box............................................................................13
Figure 20 – Katapult.................................................................................................13
Figure 21 – KRunner.................................................................................................14
Figure 22 – Butler.....................................................................................................14
Figure 23 – LaunchBar..............................................................................................14
Figure 24 – Quicksilver............................................................................................15
Figure 25 – Todos.....................................................................................................15
Figure 26 – Ubiquity.................................................................................................16
Figure 27 – a’ctivAid’s Renaming Extension............................................................21
Figure 28 – Bulk Rename Utility............................................................................22
Figure 29 – Blaze Interface......................................................................................39
Figure 30 – Blaze Performing a Calculation.............................................................40
Figure 31 – Blaze Browsing the File-System.............................................................40
Figure 32 – Blaze Tolerating Token Swapping.........................................................42
Figure 33 – Blaze Tolerating Typos........................................................................42
Figure 34 – String Contention Test Diagram...........................................................43
Figure 35 – Similarity Test Diagram.......................................................................44
Figure 36 – Item’s Name Token Weighting...............................................................45
Figure 37 – Adding a New Command to Blaze.........................................................47
Figure 38 – Adding a New Website to Blaze.............................................................47
Figure 39 – Adding a New Quick Text to Blaze.........................................................48
Figure 40 – Creating an Email with Blaze.................................................................49
Figure 41 – Blaze’s Task Bar Icon Before and After a Repetition was Detected........51
Figure 42 – Blaze Presenting an Information Icon on the Top Right Corner............51
Figure 43 – Blaze Assistant......................................................................................52
Figure 44 – Blaze Automating a Recurrent Task on Notepad.................................53
Figure 45 – An Excel Book, Targeted for a Repetitive Task.......................................54
Figure 46 – Blaze Automating a Recurrent Task On Excel.......................................54
Figure 47 – Photos With Strange Names..................................................................55
Figure 48 – Blaze Automating a File-System Recurrent Task..................................55
Figure 49 – Blaze’s Layer Architecture............................................................................56
Figure 50 – ContextLib’s Basic Architecture....................................................................57
Figure 51 – Communication Procedure Between Blaze Assistant and the Automation Agents ...58
Figure 52 – The Suffix-tree Representing “abcabd”..............................................................66
Figure 53 – First Version of Blaze Assistant.......................................................................68
Figure 54 – Blaze Assistant Revised..................................................................................68
Figure 55 – Blaze Assistant Showing Color Text Formatting..............................................69
Figure 56 – Narrative Example #1......................................................................................70
Figure 57 – Narrative Example #2......................................................................................71
Figure 58 – Blaze’s Automation Editor................................................................................72
Figure 59 – Overall Results from User Tests......................................................................75
Figure 60 – Narrative Containing an Extra Token...............................................................76
Table 1 – Basic Application Launcher Features Comparison .................................................. 18
Table 2 – Integration and Customization Features Comparison ........................................... 20
Table 3 – Automation and Context Awareness Features Comparison ................................. 23
Table 4 – Basic Application Launcher Features Suitability ................................................. 25
Table 5 – Integration and Customization Features Suitability ........................................... 25
Table 6 – Automation and Context Awareness Features Suitability .................................. 25
Table 7 – Automation Basis Features Comparison ............................................................. 34
Table 8 – Direct Interaction Features Comparison ........................................................... 35
Table 9 – Automation Basis Features Suitability ............................................................... 37
Table 10 – Direct Interaction Features Suitability ............................................................ 37
Table 11 – User Action Compression Algorithm Example .................................................. 61
Table 12 – List of Mouse Actions Supported by the Observer ............................................ 61
Table 13 – List of Keyboard Actions Supported by the Observer ......................................... 61
Table 14 – List of File-System Actions Supported by the Fyle-System Watcher ..................... 61
Table 15 – List of Generalizations Created by the Observer .................................................. 63
Table 16 – List of Functions Supported by the Observer ...................................................... 64
Table 17 – Function Priority List (6=Highest Priority) ....................................................... 65
Table 18 – Results from Tests Regarding File-System Repetitions ....................................... 75
Table 19 – Results from Tests Regarding Application Repetitions ...................................... 76
Table 20 – Results from Tests Regarding Typical Application Launcher Features .................. 77
Table 21 – Results from Tests Regarding Advanced Automation Tasks ............................... 78
Table 22 – User Rating of Blaze’s Features ........................................................................ 78
Table 23 – User Rating of Blaze’s Interface ........................................................................ 78
Table 24 – Test results From Intensive Use Test ................................................................... 80
Table 25 – Test results From Idle Use Test .......................................................................... 81
1 INTRODUCTION

These days, operating systems have been growing strong, teeming with exciting features and appealing user interfaces. These, in turn, provide several interactive aids, with the intent of helping the user to face most of the common repetitive tasks that arise from everyday usage. As an example of this, think of Microsoft Windows' Start Menu or even Apple Mac OS's Dock, which aim to provide faster ways to access and launch applications that are often used by the user. However, operating systems’ interfaces have not always offered this sort of mechanisms. As a matter of fact, these interfaces have been subject of a never-ceasing evolution, with the goal of quickly launching most used applications and preventing the user from having to remember any commands at all.

This fierce evolution has, however, begotten interfaces that have lost the expressive power once granted by command line interfaces. Whilst requiring the user to memorize valid sets of commands, those interfaces offered the user a more precise control over their files and applications, by providing mechanisms such as wildcards and the ability to affect many files and applications, at once, with a single command. Furthermore, command line interfaces often offer scripting capabilities, providing the user with the means to automate repetitive tasks. Examples of well known command line interfaces are the Microsoft Windows Command Prompt and the Bash shell, appearing on Mac OS X and most systems built on top of the Linux kernel. Both allow the user to build scripts – batch files in the former case and bash scripts in the latter – which can perform a set of repetitive tasks, over any file or application.

Nowadays, scripts are still often used, especially to automate unique repetitive tasks that might arise from daily interaction with a computer. There are several examples of such tasks which demonstrate where current automation mechanisms seem to be failing.

Imagine that you went on vacations, and you took around fifty photos with your digital camera. As you come back, it is time to download all photos from the camera to the PC. After doing this, you notice that all downloaded photos have strange names, given by the digital camera, such as “SDC10050.JPG”, “SDC10051.JPG”, “SDC10052.JPG”, etc. Wanting to have more meaningful names, such as “Summer Vacation 01.jpg”, “Summer Vacation 02.jpg”, and so on, it is clear you must rename them. Depending on your skill level, regarding the operative system you use, you may have different options to face the problem. If you are inexperienced, your only options are to rename the photos, one by one, or try to download a renaming too tool from the Internet, and spend as much time learning how to work with it as you would spend renaming the photos by hand. However, if you are proficient with your operative system and familiar with programming languages, you could just do a script to accomplish the task, by iterating through all JPG files in that folder and renaming them. Therefore, you need some programming skills and knowledge to perform this task, on an acceptable amount of time.

As another example, imagine that you want to organize the folder where you store your Instant Messaging received files. This kind of folder tends to be messy, as it receives files of all kinds, with all sorts of strange names and extensions. You noticed that you had received a lot of files from Mary Jane, as she is really narcissistic and always inserts her name somewhere in the filename. You also noticed that those files have different extensions. Thus, you would like to move all files whose name contains the tokens “Mary” and “Jane” to the subfolder “Mary Jane”, which you just created. Yet again, for this problem to be solved on an acceptable amount of time, it would require you to build a script. However, this time, the problem forces you to do some
string parsing, which would make you spend as much time building the script as you would spend moving tens of files to the subfolder, one by one.

Leaving file system operations behind, there are several other interesting situations where the user repeats the same actions, over and over again. For instance, when an user inserts a table on a text processor, always repeating the same button pressing sequence, or even when he inserts a few sequential text tokens, like "Note #1:", "Note #2:", "Note #3:", and so on, where, clearly, there is a numerical pattern that could easily be automated. To solve the former case, one can use a macro recorder, which does not require much knowledge about the operating system itself, but may still have a steep learning curve. To solve the latter case, an advanced scripting language could be used, such as AutoIt v31. Nonetheless, this solution requires a lot of knowledge about Basic-like programming languages and about how the operating system communicates with processes. This is, thus, beyond the reach of most common users. Most common users do not have any advanced knowledge about programming languages and do not really understand how the operative system works. They only know how to use it.

The above examples clearly demonstrate that, despite the whole evolution that the current operative systems’ graphical user interfaces have gone through, they do not offer any new support mechanism to automate those repetitive tasks that might arise from everyday usage. These are quite a nuisance to the users and, actually, force them to resort to scripting languages and to former interfaces, such as the command line. However, as this sort of knowledge is beyond the reach of most common computer users, they have to handle the repetitive tasks themselves, performing the required operations by hand.

To solve this problem, we developed an innovative application, named Blaze2, which is able to automate repetitive user tasks on Microsoft Windows. Blaze is not just able to automate common tasks to all users but also those unique repetitive tasks that emerge from daily routine. This, per se, represents a successful fusion between two automation realms, which had never been completely reached before.

Blaze took the form of an application launcher as it gives back to the user the expressive power once granted by the command line. The typical application launcher interface provides an easy and fast way to launch a specific function, based on a portion of the function name, recalled by the user. This way, not only the user is exempt from having to navigate through dozens of menu entries, but also, he only needs to remember part of the function’s name. Furthermore, Blaze enhances this feature by fully tolerating typos, which is also a feature never before implemented by another application launcher. Thus, not only Blaze allows regular users to easily automate common repetitive tasks, but it also allows more proficient users to take full advantage of a command line like interface.

Besides automating essential tasks, such as application launching, calculations and web searches, common application launchers are not aware of the user context. We define user context awareness as the ability to monitor the user behavior, while he uses the operative system, and capture valuable information about what he is doing in a precise moment. As an example of this kind of information, we have the URL of the website or the file system path that the user is currently browsing, what files he has currently selected, what is the actual screen resolution or what wireless networks is he connected to. Accordingly, Blaze features several mechanisms, which provide access to the above cited information and a lot more. This

1 http://www.autoitscript.com/autoit3/ (Last visited in: June 1st, 2009)
2 Available online at http://blaze-wins.sourceforge.net/
information allows Blaze to take the application launcher concept to a whole new level and enhance its counterparts’ features, by automatically customizing services according to the user’s context and by allowing him to also reuse that information.

Notwithstanding, Blaze presents Blaze Assistant, which is a narrative based interface, that is able to propose the user automations capable of completing repetitive tasks. This proposal is non intrusive as it does not interrupt the user’s work nor does it distract him in any way possible.

In order to detect that users are performing repetitive tasks, Blaze resorts to their context to monitor their activity and detect repetitive patterns, using a well known and efficient repetition detection algorithm and a powerful new generalization system. Generalizations are built between similar user actions and describe specific relationships among them, which allow Blaze to infer future actions and, thus, offer to complete repetitive tasks in the user’s place. Blaze is able to detect repetitive tasks over the whole operating system, covering not only the file system but also any running application. This is something that was not successfully attained before, as other Programming by Example systems only cover a single application.

Blaze also features advanced scripting capabilities, which have two main goals: a) allow the user to save detected repetitions, which can later be reproduced and edited; b) allow advanced users to, easily, extend Blaze’s functionalities, without having to compile any code, just by using a regular text editor. This consists of another innovative feature, which is not presented by Blaze’s counterparts.

1.1 CONTRIBUTIONS

We developed a new approach to automate repetitive tasks, without user intervention, on a modern operating system. Our research yielded the following contributions:

- A textual prediction mechanism capable of tolerating typos and taking into account the metadata of an item, which can carry semantic meaning.
- An efficient algorithm capable of detecting repetitive user tasks, operating system widely, without requiring his intervention.
- A mechanism capable of composing sequences of actions to automate repetitive tasks, which can be parameterized by the user.
- A mechanism to generate comprehensive narratives describing a set of actions which can complete a repetitive task. These narratives cover a wide range of actions ranging from operations over the file-system to operations regarding the mouse and the keyboard.
- An advanced mechanism for user contextual information retrieval.
- An application, Blaze, which can be used in real-life settings, which embodies the contributions described above.

1.2 PUBLICATIONS

The list of all publications regarding our research follows bellow:

1.2.1 Conferences

By the time this dissertation was written, we published one paper and submitted two other, currently undergoing the peer-review process:
1.2.2 Dissemination

Blaze was released to the public in August 13\textsuperscript{th} of 2009, as an open source project at SourceForge\textsuperscript{3}. A website was created to support the project, which features a brief description of the project, a quick guide that explains how to use Blaze, and proper documentation about Blaze’s commands, hotkeys and main features.

On August 28\textsuperscript{th}, Blaze received its first review, by the website lifehacker.com\textsuperscript{4}: Blaze Launches Applications and Automates Repetitive Tasks\textsuperscript{5}. Lifehacker is a renowned technological website, with thousands of daily readers. The reviewer was quite impressed with Blaze’s capabilities. A more detailed description about this and other reviews can be found on Appendix A.

The review was excellent and, just in the day it was published, Blaze was downloaded 991 times. From August 13\textsuperscript{th} to September 2\textsuperscript{nd}, Blaze was downloaded more than 2000 times. As of this moment (September 27\textsuperscript{th}), Blaze has been downloaded over 6000 times.

1.3 DOCUMENT STRUCTURE

To support our research, we first have to analyze and evaluate all existing solutions that, somehow, contribute to solve the problem. Therefore, a survey of the state-of-the-art is described in Chapter 2, allowing us to identify the shortcomings of those solutions and build solid foundations to support Blaze’s development. In Chapter 3 we will describe our solution, offering an overview over Blaze’s approach for automating recurrent user tasks, and giving a detailed explanation of the whole system architecture. A more detailed description of how Blaze monitors the user and detects repetitions will be shown in Chapter 4, followed by an explanation of all performed tests and the respective results, in Chapter 5. After demonstrating that Blaze is, indeed, suitable to solve the proposed problem, we will discuss the major conclusions of this research, pointing out its pros and cons, and possible future work.

\textsuperscript{3} http://sourceforge.net/ (Last visited in: September 14\textsuperscript{th}, 2009)
\textsuperscript{4} http://lifehacker.com/ (Last visited in: September 14\textsuperscript{th}, 2009)
\textsuperscript{5} http://lifehacker.com/5347367/blaze-launches-applications-and-automates-repetitive-tasks (Last visited in: September 14\textsuperscript{th}, 2009)
2 RELATED WORK

Despite of all evolution that operative system’s graphical user interfaces have gone through, it is clear that a lot of expressive power has been lost, formerly granted by command line interfaces. This loss created a gap on nowadays interfaces, which have been filled by applications launchers. If you, as a user, just want to launch the calculator, why do you have to browse through several menu entries to reach it? With an application launcher installed on your system, you just need to invoke it, type “calc” and that’s it! The calculator is instantly launched. This lack of expressive power is so notable, that recent commercial operative systems include their own application launching systems, such as the Mac OS’s Spotlight and the Windows Vista start menu search. Although these application launchers may speed up user activity, they lack of significant automation tools. There are, however, a few prototypal systems that address this issue, worthy of being mentioned.

Therefore, a careful review must be paid to all current application launchers, with the intent of identifying which aspects should be improved, in order to provide users with better automation mechanisms. As there are, already, a few works that try to automate user repetitive tasks, these will also be carefully analyzed, so as to identify their shortcomings and establish a solid base to support Blaze’s development.

2.1 APPLICATION LAUNCHER ANALYSIS

Currently, there are several application launchers, available for almost all operating systems. An application launcher is an application characterized by accepting user input, in form of a string, and propose to launch an item, whose name is, somehow, related with the provided text. These items may be applications, files, folder or websites. Moreover, most application launchers share a set of common features, such as on-the-fly application launching, numerical calculations, web searching, web browsing, file system browsing, command and path auto-completion and recently accessed file browsing.

However, there a few application launchers that stand out from amongst the others. The spotlight points to those that, somehow, offer advanced automation features. These are mainly attained through macro editing features, operating system integration and manipulation capabilities, and interactive command learning.

2.1.1 Ac’tivAid

Ac’tivAid is an interesting application launcher, worthy of being mentioned due to its accessibility and automations features. However, ac’tivAid is not really that great as an application launcher, per se. The user interface is not visually apppellative at all and most of its features are used in separate windows and invoked by distinct hotkeys, which is, simply, hard to use. Despite offering most of the common features included by its counterparts, some of them are just badly implemented. For example, the file system browsing window does not give any useful tip about

\[ http://www.heise.de/ct/activaid/ (Last visited in: June 20th, 2009) \]
which folder the user is browsing nor offer an auto-completion mechanism, for names or paths. Nonetheless, this application offers some interesting features regarding accessibility, such as a ruler and a magnifying glass. Furthermore, act’ivAid provides some Microsoft Windows customization features, which allow the user to change windows’ dimension and opacity level, get additional hotkeys, have a new Task Switcher, and so on. Another remarkable feature is the ability to have multiple clipboards and, easily, manage all of them.

With respect to repetitive task automation, act’ivAid stands out from amongst the others, by being able to perform Cron Jobs, replace abbreviations and rename files and folders. This latter feature requires some parameterization, which makes it rather hard to use.

2.1.2 CAL

CAL\(^7\) is typical application launcher, with typical features. It launches applications, browses the web and the file system, supports custom commands and also features web search. The web search feature, however, requires a modifier key, such as the Alt, Ctrl or Shift, to be pressed in order to choose what search engine will be used. This, indeed, plagues the interface, forcing the user to remember which modifier key matches a specific search engine. CAL presents a simple and appropriate interface and does not introduce any new exciting features. It is, however, flawed with a bug that, occasionally, prevents the text field from getting focus, which troubles the usage of this application.

2.1.3 Colibri

Colibri\(^8\) is similar to CAL, in terms of functionality. Although having a more visually appealing user interface and a better URL recognition than CAL, Colibri loses in terms of file system browsing; not offering an auto-completion mechanism for paths consists of a substantial fault. Colibri also lacks of a web search feature and proved to be hard to configure, due to the lack of mouse support in the configuration panel.

2.1.4 Dash Command

Dash Command\(^9\) is, without any doubt, the application launcher with best balance between visual appeal and functionality. It does not just include most of the features from this type of application, but it also implements them very well. Besides indexing applications, performing web search and file system browsing, and also indexing recently accessed files, Dash Command

---

7 http://cal.sentaxsolutions.com/ (Last visited in: June 20\(^{th}\), 2009)
8 http://colibri.leetsspeak.org/ (Last visited in: June 20\(^{th}\), 2009)
9 http://www.trydash.com/ (Last visited in: June 20\(^{th}\), 2009)
presents a very well-planned integration with many applications for Microsoft Windows and with the operating system itself. It allows the user to browse the file system, preview files' contents and apply operations upon them. It also integrates very well with Google applications, such as Google Calendar and Google Maps, and allows users to browse their favorite songs and manipulate the clipboard. Furthermore, Dash Command can capture user's selected text and reuse it for web search proposes. Besides having the ability to learn new commands and being extensible, Dash Command lacks of some basic features, such as performing calculations or web browsing. The major drawback of this excellent application is to be commercial and require a payment of 19.95 USD to be fully functional.

### 2.1.5 Enso Launcher

Enso Launcher\(^9\) is an application launcher quite interesting. Besides indexing applications, performing calculations and performing web search, just like the others, Enso has the interesting ability to learn and unlearn commands in a very simple way. Furthermore, it is able to manipulate windows and windows’ tabs, which may become very handy whenever there is too many opened windows. With a few keystrokes you can easily switch between, minimize, maximize, open or close any window. However, it is a pity that Enso does not offer support for some basic features, such as file system and web browsing.

Enso has the interesting ability to detect the text currently selected by the user and allow operations to be performed upon that text. As an example, the user can select a math expression, on notepad, and type “calculate”, which will, immediately, solve the expression. This type of user context awareness is, indeed, very useful and may greatly enhance user experience. Nonetheless, the major drawback of this application is that Enso Commands tend to get tedious to use, as its usage is very verbose. For instance, to open an application the user needs to type “open” before an application’s name. Typing the application name only will result in an unknown command warning.

### 2.1.6 Executor

Executor\(^11\) is a very complete application launcher, as it presents a wide range of interesting features. Besides the typical features for this kind of application, Executor also presents the ability to explore both clipboard and command history. It can, also, manipulate and list all running applications, though the manipulation, per se, may become quite hard to use, as it is very verbose. Executor allows the user to add custom commands in a very flexible way. These commands can be bound to, not only applications, but also files and folders. Moreover, Executor integrates with Windows “sent-to” menu, to easily allow a selected item to be added as a new command.

This remarkable application also integrates with Windows Instant Search, on Windows Vista, and with Windows Desktop Search, in Windows XP, as means to extend the search to a new

---


\(^11\) [http://executor.dk/](http://executor.dk/) (Last visited in: June 20th, 2009)
level. Even though Executor offers a lot of features, most of them are only easily accessible to experienced users, as most of the associated commands are very verbose and configuration dialogs are quite complex.

### 2.1.7 Find and Run Robot

Find and Run Robot 2 (FARR)\(^\text{12}\) is an application launcher, somehow, interesting. It is rather similar to both CAL and Calibri, in terms of functionality, but with a more appealing user interface. Nevertheless, FARR is quite extendable, allowing the user to download and install plugins from the internet. These may range from simple calculators to translators and even interesting integrations with some of the Google applications. FARR is quite configurable, although this might not be easily accessible to a common user.

FARR also innovates by using heuristic scoring to rank found items. There is a wide range of heuristics employed, and advanced users can configure the differential weighting of various factors. However, the default settings may generate dubious results and changing this may be out of reach to a regular user. Take the example of the above screenshot into account. We typed "ac", hoping to get ac’tivAid, which was already installed in the computed. However, "ac’tivAid.lnk" came only in 8\(^{th}\) place, in the result list, being "Backup.lnk" the first one. As you can see, none of the first five results are related to what we were looking for, which is cumbersome. Although being configurable, this situation may be translated, to the user, as lack of reliability.

### 2.1.8 Google Desktop

Although arguable, some consider desktop search tools to be application launchers as well, given their ability to quickly find and launch indexed applications. As Google Desktop\(^\text{13}\) features the ability to focus on application launching, it was also considered for this reviewing.

As an application launcher, Google Desktop is poor. There are a lot of basic features missing, like performing calculations, integrating with media players, or file and folder manipulation. The search engine, by itself, proved to be not suitable for application launching, when compared to its counterparts. For example, typing “firefox” will lead to Mozilla Firefox but typing just “fox” will lead to no application, which is just wrong. Nonetheless, Google Desktop offers the ability to launch indexed applications, browse and search the web and look up for files and folders, just like the others. What is more, Google Desktop also integrates with outlook, to help searching email, which is, indeed, a very interesting feature.

---


\(^{13}\) [http://desktop.google.com/](http://desktop.google.com/) (Last visited in: June 21\(^{th}\), 2009)
2.1.9 Keybreeze

Keybreeze Personal Edition\(^{14}\) is a very good application launcher, featuring many of the Executor's functionalities, yet presenting a more intuitive and easy to use interface. Besides launching applications, performing web searches and file system and web browsing, Keybreeze does not feature an auto-completing system. This application integrates well with the default email client, with Skype and with Windows Media Player. It also features the ability to manipulate and customize windows, insert quick texts and create notes and memos. Another interesting aspect of this application launcher is that it shows a wizard to assist creating a new command, whenever a user types an unknown sequence of text.

With respect to repetitive task automation, Keybreeze allows the user to record macros, which consist of sequences of mouse gestures and clicks, along with keystrokes, that can later be reproduced. However, creating a macro is quite a complex task, therefore being out of reach of a common user.

2.1.10 Launchy

Launchy\(^{15}\) is, with no doubts, an example to follow, amongst the application launchers. It may not have as many features as Executor does, nor have the ability to interactively learn new commands, like Enso or Keybreeze do, but it includes the most common features from the application launcher realm, while exhibiting a simple and visually appealing user interface. Although being quite configurable, Launchy's configuration dialogs are very simple, making every supported task reachable to almost any user.

\(^{14}\) http://www.keybreeze.com/ (Last visited in: June 20\(^{th}\), 2009)

\(^{15}\) http://www.launchy.net/ (Last visited in: June 20\(^{th}\), 2009)
Not neglecting the power of simplicity of use and the broad range of offered features, make Launchy the application launcher of choice for thousands of users, throughout the entire world.

2.1.11 Shuriken

Shuriken\textsuperscript{16} is still being forged and, thus, does not present any new features, in addition to those already expected from this kind of application. It also lacks of some basic application launcher features, such as performing calculations, web search and file system and web browsing. Moreover, it does not offer much room for customization.

![FIGURE 11 – SHURIKEN](image)

2.1.12 Skylight

Skylight\textsuperscript{17} is a quite nice and aspiring application launcher, implementing most of its QuickSilver counterpart's features. Skylight presents the usual features, such as application launching, calculations, file system and web browsing, web searching, and so forth. What is more, this application launcher has the ability to learn abbreviations and reuse the currently selected text for searching proposes. It, also, integrates with other applications, such as media players, instant message clients and with Windows Desktop Search, allowing it to index files’ contents. Like QuickSilver, Silverlight has the ability to learn from user habits.

Although presenting a visually appealing interface and a lot of features, Skylight proved to be, sometimes, hard to use, denoting some room for improvement in the search engine and result exhibition.

\textsuperscript{16} http://shuriken.codeplex.com/ (Last visited in: June 20\textsuperscript{th}, 2009)

\textsuperscript{17} http://www.candylabs.com/skylight/ (Last visited in: June 20\textsuperscript{th}, 2009)
2.1.13 Speed Launch

Although not having many of the features typical to this kind of application, such as automatically indexing applications, performing calculations or file system and web browsing, Speed Launch\textsuperscript{18} is still an interesting tool, with a rewarding and fun to use learning system. Speed Launch has the ability to learn new commands, which can launch applications, websites, files and folders. For this, the user just needs to drag the desired item to the bull’s eye on the desktop, causing a form to show up, where the commands name may be specify. Moreover, this learning system allows the user to specify parameterized search queries in a simple way, which comes to be very handy while searching the web.

2.1.14 WinSilver

WinSilver\textsuperscript{19} is, like Skylight, another QuickSilver clone for Microsoft Windows. It presents similar features to those of Skylight, such as application indexing, file system and web browsing, recently accessed files and folders browsing, folder item manipulation, and so on. However, some of these features are not very well implemented. A few tests were carried out with the intent of testing those. For example, while trying to zip a folder, the compressed file was created empty. It is also clear that this application does not correctly weight indexed items. For instance, while looking for “firefox”, WinSilver first showed Mozilla Firefox’s bookmarks and only after that, the application’s shortcut. These flaws make WinSilver quite hard to use and trust. Furthermore, there are a few typical features missing and there is not much room for customization either.

\textsuperscript{18} http://www.officelabs.com/projects/speedlaunch/Pages/default.aspx (Last visited in: June 21\textsuperscript{th}, 2009)
\textsuperscript{19} http://www.winsilver.net/ (Last visited in: June 21\textsuperscript{th}, 2009)
2.1.15 Dmenu

Dmenu\textsuperscript{20} may be considered an application launcher, although it is more like a command manager, for Linux. Dmenu indexes all commands that can be launched from the command line and does not present any additional features. The interface is quite simple and not very appellative, consisting of a menu bar that just stands there, at one of the screen’s edge. Dmenu does not allow much customization either, allowing only the colors and the position on the screen to be modified.

2.1.16 Gmrun

Gmrun\textsuperscript{21} is a similar tool to dmenu, being just a surrogate for the terminal. Like the latter, it just indexes commands that can already be launched from the command line. Albeit gmrun can handle URLs, this feature must first be configured through editing of a file, by hand, which requires some domain knowledge and, therefore, is out of reach of most common users. Nonetheless, gmrun presents a few interesting improvements, when compared to dmenu. For instance, it presents an auto-completion mechanism, command history and a finer user interface.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig15.png}
\caption{WINSILVER}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig16.png}
\caption{DMENU}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig17.png}
\caption{GMRUN}
\end{figure}

\begin{flushleft}
\footnotesize
\textsuperscript{20} http://tools.suckless.org/dmenu (Last visited in: June 21\textsuperscript{th}, 2009)
\footnotesize
\end{flushleft}
2.1.17 GNOME Do

GNOME Do is one of the best application launchers currently available for Linux, being inspired in GNOME Launch Box, for Linux, and QuickSilver, for Mac OS. Besides presenting a very visually appealing user interface, GNOME Do features several functionalities that can be managed through plugins. These plugins are distributed with the application itself and can easily be activated, deactivated and configured by the user. Also, setting up GNOME Do is equally easy, therefore being reachable to most users.

GNOME Do includes 35 plugins and each one of them offer a specific functionality. For example, Do can index any file or folder, Mozilla Firefox’s bookmarks and Gmail contacts; look up for word definitions in a dictionary, take screenshots and launch applications; perform calculations and monetary conversions, through Google Calculator, and integrate with Google Calendar and Google Maps; search the web, manipulate windows and integrate with Pidgin Instant Messaging client. Although this huge amount of plugins provides a wide range of useful features, it also presents a few drawbacks. With so many plugins competing between them, GNOME Do has a hard time figuring out which plugin should assist the user’s input. Sometimes, an inappropriate plugin is chosen.

2.1.18 GNOME Launch Box

![GNOME LAUNCH BOX](image)

GNOME Launch Box is a simple application launcher, for Linux, which lacks of some basic features, such as performing calculations and browsing the file system and the web. In the other hand, Launch Box integrates quite well with Evolution e-mail client, allowing the user to easily access any of his/hers email contacts. It, also, indexes recently accessed files and any file contained in the desktop, as well as Mozilla Firefox bookmarks.

2.1.19 Katapult

Katapult is a similar application launcher to GNOME Do. However, while the latter is written to work with GNOME desktop environment, the former is written to work with the KDE one. Katapult offers the typical features, from its realm, such as application indexing, Mozilla Firefox bookmark indexing, integration with Amarok media player, perform calculations, spellchecking, web searching and file and folder browsing. Katapult is very easy to configure and offers some customizations options.

2.1.20 KRunner

KRunner\textsuperscript{25} is another application launcher, written to work with the KDE environment. Like Katapult, KRunner performs calculations, browses and searches the web and indexes applications and Firefox's bookmarks. Although it allows browsing the file system, an auto-completion system is not provided, which is a pity. Another major drawback is the fact that the results are only shown after the user types the third letter of the command.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig21.png}
\caption{KRUNNER}
\end{figure}

2.1.21 Butler

Butler\textsuperscript{26} is a quite complete application launcher, for Mac OS X, and can easily be compared to Executor and ac’tivAid, in terms of functionality. Butler launches applications, browses and searches the web, manages web browser's bookmarks, runs AppleScripts, offers additional clipboards and a finer clipboard management, inserts quick texts, integrates with iTunes, enhances Spotlight search, simulates keystrokes and manipulates files and folders.

Although offering several useful features, Butler may be, sometimes, chaotic to use. The simple fact that the web search and application launching windows are distinct from one another, make it hard to figure out if both features belong to the same application or not.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig22.png}
\caption{BUTLER}
\end{figure}

2.1.22 LaunchBar

LaunchBar\textsuperscript{27} is an efficient application launcher for Mac OS X, featuring most of the typical functionalities from this category. LaunchBar can index other applications and web browser's bookmarks; performs calculations and browses and searches the web; integrates with media players, email clients, and web browsers; manipulates files and folders; and indexes recently accesses folder items. This application launcher is, clearly, very complete and features a lot more functionalities. Although being quite configurable and presenting a visually appealing user interface, LaunchBar suffers from the same shortcoming as Dash Command: it is a commercial application.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig23.png}
\caption{LAUNCHBAR}
\end{figure}

\textsuperscript{25} http://do.davebsd.com/ (Last visited in: June 21\textsuperscript{th}, 2009)
\textsuperscript{26} http://www.manytricks.com/butler/ (Last visited in: June 21\textsuperscript{th}, 2009)
\textsuperscript{27} http://www.obdev.at/products/launchbar/index.html (Last visited in: June 21\textsuperscript{th}, 2009)
QuickSilver is, probably, the most well known application launcher for Mac OS X. Its success is due to the same reasons that justify Launchy’s success: it is simple, fast, good-looking and covers most important features. Of course the concept of “simplicity” is rather different from the one considered in Launchy’s case. Launchy is quite simple, when compared with its Microsoft Windows’ counterparts. QuickSilver, whilst being more complex than Launchy, is, for sure, simpler than its Mac OS X’s equivalents.

QuickSilver offers the most common features, in the application launcher category. Nevertheless, it can be easily extended through plugins, which can be automatically installed along with QuickSilver, in the installer wizard. The extensions allow this remarkable application to integrate with media players, email clients, web browsers, and so on. A noteworthy aspect of QuickSilver is the ability to manipulate files. This excellent feature, also present in some of its homologous rivals, allows the user to create, open, edit, rename and delete files on-the-fly. It is also interesting to point out that QuickSilver presents a satisfactory balance between configurability, customization and simplicity.

Todos29 is an interesting application launcher, with quite a different approach. Instead of allowing the user to search the desired applications, by typing in their names, Todos opts for displaying shortcuts with the respective icons. This approach is quite interesting, since it may be, sometimes, easier to remember the application’s icon instead of the name.

Although Todos has a very limited range of features and does not offer much room for customization or configuration, it presents the user with a very appellative user interface and simplicity of use.

Ubiquity30 is an extension for Mozilla Firefox that has the intent of connecting the web with a natural-language-like command interface, in an attempt to find new user interfaces that could make it possible, for everyone, to do common web tasks more quickly and easily. In other words, it represents a new, attractive and powerful way of interacting with the web.

Ubiquity works pretty much like an application launcher, but inside Firefox. The user presses a hotkey, a panel and a textbox show up, and then he just needs to type a command. This application extends Firefox functionality by offering means to automate well known recurrent tasks. For example, Ubiquity features web search, through several search engines, and has the

28 http://docs.blacktree.com/quicksilver/what_is_quicksilver (Last visited in: June 21th, 2009)
ability to send emails on the fly, by integrating with Gmail. This, in turn, allows Ubiquity to be aware of all users email contacts.

Another very interesting and useful feature of Ubiquity is its ability to be aware of what text the user has currently selected and perform operations upon that text. These operations may call upon commands that already integrate with other web applications. For instance, if the user has currently selected the word “Lisbon”, on a web page, and wants to know where it is, he just needs to type “map” on Ubiquity. Accordingly, Google Maps will open and center Lisbon city. This feature can also be used to, for example, perform calculations. Selecting “3+3” on an online form and typing “calc” will replace the expression by the result “6”. Furthermore, the selected text can be referenced by the keyword “this”, which can be used inside other commands.

There are, still, a lot more noteworthy features, that would be interesting to denote. Nonetheless, this extension is, indeed, a great advancement in the intelligent interfaces scope, which can really automate user experience.

![Figure 26 - Ubiquity](image)

---

### 2.2 APPLICATION LAUNCHER EVALUATION AND DISCUSSION

After reviewing 25 applications, a comparison was performed, to identify and clarify their shortcomings. 27 features were taken into account and divided into 3 categories: 1) Basic Application Launcher features, 2) Integration and Customization features, and 3) Automation and Context Awareness features. Each one of these categories will be explained in the next subsections. Each subsection will contain a table showing which features are implemented by a specific application. A checkmark (✓) denotes that a feature is met by the application and an x mark (✘) denotes otherwise.
2.2.1 Basic Application Launcher Features

Although not being of great relevance for automating recurring tasks, which emerge from everyday usage, there is a set of features that must be considered. Examples of these are file system and web browsing, performing calculations, web searching, or having an appellative user interface, which consist of the most common basic features, implemented by most application launchers. The following 16 items are considered to be Basic features:

- App Launching – Indexes and launches applications.
- Auto-completion – Features name and/or path auto-completion mechanism.
- Browse FS – Browses the file system.
- Browse Recent – Browses for files and folders recently accessed.
- Browse Web – Opens web addresses.
- Calc – Performs calculations.
- Custom Cmds – Supports creating new commands.
- Customizable – Supports customization, such as skin or font changing.
- Easy Config – Is easy to configure.
- Easy Use – Is easy to use.
- Extensible – Can be functionally extended by plugins or scripts.
- Good Look – Has a visually appealing interface.
- History – Provides command history.
- Misc – Supports other miscellaneous features.
- Run Cmd – Runs command line commands.
- Web Search – Search for terms in one or more search providers.

Table 1 depicts the comparison between all application launchers, regarding their basic features. Although many application launchers implement these functionalities, there are a few that are worthy of being mentioned. For instance, Executor offers a pretty good support for file system browsing, featuring a competent auto-completing mechanism, and also a good web browsing system, which can process all kinds of URL. Some of other application launchers can only process URL beginning with “www”. Launchy, in turn, presents the user with a visually appellative user interface and simplicity of use. It also performs calculations with a familiar math notation, in opposition to some of the other application launchers, which require a specific, hard to master, notation.

There are, also, a few application launchers that are noteworthy, but by the wrong motives. Some, like ac’tivAid, Colibri and Keybreeze, do not feature auto-completing mechanisms, which diminishes the file system browsing potential and just makes the application harder to use. Others, like Executor, have very verbose commands. Although this might be required while using advanced features, it might be troublesome when using simple features, such as the calculator. Imagine that you want to solve the following calculation: 3 + 3 * 2. With Launchy, you can just type “3 + 3 * 2”, while with Executor, you have to type “#3 + 3 * 2”. The user needs to remember that the “#” is needed for Executor to process the string as a calculation. If the user misses it, he will not get the desired result. It is also noticeable that some application launchers are quite hard to configure, such as CAL, Colibri or even Executor. This might be due to complex configuration forms, in CAL’s and Executor’s case, or lack of mouse support in configuration mode, in Colibri’s case.
This category covers all features regarding customization and integration with the operating system and with third-party applications. Examples of these are integration with email clients, web browsers and instant messaging clients; the ability to learn new commands; and the ability
to manipulate folder items and windows. These features are less common than the basic ones, however, they may offer a more valuable contribute for recurrent task automation. For instance, the ability to manipulate folder items is suitable to automate tasks, such as zipping files on-the-fly, with a single command. This task would, otherwise, require more steps to be accomplished. The following 7 items are considered to be Integration and Customization features:

- **App Integration** – Integrates with other applications, such as email clients, media players or instant message clients.
- **Clipboard Mng.** – Offers addition clipboards or enhanced clipboard management.
- **Customize OS** – Can customize custom operative system.
- **Learn Cmds** – Interactively learn new commands.
- **Manage Files** – Performs operations over files and folders, such as editing, deleting and compressing.
- **Manage Windows** – Performs operations over windows, like minimize, maximize, close, and so on.
- **Memos** – Creates notes and memos.

TABLE 2 shows the comparison between all application launchers, with respect to Integration and Customization features. Some application launchers excel within this scope, such as Keybreeze, which can interactively learn new commands. One task that tends to be hard to accomplish is teaching a new command to the application launcher. This task usually requires the user to access the settings form and add a new command, which, in turn, may require some parameterization. Keybreeze bridges this gap by showing a popup, which assist creating a new command, whenever the user types in an unknown command. Enso also presents the user the possibility of teaching new commands, by simply using the “learn” command. Moreover, Enso also excels in window manipulation. With a single command, the user can switch, minimize, maximize and close any window.

In terms of integration with other applications, Dash Command excels in integrating with web apps, such as Google Maps or Google Calendar, and with the web browser. On the other hand, some application launchers, such as Katapult and Quicksilver, integrate very well with media players and email clients, while others like GNOME Do and Skylight, can integrate with instant message clients.

There are, however, two cases of integration whose utility is doubtful: Managing Files and Managing Windows. Take the example of Dash Command into account. Dash allows the user to type “zip” followed by the name of the folder or file to be compressed. Although this might be useful, it is as easy as simply right-clicking the item, hovering your favorite compression application’s entry and choosing “Add to “<folder name>.zip””. On the other hand, Enso and Executor allow the user to manipulate windows, by typing commands to minimize, maximize, close or switch between windows. Although these commands are interesting, they require the user to remember the window title or the command name. Commands such as “maximize” or “minimize” are too long and it might be easier to click in the maximize/minimize box or just Alt + Tab.
<table>
<thead>
<tr>
<th>Application</th>
<th>App Integration</th>
<th>Clipboard Mng.</th>
<th>Customize OS</th>
<th>Learn Cmds</th>
<th>Manage Files</th>
<th>Manage Windows</th>
<th>Memos</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac'ativAid</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>CAL</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Colibri</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dash Command</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Enso</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Executor</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARR</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Google Desktop</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keybreeze</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Launchy</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shuriken</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skylight</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speed Launch</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WinSilver</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dmenu</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gmrun</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GNOME Do</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GNOME Launch Box</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Katapult</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRunner</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Butler</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LaunchBar</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quicksilver</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Todos</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ubiquity</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 2 – INTEGRATION AND CUSTOMIZATION FEATURES COMPARISON**

Being application integration and interactively learning new commands the most remarkable features of this category, the reader can easily notice that there are only four application launchers that include these features. However, some of these, like Enso or Keybreeze fail in many aspects, especially regarding basic features. It would be interesting to have a better synergistic communion between these two categories.
2.2.3 Automation and Context Awareness Features

This category covers all features directly regarding repetitive task automation and user context awareness. Examples of these features are the ability to record macros, replace abbreviations and be aware of currently selected text. Although these features are of major importance for recurrent task automation, they are seldom implemented by any application launcher. The following 4 items are considered to be Automation and Context Awareness features:

- **Basic Automation** – Basic automation features, such as file renaming or abbreviation replacement.
- **Context Awareness** – Is aware of user context and reuses it to perform operations. Example of user context is the text currently selected by the user.
- **Learn Automate** – The ability to learn how to automate repetitive tasks and complete them in the user’s place.
- **Macros** – Supports macro recording and reproducing.

TABLE 3 represents the comparison between all application launchers, with respect to Automation and Context Awareness features. It is easily noticed that none of the review application launcher has the ability to learn how to automate repetitive tasks and only 4 of them are aware of the user context.

User context awareness is a very useful tool to assist user task automation. User context is composed, not only by currently selected text and files, but also other information such as screen resolution, the file system path or web address currently being browsed, clipboard contents or even wireless networks to which the user is connected. To think of a use to this kind of information, imagine that you would like to have your speakers’ volume decreased to 10%, whenever you connect to your work place’s wireless network, and raised to 85%, when connected to your home’s wireless network.

Notwithstanding, the few application that are aware of the user’s context, do not take full advantage of this feature and are limited to only performing operations upon selected text and files. Dash Command goes a little further, being able to detect what webpage the user is currently browsing, to add as a shortcut, and detecting what files are currently selected. However, there is still a lot of room for improvement. For instance, what if the user wants to know his external IP address on-the-fly? Does he really need to go check it on some website? This feature is, indeed, totally context related.

We also have to take into account basic automation features, such as, quick text insertion and file renaming. File renaming is a crucial recurrent task and, therefore, needs to be automated. Activ’Aid has a renaming extension, aimed to solve this problem. However, as you can see in Figure 27, it is not much clear of how it works neither does it offer much room for parameterization. On the other hand, dedicated tools, such as Bulk Rename Utility31, tend to offer too much parameterization, which just makes them too difficult to use. This is easily noticed in Figure 28.

---

31 http://www.bulkrenameutility.co.uk/Main_Intro.php (Last visited in: June 26th, 2009)
The best way to solve the renaming problem is, somehow, related to the Learn Automate feature. There could be an intelligent agent that would be able to learn the renaming pattern, just by watching the user doing it a couple of times. Therefore, the same agent would be able to accomplish the task in the user's place.

Learning to automate is, with no doubts, the most ambitious feature of them all. Understanding which actions compose a repetition and figuring out how to complete them, in the user's place, is a though challenge. No wonder that none of the review applications presents this feature.

A last, but no least, there are system wide macros. Macros are one of the most used mechanisms to automate recurrent tasks. Both Keybreeze and Butler implement this feature, although, both suffer from the same problem: macros are not adaptive. For instance, recorded mouse clicks always occur in the same location, no matter if the user changes windows' positions or not. Recorded tasks, in which text is inserted, require a certain window to have focus. If the wrong window has the focus, the reproduced key presses may have catastrophic results. Due to its lack of adaptability, macros have little use to automate repetitive tasks that arise from everyday usage, which, in turn, tend to be unique.

Moreover, Keybreeze does not really record macros. The user needs to build them using a macro editor, which explicitly involves the user in a slow programming process. The ability to record the interaction would be preferable, as it takes less time and is simpler and clearer to the user. Furthermore, most interactions regarding mouse clicks have delays associated to them. By capturing the delay in the recording process, the user does not need to try guessing how many milliseconds should be elapsed before a certain action is preformed.
As we can see, few are the application launchers that take user context into account and, even those, do not fully explore its potential. Also, only some of them implement some sort of basic automation features, such as quick text insertion and file renaming. Although basic, these features are not that easy to use, forcing the user to resort to complex third-party applications. Not so basic are the macros. Only two of the review applications implemented macros, although they are not adaptive and, therefore, not very appropriated to automate the unique tasks that arise from the everyday usage. To finalize, it is a pity that there is not any application launcher that can learn how to automate users' recurrent tasks. It would be interesting to have a system that properly implements these four features, specially that last one.

<table>
<thead>
<tr>
<th>Application</th>
<th>Basic Automation</th>
<th>Context Awareness</th>
<th>Learn Automate</th>
<th>Macros</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac'tivAid</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>CAL</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Colibri</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Dash Command</td>
<td>✘</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Enso</td>
<td>✘</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Executor</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✓</td>
</tr>
<tr>
<td>FARR</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Google Desktop</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Keybreeze</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
<td>✓</td>
</tr>
<tr>
<td>Launchy</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Shuriken</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Skylight</td>
<td>✘</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Speed Launch</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>WinSilver</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>dmenu</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>gmrun</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>GNOME Do</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>GNOME Launch Box</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Katapult</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>KRunner</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Butler</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>LaunchBar</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Quicksilver</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Todos</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
</tr>
<tr>
<td>Ubiquity</td>
<td>✓</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
</tr>
</tbody>
</table>

**TABLE 3 – AUTOMATION AND CONTEXT AWARENESS FEATURES COMPARISON**
Upon this exhaustive evaluation, it was possible to identify a set of implications, which need to be taken into account, in order to develop a proper solution based on the application launcher abstraction. The typical application launcher interface offers the lost expressive power, once granted by command line interfaces, while properly blending with the host operating system. Such implications are:

- **Auto-completion mechanism** – Auto-completion simplifies command usage and recall. Moreover, it is essential for file system browsing, in order to avoid forcing the user to complete, by hand, long paths.

- **Non-verbose commands** – Commands should be as less verbose as possible. Commands such as "$W$ or "$P$" may be too hard to remember. Therefore, similarity to already well known nouns and the absence of strange symbols are the keys to attain this feature.

- **Easy command usage** – The user should not have to deeply memorize commands’ names neither should he be too much concerned about commands’ syntax. For instance, the application should tolerate typos and abbreviations. Also, commands syntax should be as simple as possible, avoiding the use of uncommon special characters to denote parameterization. Moreover, if the user mistakenly types the parameter before the commands name, the agent should correctly infer what he was trying to do.

- **Simplicity of use** – The application should be very simple to use: simple commands, visually appealing interface and easy configuration forms. However, there must be a way around to allow power-users to do further configuration and customization, be it by scripting or by providing advanced configuration forms.

- **Interactively learn new commands** – There must be a way to teach new commands without having to pass through the configuration form. This way must be as simple as possible and resort to the user context, as much as possible, to identify any parameterization clauses.

- **Smart agent** – The user should not have to remember complex notations in order to use a common feature. The user should just types what he wants and is of the agent's responsibility to correctly interpret the command.

- **Explore user context** – User context awareness should be explored to a new level, and take part on every action that the agent performs towards the user.

- **Learn how to automate** – The application (agent) should learn what actions belong to a repetitive task and what tasks should be automated. This must be attained in a non-intrusive way, without interfering with the user's activity. Upon identifying a repetitive task, a suggestion must be made to the user, offering the possibility to perform the remainder of the task, in the users place. This suggestion must, also, be made in a non-intrusive way.

- **Adaptive macros** – The application should record macros that can adapt themselves to new situations, such as different window locations and different window instances.

Besides these implications, it was also possible to summarize which features are essential and which are optional, taking into account the aim of automating recurrent tasks. Tables 4, 5 and 6 summarize feature suitability, regarding each category. Although optional features are of smaller relevance, they can also be implemented as extensions, if anyone feels the need to use them.
2.3 AUTOMATION MECHANISMS ANALYSIS

Although there are several application launchers currently available, which can speed up user activity, they lack of significant automation tools. It is, therefore, imperative to answer a few questions, in order to design an application capable of automating repetitive tasks. First: How do we detect a repetition? How do we automate it? Second: How can we interact with the user’s context? Fortunately, there are a few prototypal systems that address these issues, worthy of being analyzed.

But first, we need to clarify the “automation” concept. We define Automation as set of generalized actions which can be executed in order to fulfill a recurrent action, in the user’s place. We define generalized action as an action which is produced taking into account the relationship established between two former actions, in order to adapt to new situations, different from the ones in which those were first detected. Therefore, a system capable of automating repetitive tasks is a system that is able watch the user, learn which tasks are or not recurrent and generalize each repetitive tasks’ actions, in order to create new actions suitable to accomplish the user’s forthcoming operations.

As an example, take the first one provided in the Introduction, where you want to rename all of your fifty digital photos from “SDC10050.JPG”, “SDC10051.JPG”, “SDC10052.JPG”, etc., to something more meaningful like “Summer Vacation 01.jpg”, “Summer Vacation 02.jpg”, “Summer Vacation 03.jpg” and so on. A system capable of automating repetitive tasks would monitor your actions and be aware of what you were doing. It would know that your actions represent file renaming under the specific numeric sequence pattern “SDC10%\(i\).JPG”, where \(%i\) represents the numeric sequence 050, 051, 052 … 100. Consequently, your monitored actions would be
generalized into new actions that would rename the remaining files in that folder, consistently with the detected pattern. These actions, therefore, compose an Automation which, when executed, will complete the repetitive task for you.

2.3.1 How to Detect a Repetition? How to Automate It?

One of the major problems, regarding intelligent interfaces, has always been detecting what the user is up to, what’s his/her goal.

2.3.1.1 Programming by Example based on Common Sense

Fraaborg and Lieberman brought us Creo [7], a programming by example system, which is able to automate recurrent tasks, in Internet Explorer, and identify high-level user goals. Creo has the interesting feature of integrating with a data detector system, named Miro, which uses the semantic power of webpage’s contents to decide which automations should or not be invoked. To attain this, Creo resorts to knowledge data bases, such as MIT’s ConceptNet [32] [18] and Stanford’s TAP [33] [8] [9]. Both consist of machine-readable logical predicates concerning everyday knowledge about the world, which cover a wide variety of topics, like: music, movies, actors, television shows, books, sports, companies, toys, countries, cities, electronics, video games, and drugs.

Creo embraces the macro recorder interface. It records sequences of actions, on demand, which can later be reproduced. However, what differentiates Programming by Example systems like Creo from common macro recorders is their ability to generalize information. Creo determines if the input should or not remain static, based on three heuristics: 1) if the input consists of personal information, 2) if the name of the text field matches a predetermined list of fields that should remain static, and 3) the number of generalizations of the input. If Creo determines that the input should be generalized, then it looks up for relevant relationships for the input, in both ConceptNet and Tap, and builds the appropriate generalizations. These, therefore, are leveraged by Miro when determining whether the recording should be activated based on the semantic context or not.

Miro, in turn, determines users’ potential goals based on the web page they are browsing, by matching the generalizations of terms and phrases on the page to the set of generalizations associated with the recordings. Miro then converts plain text into hyperlinks for a particular recording. This interesting feature allows that recorded sequences of actions might adapt and be reproduced while facing new situations, different from the ones in which they were created.

2.3.1.2 Text Prediction based on Common Sense

In applications that receive text as user input, there is the need to process that text in order to perceive the user’s intention. Therefore, textual prediction algorithms have to be applied. Stocky [30] enumerates a few ways of predicting user text. One of the first predictive typing aids was the Reactive Keyboard [2], which made use of text compression methods [32] to suggest completions. These methods generally suggest words based on: frequency, either in the context of relevant corpora or what the user has typed in the past; or recency, where suggested words are those that the user has most recently typed.

32 http://conceptnet.media.mit.edu/ (Last visited in: June 27th, 2009)
Stocky also suggests a new approach of textual prediction, based on common sense. To achieve this, as the user types, a system queries OMCSNet\(^3\) [19], a knowledge database similar to ConceptNet, for the semantic context of each completed word, disregarding common stop words. OMCSNet then returns a list of phrases, each containing one or more words, listing first those concepts more closely related to the queried word. As the system proceeds down the list, each word is assigned a score: \( \frac{1}{\log_{5}(5+n)} \), where \( n \) increments along with the list index. This way, the score decreases along with the resemblance between the words and the concept specified by the user. The base 5 scoring algorithm was chosen based trial-and-error.

Upon weighing each phrase, the words are added to a hash table of potential word beginnings and complete words. A words final score results from summing that word’s individual scores over all appearances in semantic contexts for past queries. Thus, as the user begins to type a word, the suggested completion is the word in the hash table with the highest total score that starts with the typed letters.

This prediction method attained similar results to frequency and recency based approaches, for most cases. Nevertheless, this approach is advantageous when there is an adequate coverage of the addressed topics in the OMCSNet. Furthermore, it excels in cases of low word repetition.

2.3.1.3 Implicit Programming by Example

Programming by Example (PbE) is a very important concept for intelligent interfaces. PbE systems learn how to automate tasks by learning which actions belong to a repetitive task, from an example provided by the user. For instance, the user starts a recording and the system learns from every action the user performs before the recording stops. However, this approach involves the user explicitly in the programming process. He needs to be aware of the macro metaphor, in which his actions need to be recorded in order to later be reproduced.

Fortunately, there is an interesting approach to solve this issue. Ruvini introduces the Implicit Programming by Example concept [26], which is based on two key ideas: 1) avoid the macro recorder metaphor so the user does not have to start and stop the recording; 2) the repetitions are automatically detected from observing the user’s behavior and automations are automatically inferred. This avoids the user to be explicitly involved in the programming process, reducing the user effort as much as possible. This is important for several reasons: 1) there are times when the user is not aware of the repetitions he is performing, and 2) inferring the correct automation from few examples is rather difficult, which confers a low reliability to PbE systems and frustrates the user, while facing complex cases. In implicit PbE systems, as the user does not make any effort, every correct suggestion the system makes is considered bonus and every incorrect one does not hurt.

Ruvini mentions a few Implicit Programming by Example systems that are noteworthy, such as Eager [1] and APE [27] [28]. Eager is an assistant for the HyperCard environment, which is able to detect the first iterations of a loop in the user’s behavior and to propose to complete the loop until a condition is satisfied. APE, in turn, is an assistant for a programming environment which offers to automate repetitive sequences of actions or commands, to complete loops or to write repetitive portions of code. It extends Eager functionality by offering automation even if the different iterations of these repetitions are not consecutive in the user’s actions history. The user’s actions history is defined by being the sequence of actions the user performs in a work session.

\(^{34}\) http://web.media.mit.edu/~hugo/omcsnet/ (Last visited in: June 27th, 2009)
An implicit PbE system has to main tasks: 1) learn what to automate from observing the user's behavior, and 2) learn when to make a suggestion to the user. Learning what to automate is not an easy task. Whilst the macro recorder resorts to the user to explicitly mark where an iteration begins and ends, an implicit PbE system has to identify the iterations without explicit intervention from the user. A few types of repetitions were identified:

**Constant**: all iterations from one repetition are identical. For instance, repeatedly inserting a table with a specific number of rows and columns in a text processor.

**Loop**: all iterations comprise the same actions but are repeated over a set of objects. For example, formatting all the pictures in a document.

**Nested loops**: iterations are identical but embedded in two nested loops. For example, for each document \( d \) in a folder, edit \( d \) and add a border to each picture in \( d \).

**Conditional loop**: iterations are identical but are repeated over a set of objects exhibiting a certain property. For instance, for each message \( m \) in the mailbox, forward it to Dan if the sender of \( m \) is Tessa.

**Variable loop**: the iterations are repeated over a set of objects and depend on a property of the object. For example, for each message \( m \) in the email box, select \( m \) and save it to ‘SubjectOf(\( m \)).txt’.

Detecting loops and nested loops requires the system to identify, from a few iterations, the body of the loop and the set of affected objects. APE is able to detect loops even if the iterations are not consecutive. Each time APE finds a constant repetition it assumes it is a loop candidate and checks if it has been iterated over objects belonging to the same set. To attain this, it compares the actions preceding the occurrences of the repetition, using a similarity measure based on well known predefined sets, like sequences of integers, days of the week, alphabetic sequences, files in a specific folder, and so forth.

Conditional loops and variable loops are much more difficult to detect, as all iterations are different because no action repeats exactly the same. Therefore, iteration identification cannot be attained with repetition searching algorithms.

### 2.3.1.4 The APE system

The Adaptive Programming Environment [28], or APE for short, is composed by three software agents: the **Observer**, the **Apprentice** and the **Assistant**. These three work together in the background, without any user intervention.

The **Observer** monitors user's actions and stores them in domain objects into the actions' history. Also, the Observer notifies the Apprentice and the Assistant that the user has performed a new action.

The **Apprentice** has two main goals: 1) detect repetitive user tasks and 2) examine the situations in which the repetitive tasks were performed. To attain this, two machine-learning algorithms are employed to learn situation patterns and build two sets: 1) When – set of
situation patterns matching the situations in which the user has performed the detected recurrent tasks, and 2) What – set of the user’s habits. A situation pattern consists of a regular expression matching one or more situations. A habit is a pair of situation patterns and repetitive tasks such that the situation patterns match all situations preceding the repetitive task.

To learn the When-set a decision-tree algorithm is used, named C4.5 [25]. This algorithm presents a low computational time and offers adequate results. However, it proven to be not suited to learn the What-set, as it generates too-general situation patterns. Therefore, a new algorithm, named IDHYS [27] had to be developed to learn the What-set. IDHYS is a concept learning algorithm inspired by the Candidate-Elimination algorithm (CEA) [22]. Inductive concept learning consists in acquiring the general definition of a concept from training examples of this concept, each labeled as either a positive example or a negative example of that concept. All the situations preceding a repetitive task can be seen as positive examples of the concept, while situations preceding other tasks are considered to be negative examples. The searched definition is, therefore, a set of situation patterns that match the situations in which the repetitive task was performed.

However, this approach can also be seen as a hypothesis space search problem [21]. IDHYS searches the hypothesis space for conjunctions of two situation patterns. These hypotheses are the most specific generalizations built from the provided positive examples. The whole process is incremental as it starts with a very specific hypothesis and progressively generalizes it with subsequent positive examples.

The Assistant inspects the When-set, whenever the user performs an action, in order to detect if the user is performing a repetitive task or not. If the actions performed by the user correspond to one or more learned situation patterns, then it inspects the What-set to infer what repetitive task the user is going to perform. Then, the Assistant displays a window, without interfering with the user’s work, showing the actions composing the repetitive task. The user can just ignore the window or select one of the suggestions, which will cause the Assistant to successively perform those actions.

2.3.1.5 Issues Related to Repetitive Task Detection

Ruvini points out a few problems concerning repetitive task detection [26]. In implicit PbE, learning to automate is based on two processes: 1) identify positive examples of the iterations composing the repetitive task, and 2) infer a predictive pattern from those examples. For the first process, the existing approaches present two major limitations: they are not incremental and the employed algorithms are not adequate for some search cases. An example of this is the APE’s inability to detect repetitive actions when the order in which the actions are performed changes between iterations, just like the Conditional and Variable loops’ cases.

A possible solution for this problem might be recording user’s actions in a structured multi-level history, as proposed by Kosbie and Myers [12], where a set of low-level actions can be grouped together to compose a higher-level action. For instance, opening the “Save As” dialog, editing the file name and closing the “Save As” dialog can be grouped into a higher-level action such as “Save As”. This approach solves the action ordering problem since the detection is taken care at a higher level.

For the second process, the actual approaches infer predictive patterns from potential positive examples, based on predefined domain dependant operators to identify the affected objects and predict future iterations. Lau and Weld [14] propose an approach based on domain independent operators. They have shown that repetition automation can be seen as a prediction problem:
predict the next user iterations from the previous ones. This allows implicit PbE systems to resort to several machine learning algorithms to predict the next iterations. However, such systems cannot simply predict iterations from examples; they also have to be able to examine alternative hypotheses about the concerned iterative task. Lau et al [16] proposed an algorithm to accomplish this task, based on a structured multi-level representation of the user’s behavior, in a similar fashion of the one described by Kosbie and Myers.

There are also a few problems concerning the performance of the employed algorithms, since learning when to make a suggestion is a typical machine learning problem. Davidson and Hirsh suggested to weight recent user actions using an exponential decay function [3]. On the other hand, Yoshida and Motoda have shown interest in using meta-knowledge in the representation of the users actions’ history [33]. Examples of meta-knowledge might be the flow of information between commands.

2.3.1.6 Programming by Example using Version Space Algebra

As we could see from the previous subsections, learning how to automate can be solved using hypothesis space search algorithms. As matter of fact, there is a system that takes advantage of this approach and enhances it to a new level. SMARTedit\(^35\) [16] [15] consists of a text editor, which can automate repetitive tasks, resorting to machine learning mechanisms based on version space search [21]. Lau introduces the Version Space Algebra [13], which allow the composition of more complex version spaces by composing together simpler version spaces. In other words, it defines how sets of small and simple functions can be combined in order to compose more complex ones, therefore constituting a hierarchical structure.

SMARTedit adopts the macro recorder metaphor. The user needs to explicitly start and stop the iteration recording. At the beginning of a recording, SMARTedit has a hierarchical version space, containing all possible programs that it is able to learn. As the user goes on, some functions are thrown out of the version space, because they are not consistent with the observed actions. This way, as long as the user is recording his actions, the version space gets increasingly more specific, allowing SMARTedit to pick the best choice, once the recording is finished.

SMARTedit picks the most likely action by weighing the resulting application state of each version space. First, it takes snapshot of the current application state and treats it as input, executing each of the functions in the version space on this input, in order to produce a set of output states. If the result is a single output state, then it is presented to the user with 100% probability of being what the user intended to do. If there is more than one result, the one with highest probability is chosen.

It is interesting to point out that, to build a hierarchical version space and to composite more complex version spaces from simpler ones, SMARTedit relies on domain knowledge, taking into account the importance of each domain object.

2.3.2 Being Aware of the User’s Context

There are, already, a few applications that focus on user context. Alan Dix [5] briefly describes three of them: onCue, a desktop internet-access toolbar, SnipIt\(^36\), a web-based bookmarking

\(^{35}\)http://tlau.org/research/smartedit/ (Last visited in: August 1\(^{st}\), 2009)

\(^{36}\)http://www.snipit.org/ (Last visited in: June 27\(^{th}\), 2009)
application, and ontoPIM\textsuperscript{37}, an ontology-based personal task-management system. These systems embody context issues to differing degrees, which are of major importance of study, in order to, efficiently, explore the user context.

### 2.3.2.1 A few Implications

To onCue, user context results from autonomously inspecting the clipboard to suggest actions. On the other hand, Snip\textsuperscript{t} captures user's selected text. The former is a kind of \textit{incidental interaction} \textsuperscript{[6]}, where the user's intention is focused on one thing, which is copying text, and "incidentally" some other system actions occur, like onCue changing its icons.

In cases where the system takes the initiative within the interaction, it is important that the system embodies principles of \textit{appropriate intelligence} \textsuperscript{[4]}, by embedding the intelligence within an appropriate interaction framework. No matter how simple an intelligent system is, it needs to follow two important rules: 1) "try getting thing right as often as possible", and 2) "when things go wrong, do not mess up with the user". In other words, the system must not interrupt the user's work nor, somehow, distract him. An old yet memorable example of an agent that does not respect these rules is the Microsoft Office 2000 and XP paper clip. He can say quite useful things when it is right, but if it is wrong... it has already interrupted your typing and spoiled your chain of though.

### 2.3.2.2 A few Architectural Remarks

It was also denoted that the recognizer-service paradigm proved to very flexible. The recognizer is a separate module, responsible for examining a certain source of data. The service, in turn, is responsible to perform actions upon recognized data. This architecture allows recognition to be separate from a single action and, therefore, offers the possibility of bounding additional actions to an already existing recognized type. onCue is a good example that embraces this paradigm.

onCue is also notable by working across many applications, resorting only to the clipboard. As a matter of fact, the clipboard is usually the only truly application independent source of data on a graphical user interface system. Furthermore, for this reason, there is another application named Citrine\textsuperscript{38}, which is purely based on intelligent clipboard to clipboard interactions [31].

### 2.3.2.3 Types of Context

In order to be aware of the user context, we first must clearly identify what the user context is. Alan Dix categorized the user context in four types [5]:

**WHAT** – data currently focused by the user. For example, currently selected text or clipboard contents.

**WHERE** – Immediate environment. For instance, emails with a similar subject to another or the country of origin of a web page.

**WHEN** – Trace of recent user activity. For example, knowing the last visit website and when it happened.

**WHO** – Profile and/or preferences of the user and long-term activity. For instance, influencing the results, admitting as a factor the user's nationality.

\textsuperscript{37} http://www.comp.lancs.ac.uk/~dixa/papers/ontopim-2006/ (Last visited in: June 27th, 2009)

\textsuperscript{38} http://www.cs.cmu.edu/~citrine/ (Last visited in: June 27th, 2009)
Although these types are all in the scope of digital environments, it is clear that similar categories are common in ambient intelligence.

2.3.2.4 Beyond the Virtual World

As a matter of fact, a lot of research has been made in the context of ubiquitous systems. These aim to explore virtual and electronic resources, such as sensors, in order to capture information inherent to the environment that surrounds the user and about the user himself. This information is used to help him to navigate and adapt to the real and physical word that surrounds him. This is, indeed, an interesting area to explore, even if it is just to find out how further we can extend the "user context" concept.

Hsinyi Peng [24] explores the interactivity concept and its applications in ubiquitous-learning systems. Nowadays, the rapid progress in the mobile and wireless technologies realm has created new computational environments, which are shaping the way how people learn and work. Several embedded devices and software applications are constantly connected to the internet wirelessly, on today's portable gadgets. This new Internet-ready environment has been called a ubiquitous-computing environment. One of its most notable features is the use of wireless communication resources with sensors to capture user and environmental information and provide personalized services. This feature is often called “context aware” computing [10], and is of major interest to expand the “user context” concept.

2.4 AUTOMATION MECHANISMS EVALUATED AND DISCUSSION

After reviewing the most relevant automation mechanisms to date, a careful evaluation was paid to them, followed by a discussion. A total of 14 features were taken into account for comparison, divided by 2 categories: 1) Automation Basis Features – features that serve as a basis for automation mechanisms and 2) Direct Interaction Features – features that have direct influence in the way how users interact with the system. Both categories will be evaluated in the next two subsections, which will contain tables depicting which features are used by a specific mechanism. A checkmark (✓) denotes that a feature is met by the mechanism and an x mark (✘) denotes otherwise.

2.4.1 Automation Basis Features

This category covers all features that, somehow, contribute substantially for a good automation mechanism. Some have a more direct impact, such as Detecting Repetitions or exploring a Hierarchical Hypothesis Space; others have a lesser impact, such as Semantic Generalization or User Context Awareness. Nonetheless, these features are important as they provide support to Direct Interaction Features. The complete list of the Automation Basis Features is the following:

- Application Independent – Uses application independent means to capture user data.
- Detect Repetitions – Is able to detect iterations of a recurrent task, by using repetitive pattern searching algorithms.
- Detect NC Repetitions – Detects a repetition even if its iterations are non consecutive in the user's actions' history.
- Hierarchical Hypothesis Space – Uses a hierarchical hypothesis space as a base for the learning algorithm.
- Hypothesis Space – Uses hypothesis space as a base for the learning algorithm.
- Physical Context Awareness – Is aware of the user’s context, concerning the surrounding physical world.
Semantic Generalization – Uses semantic data bases to generalize user’s actions, according to their semantic meaning.

Virtual Context Awareness – Is aware of the user’s context, concerning the virtual environment inherent to the operating system or a specific application.

Table 7 shows the comparison of all significant mechanisms that can be used to automate user tasks, regarding automation basis features. Although more mechanisms where analyzed, only 6 are shown as these manage to reuse the ones that were not included. For instance, it is not essential to mention the Version Space Algebra mechanism as it is already featured by SMARTedit, which is already mentioned.

Detecting repetitions is, with no doubt, the most important feature of this category. Using repetitive pattern searching algorithms is required in order to detect which actions belong to a repetitive task or not, without user intervention. From the reviewed mechanisms, APE and Eager are the only systems featuring repetition detecting algorithms. They are able to infer predictive patters from monitored user actions. Although, APE is the most notable one as it can detect non consecutive iterations. Both SMARTedit and Creo fail in this matter, as they adopt the macro recorder metaphor and, therefore, require the user to be explicitly involved in the programming process. However, SMARTedit presents a few advantages when compared to APE. SMARTedit’s version space algebra allows it to combine hypothesis spaces in order to compose more complex actions from simpler ones. This allows SMARTedit to examine several alternative hypotheses about an iterative task, which cannot be done by APE. Also, APE’s Learning to Suggest algorithm, IDHYS, works differently from SMARTedit’s version space algebra. SMARTedit’s version space is hierarchical and is initialized with all programs that can be done. As the user performs more actions, functions are thrown out as they stop being consistent, causing the space to get more and more specific as the user keeps going on. APEs algorithm works opposite. As more and more actions are performed, the version space gets more and more generalized.

Creo, in turn, has semantic generalization. It is able to generalize user actions based on its semantic meaning, allowing these actions to be later reproduced over items with similar meaning. Creo attains this by resorting to a common sense data base and a data detector named Miro.

Alan Dix [5] points out a few important architectural issues of which the most important one is having an application independent means to capture user data. This is very important as there are too many different applications nowadays and not all of them respect standard protocols. Picking up again the application launcher subject, take Enso as an example. Enso as the interesting feature of spellchecking currently selected text. Although this feature works with many applications, our testes revealed that it is not able to capture selected text in some applications, such as Google Chrome. As Enso is not open source, we could not check its code, but probably it resorts to Microsoft Active Accessibility (MSAA) or other application dependent mechanism to capture selected text. Using the clipboard would be an easier and reliable way to capture user selected text as it works on every application that allows its text to be copied.
Alan also identified four types of user context: What – data currently focused by the user; Where – Immediate environment; When – Trace of recent user activity; and Who – Profile and/or preferences of the user and long-term activity. These types are of major importance to explore the user’s virtual context, as they represent valuable sources of data. Nevertheless, Where, When and Who types can be extended with information from the outward world. Nowadays, portable computers are common and replete of wireless resources, such as Wi-Fi network cards and Bluetooth devices. These can be used to capture valuable information regarding the user’s physical location and where he has been, in order to personalize services [24] [10].

As we can see, none of the reviewed systems has, simultaneously, the ability to detect repetitive non consecutive repetitions, be aware of both virtual and physical user contexts and feature an application independent means of capturing user data.

### 2.4.2 Direct Interaction Features

This category includes all features that have a direct impact in the way of how the user interacts with the system. Examples of these are the ability to learn how to automate and to suggest, to add semantic meaning to usable items or user text prediction based on common sense knowledge. These features are directly backed up by the ones in the previous subsection. The list of Direct Interaction Features taken into account is the following:

- **Common Sense Prediction** - Predicts user text resorting to a semantic data base.
- **Explore Alternatives** – Is able to explore alternative automations.
- **Generalized Actions** – The actions composing the generated sequence to automate the repetitive task are generalized. Therefore, the sequence can adapt to new situations, different from the one in which it was first recorded.
- **Learn to Automate** – Learn what actions belong to a repetitive task, without the user’s intervention, inferring a predictive pattern from the monitored activity. This pattern
should be used to build a new sequence of actions that will fulfill the remainder of the task.

- **Learn to Suggest** – Learn when to make a suggestion to the user without disturbing him.
- **Semantic Meaning** – User’s actions over a certain item are conditioned by the item’s semantic meaning.

Table 8 depicts the comparison of all significant mechanisms regarding direct interaction features.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Sense Prediction</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Explore Alternatives</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Generalized Actions</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Learn to Automate</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Learn to Suggest</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Semantic Meaning</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Both APE [28] and Eager [1] are able to learn how to automate and learn how to suggest, though APE is superior to Eager as it is able to detect non-consecutive iterations. Learning to Automate is backed up by Detect Repetitions feature from the previous subsection. It uses a repetitive pattern searching algorithm, named KMR, to inspect the user’s actions’ history and detect whether he is performing a repetitive task or not. Learning to Suggest is backed up by the Hypothesis Space feature, as APE uses the IDHYS concept learning algorithm to explore hypothesis spaces. Although SMARTedit [15] embrace the macro recorder metaphor and, therefore, is not able to learn how to automate, it can explore alternative tasks to the one actually being performed by the user, due to the use of version space algebra. APE, in turn, is not able to do that.

Creo [7] is another system that also embraces the macro recorder metaphor and, therefore, is not able to learn how to automate. However, it is able to learn when to suggest using common sense. Imagine someone using Creo, recording all actions composing a task of buying potatoes. These actions are generalized to be sensible to a certain type of data, such as vegetables. If the user goes to another webpage and finds other words related to vegetables, for example, lettuces, Miro automatically highlights lettuces as a link to execute that recorded task, in which reproduced actions will adapt to a new vegetable. Pressing lettuces would result in buying them on his favorite online store. Creo can, therefore, Generalize Actions and use the Semantic Meaning of the affected items to personalize the interface and the services provided.

Stocky [30] describes a text predicting mechanism that resorts to common sense knowledge to weight the indexed items. The results from his testes showed that this innovative approach excels if there is not too much repeated words in the user input and if the knowledge base has an
adequate coverage of the addressed topics. We can see that text prediction regarding the semantic value from the affected items, is an interesting and worthy of being explored way for guessing what the user is trying to say to an intelligent interface.

It is easily noticeable that there is not a single mechanism that, simultaneously, learns how to automate and to suggest, generates generalized actions that can adapt to new situations, and capable of taking into account the semantic meaning of the affected items to the user. Furthermore, all of the reviewed mechanisms are only applicable to a single application. It would be interesting to have an automation mechanism with those features that would operate over an entire operating system and all applications running on it.

2.4.3 Discussion

By evaluating these automation mechanisms we were able to identify a few high-level requirements for a system capable of automating user’s recurrent tasks, without disturbing him. These high-level requirements are:

- **Learn to Automate** – Identify which actions belong to a repetitive task. For this, the system must monitor user activity and use a repetitive pattern search algorithm to identify predictive patterns. The system has to be able to detect recurrent iterations of a task even if these occur non-consecutively in the actions’ history.

- **Generalize Actions** – Once a predictive pattern of actions was detected, these should be generalized into new actions capable of completing the repetitive task in the user’s place. These generalized actions must, therefore, have the ability to adapt to new situations, different from the ones in which their predecessors were detected, in order to later be reproduced in an always-changing environment.

- **Learn to Suggest** – Know when to suggest an automation capable of completing the repetitive task. This suggestion must be non-intrusive as the user should not be distracted from what he is doing.

- **Be Aware of the User Context** – Know as much as possible about the user environmental context. What text he has currently selected, what website he is browsing, what files he as currently selected and in which folder, and even to which wireless network he is connected to. To capture this information, the system must feature capabilities to explore both user virtual and physical environment, such as the ones provided by the operating system and wireless devices. Furthermore, the system must have an application independent means of capturing user data in the virtual environment, in order to be compatible with any application.

- **Take Semantic Meaning Into Account** – Offer a mechanism that allows items to be classified according to their semantic meaning. This provides the user with better item retrieval and reuse.

Tables 9 and 10 summarize which automation basis features and which direct interaction features are essential or optional. Since we are aiming for an implicit Programming by Example system, we considered as optional the ones directly associated with explicit PbE systems or with specific algorithms, such as Version Space Algebra or IDHYS. Although these algorithms are adequate for a single application, they might not be suitable for an operating system wide application.

Even though we are considering the items semantic meaning as valuable information to enhance its retrieval and reuse, we consider common sense text prediction and semantic generalization as optional features because not all semantic meaning has to be related to common sense
knowledge. The use of semantic databases might be too computationally expensive for an operating system-wide application.

It is also clear that APE is the most notable reviewed implicit PbE system. We found its three-agent architecture to be inspiring. It is desirable for a system capable of automating recurrent tasks to have an agent for monitoring user activity, other agent focused in learning what to automate, and yet another one focused on learning when to suggest.

<table>
<thead>
<tr>
<th>Essential?</th>
<th>✓</th>
<th>✓</th>
<th>✘</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Application Independent</td>
<td>Detect Repetitions</td>
<td>Detect NC Repetitions</td>
<td>Hierarchical Hypothesis Space</td>
<td>Hypothesis Space</td>
<td>Physical Context Awareness</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✘</td>
<td>✘</td>
<td>✓</td>
<td>✘</td>
</tr>
</tbody>
</table>

TABLE 9 – AUTOMATION BASIS FEATURES SUITABILITY

<table>
<thead>
<tr>
<th>Essential?</th>
<th>✘</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Common Sense Prediction</td>
<td>Explore Alternatives</td>
<td>Generalized Actions</td>
<td>Learn to Automate</td>
<td>Learn to Suggest</td>
<td>Semantic Meaning</td>
</tr>
</tbody>
</table>

TABLE 10 – DIRECT INTERACTION FEATURES SUITABILITY
3 A FULL-FEATURED AUTOMATION SYSTEM

We propose a new Implicit Programming by Example approach, consisting of a system named Blaze, which aims to address the problems concerning the identification of recurrent tasks, operating system wide, and inferring and suggesting automations capable of automating those tasks.

Just like APE, Blaze consists of three automation agents: the Observer, responsible for monitoring the user; the Apprentice, responsible for learning when the user is performing a repetitive task; and the Assistant, whose function is to suggest to the user a sequence of actions capable of automating a recurrent task. Apart from this architectural similarity, blaze is quite different from APE. It provides a much wider range of functionally, by being able to monitor user activity at the operating system level, whilst APE's approach is focused on a single application. Our approach also presents a repetition detection algorithm slightly faster than the APE's KMR, which consists of a significant improvement.

Blaze is also inspired on Creo's ability to save macros that can adapt to new situations, by taking the affected items' semantic meaning into account. Blaze adopts a similar approach by implementing Adaptive Macros, which are macros that can adapt to new situations, like the ones in which affected windows' positions may be different from the ones in which the macro was first recorded, and reproduced mouse clicks would therefore miss their targets. Blaze's adaptive macros circumvent this obstacle, by using the mouse's relative position to the window. This grants our system with the ability generate and export automations that can adapt to new situations.

Blaze also stands out by presenting advanced user context awareness. While other approaches and application launchers do not take much more than the user's selected text or files into account, Blaze offer means to explore much more information and to reuse it. Blaze allows users to embed wildcard commands with other Blaze commands, which refer to contextual data. This provides the users with means to manipulate their own contextual information, which no other system can provide. Moreover, Blaze uses the clipboard to capture user selected contents, which is the only true application independent means of exchanging data. Other applications launchers resort to other dependent mechanisms, which prevent them from working with some applications.

After reviewing all available application launchers and automation mechanisms, we were able to identify their shortcomings and establish 7 main requirements that must be met in order to develop an effective system, capable of automation recurrent tasks without major user intervention. These requirements also take into account the implications identified in sections 2.2.4 and 2.4.3.
The established requirements are:

1. The interface must be expressive and yet simple. Therefore, we chose the typical application launcher interface for Blaze.
2. As we picked an application launcher as the main interface, we have to enhance its typical features to a new whole level.
3. The system must be aware of the user context and use that information to speed up his activity. Also, the users must be allowed to reuse their own contextual data.
4. The system must be easily extensible, as the user should not have to compile code in order to add new commands.
5. The system must detect repetitions by examining the user's activity and, therefore, offer to auto-complete the task for the user.
6. Both the repetition detection and the suggestion must not distract the users from what they are doing.
7. Synthesized automations should be able to adapt to new situations, different from the ones in which they were initially detected. Also, these automations should be able to be saved for later reproduction.

The next subsections will address the approaches adopted to meet these requirements and give a detailed description of the solution's architecture.

3.1 BLAZE AS AN APPLICATION LAUNCHER

The typical application launcher was chosen to be the interface for Blaze as it provides a lot of expressive power and, yet, integrates very well with the host operating system. A typical application launcher is composed by a main graphical user interface, usually disposed as a landscape rectangle, and a textbox, in which the user types commands that are processed by the application, once the user hits the return key. The interface itself is usually simple and visually appealing. Consequently, this kind of interface offers the expressive power once granted by former command line interfaces, allowing the user to launch applications and run commands without having to navigate through dozens of menu entries. By adopting this approach, we meet requirement 1 (an expressive yet simple interface). Figure 29 depicts Blaze Interface.

By adopting the application launcher metaphor, Blaze implements most of the features included by other application launchers and enhances them to a new level. Major enhancements were carried out on three major areas: text prediction, user context awareness and system extensibility.

These enhancements will be addressed in the next subsections.

3.1.1 Features List

Blaze offers most of its counterparts' features and a few more. Taking into account the features identified in subsection 2.2, Blaze implements the following:
Basic Application Launcher Features:

- Indexes and launches applications.
- Indexes text files’ contents.
- Indexes ID3 tags from music files.
- Features a name and path auto-completion mechanism.
- Browses the file system. (Figure 31)
- Browses for files and folders recently accessed.
- Opens web addresses.
- Performs calculations. (Figure 30)
- Supports creating new commands.
- Is easy to configure.
- Is easy to use.
- Can be functionally extended by plugins and scripts.
- Has a visually appealing interface.
- Provides command history, by learning which commands are more often used by the user.
- Runs command line commands. This feature is enhanced with context information, allowing commands to be applied over folders currently opened on a windows explorer window.
- Search for terms in several search providers.

![FIGURE 30 – BLAZE PERFORMING A CALCULATION](image)

![FIGURE 31 – BLAZE BROWSING THE FILE-SYSTEM](image)

Integration and Customization Features:

- Integrates with other applications, such as email clients.
- Interactively learns new commands, resorting to context awareness to identify the new command’s name and executable path.
- Performs operations over files and folders, such as editing, deleting and compressing, resorting to command line operations over a specific folder.

Automation and Context Awareness features:

- Basic automation features, such as file renaming and quick text insertion.
• Is aware of user context and reuses it to improve his experience. Blaze can capture selected text by the user in any application; selected files and folders on Windows Explorer; address of the website or folder currently being browsed; and more.
• The ability to learn how to automate repetitive tasks and complete them in the user's place.
• Supports macro recording and reproducing. Blaze introduces Adaptive Macros, which can adapt to new situations, when reproduced, different from the one in which it was first recorded.

3.1.2 Improved Text Prediction

In order to identify what application or what command the user is willing to launch, an application launcher needs a text prediction engine. This engine is responsible for matching the user input to an item in the index.

3.1.2.1 Limitations of Current Algorithms

Common application launchers use a basic text prediction engine. On one of these, the matching is done by comparing the text inserted by the user with each name contained in the database, performing a string contention test. This test returns a value which is used to weigh each database item. If the test fails and no textual contention exists between the item and the input text, then the item is discarded. The test fails if not all of the input’s characters are matched with the item’s name, respecting the order in which they are presented. The items are then shown to the user according to their score, those with the best scores and, consequently, the ones closer to what the user was looking for, are displayed at the top.

For example, imagine that the user is looking for Mozilla Firefox, which is an indexed item, and types “ffox”. The comparison test between that item and the input, for a basic text prediction engine, would be:

| Indexed Item: | Mozilla Firefox |
| User Input:   | f fox |

As we can see, “ffox” is totally contained by the string “Mozilla Firefox”. To the “Mozilla Firefox” item would be assigned a good score, which would depend on the adopted implementation. Now imagine that ”Foxy Jumper” was also an indexed item. The comparison test between “Foxy Jumper” and “ffox” would be:

| Indexed Item: | Foxy Jumper |
| User Input:   | ffox |

The reader can easily notice that “ffox” is not totally contained by “Foxy Jumper”, as the second “f” is not matched. Therefore, the item ”Foxy Jumper” is discarded and will not be displayed to the user.

Clearly, the basic approach presents some limitations. For instance, it does not tolerate typos. Imagine that the user was looking for ”Mozilla Firefox” and typed ”fierfox”:

| Indexed Item: | Mozilla Firefox |
| User Input:   | fierfox |

As the third and fourth characters of the user input are not orderly matched in the item’s name, ”Mozilla Firefox” is a discarded item and will not be suggested to the user. But this is not what
the user was expecting for; he just mistyped two characters and it may take a while before he
notices that.

Although less severe, there is another important limitation. If the user swaps two text tokens, by
mistake, this will count as an error, no matter if they are correctly written or not. For example,
imagine that the user was looking for “Mozilla Firefox” and typed “firefox mozilla”:

<table>
<thead>
<tr>
<th>Indexed Item:</th>
<th>Mozilla Firefox</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Input:</td>
<td>firefox mozilla</td>
</tr>
</tbody>
</table>

No matches were found at all. Nonetheless, not only by mistake, writing “mozilla” after “firefox”
could also be used for item disambiguation.

3.1.2.2 A Typo-Tolerant Approach

We developed a new text prediction engine that overcomes the identified shortcomings. Our
engine extends the basic one by performing the string contention test at the token level and by
using a metric to measure the amount of difference between two text sequences.

To overcome the problem concerning the token order, we extended the basic contention test,
not only to the entire user input and items’ names, but also to the text tokens composing them.
Figure 32 demonstrates Blaze tolerating token swapping.

FIGURE 32 – BLAZE TOLERATING TOKEN SWAPPING

To overcome the problem concerning typo tolerance, we complemented the basic string
contention test with a similarity test. Our test first checks if a user’s input token is contained by
an item’s name token or vice-versa and, if not, if they are similar. Whilst our string contention
test is similar to the basic one, the similarity test uses the Levenshtein distance metric [17] to
measure the amount of difference between the two tokens. If the measuring result is less than
the length of the item’s name token, then the typo should be tolerated. This is called the typo
tolerance threshold, which was found by trial-and-error. If the typo tolerance threshold was a
fixed number instead of being the token’s length, then words smaller than that number would
always be accepted while greater words would be rejected even with a tolerable amount of
typos. The bigger the word, the more correct it would have to be.

Figure 33 depicts Blaze tolerating two typos, one for each token typed by the user.

FIGURE 33 – BLAZE TOLERATING TYPOS

Every item is weighted, when tested against user input. The item with the lowest difference
score is probably the one the user is looking for. For commands which present no typos, both the
string contention and the similarity algorithms have resembling metrics. However, the similarity
algorithm is the only one that can be used when typos are present. This allows us to make a
performance enhancement, by using the string contention test whenever a user’s input token is
contained by an item’s name token, or vice-versa, instead of only using the Levenshtein distance.
This represents a significant performance gain, as the former is an $\Theta(max(m,n))$ algorithm whilst the latter is a $\Theta(m \times n)$ one, being $m$ the size of an item’s name token and $n$ the size of a user’s input token. This is, with no doubt, the main reason for using two algorithms instead of only one.

The string contention test is similar to the one described in the last section. Our algorithm adds to the token weight the offset from the beginning of the item’s name token and the first character that is a match on the user’s input token and, also, the offset from the last character that is a match and the length of the item’s name token. Furthermore, in order to penalize character disordering, this algorithm adds extra weight to characters that are not matched in the middle of item’s name token. This extra weight is called *disorder penalty* and consists of multiplying the offset by 3. This penalty was selected through trial-and-error, as a number lower than 3 would allow words with a high disorder rate to be promoted over correct words that did not appear in the beginning of the name, and a number greater than 3 would penalize disordering too much. Therefore, if a character from the user’s input token is not matched in the item’s name token, then the string contention method returns an invalid result. Else, it returns the measurement. The string contention test can better be understood by following this diagram:

![String Contention Test Diagram](image_url)

**FIGURE 34 – STRING CONTENTION TEST DIAGRAM**

Imagine that the user typed “ffox” whilst looking for “Mozilla Firefox”. The text prediction engine would test the “ffox” token against both item’s name tokens, such as “Mozilla” and “Firefox”. The string contention algorithm would perform the following way:

<table>
<thead>
<tr>
<th>Item’s name tokens:</th>
<th>Mozilla</th>
<th>Firefox</th>
</tr>
</thead>
<tbody>
<tr>
<td>User’s input token:</td>
<td>ffox</td>
<td>ffox</td>
</tr>
</tbody>
</table>

Note that the letters in black are the ones that had no matching, the ones in blue had the same relative positions and the ones in red had different relative positions in both tokens. Therefore, being the second f’s position 3 (starting at 0 and discarding the already matched ‘f’), in token “Firefox”, and multiplying it by 3 (as penalty), the “Firefox” token would be measured in 9. The “Mozilla” token, in turn, would have an invalid measure and would be submitted to the similarity test.
If the return value of the string contention test is valid, it is assigned as the weight of the tested item’s name token. If there is no string contention between the item’s name token and the user’s input token, then the similarity test is applied. The similarity test consists of computing the difference level between the two tokens by applying them the Levenshtein distance. The result of this measure is then compared against the typo tolerance threshold. If the former is less than the latter, then the similarity test returns the Levenshtein distance. Otherwise, it returns an invalid result. Yet again, if valid, the returned value is assigned as the token’s weight. If the result is invalid, the item’s name token is discarded. The similarity test can better be understood by following this diagram:

```
FIGURE 35 – SIMILARITY TEST DIAGRAM
```

Imagine that the user was looking for “Mozilla Firefox” and typed “Faiafox”. As the text prediction engine tests “Faiafox” against both “Mozilla” and “Firefox”, both name tokens receive invalid measures from the string contention test, which forces them to be submitted to the similarity test. The similarity test would perform the following way:

```
Item's name tokens: Mozilla Firefox
User's input token: faiafox faiafox
```

Note that the letters in blue had direct matching and the ones in red correspond to typos. Trivially, the “Mozilla” token was discarded as it had 7 typos which is equal to the typo tolerance threshold (it should be less in order to be accepted). The “Firefox” token was accepted with a measure of 3 as it is the number of typos between the two tokens. As 3 is less than the tolerance threshold, which is 7, the token is accepted.

So, for every comparison between an item’s name token and a user’s input token, the final weight for each of the former tokens is measured by the following procedure, also described in FIGURE 36: First, we calculate the position of the item’s name token in the whole name. This will promote the tokens that appear first in the name. Then, identify if the user’s input token is contained by the item’s name token or vice-versa. If it is, apply the String Contention Measure to both tokens and assign it to the item’s name token weight. If not, apply the Similarity Measure to both tokens and assign it to the item’s name token weight as well.

Now, if the user’s input token was contained by the item’s name token, final weight = weight + position. If it was the other way around, final weight = weight + position + (ε weight)². This additional (ε weight)² consists of an exponential decay function, determined through trial-and-error, which adds extra weight to the item’s name if it is contained in the user’s input token. When the two are presented, this will promote the case in which the user’s input token is contained in the item’s name token over the one describing the inverse situation.
For example, imagine that the user typed “firef” while looking for “Mozilla Firefox”, but there was also an application named “Fire”:

<table>
<thead>
<tr>
<th>Item's names tokens:</th>
<th>Firefox</th>
<th>Fire</th>
</tr>
</thead>
<tbody>
<tr>
<td>User's input token:</td>
<td>firef</td>
<td>fire</td>
</tr>
</tbody>
</table>

Take into account that letters in blue were matched and the ones in red had no matching between the two tokens. If the user typed “firef” it is more probable that the wanted item is the “Firefox” instead of the “Fire” one, as the additional ‘f’ would have to be considered an error in the latter case. Therefore, in order to penalize this potential error, the additional \((e^{weight})^2\) is appended to the weighting function. The additional weight was picked taking into account that the decay function should be exponential and should vary along with the token’s original weight. The additional exponentiated 2 is needed to allow items in a situation of potential error, with low original weights, to still be adequately penalized.

On the other hand, if there was no string contention between the user’s input token and the item’s name token but both were similar, then final weight = weight + position + \((e^{weight})^3\). The additional \((e^{weight})^3\) is used to promoted string contention over string similarity, by resorting to the aforementioned decay function with a higher exponent.

**FIGURE 36 – ITEM’S NAME TOKEN WEIGHTING**
After all user’s input tokens are processed, the item is only accepted if, and only if, the number of accepted item’s name tokens is equal to the number of processed user’s input tokens. If this is the case, the item is accepted and its total weight results from the sum of the weights of all accepted tokens.

3.1.2.3 Taking Metadata into Account

We also extended our approach to take into account the metadata of an indexed item. While testing an item’s name against the user input, not only the user’s input tokens are tested with the item’s name tokens, but they are also tested with the semantic keywords associated to the item. These keywords are generated at indexation time, by indexing plugins that cooperate with the main indexing engine. Semantic keywords are treated in a similar way to name tokens, but have an extra weight, in order to promote search by name, which is still more intuitive.

Currently, Blaze has two indexing plugins, which add metadata to indexed items. One of them indexes ID3 tags from music files, adding keywords related to the song authors, composers, album, year, and so on. The other adds as keywords the contents of text files. This allows the user to search for songs by artist or album, even if these attributes are not included in the item’s name, or even by an excerpt of text from any indexed text file. Blaze can, thus, function as a simple retrieval tool.

By improving the textual prediction engine we meet requirement 2 (enhance application launcher’s typical features to a new whole level).

3.1.3 Improved User Context Awareness

Regular application launchers do not usually take user context into account. Only a few explore this field, limited only to capturing selected text, files or the address of the webpage currently being browsed. Furthermore, this information can only be used by the application launcher itself, not allowing to user to reuse it along with other commands.

To bridge this gap, we developed a software layer, named ContextLib, which can retrieve user context information from several sources. ContextLib can retrieve information about currently selected text, files and folders; the address of the directory or webpage currently being browsed; computer display settings; the SSID of connected wireless networks; provides clipboard management; and much more. This library is also responsible for monitoring user activity, detecting recurrent tasks and for suggesting automations to complete those tasks in the user’s place.

ContextLib exposes a public interface, which is already partially documented, and can be used by any application. Therefore, this library allows Blaze, Blaze’s plugins and any other application to retrieve valuable information about what the users are doing, where might they be (through wireless connections), what is currently selected, and information about the machine in which they are working on.

3.1.3.1 Using Context Information

Blaze uses context information to speed up user activity, by performing operations beforehand, which, otherwise, the user would be forced to do by hand. For instance, if the user wants to add a new command to Blaze, he just needs to select the desired executable from a Windows Explorer window, open Blaze and type “Add New Command”. Blaze will automatically detect the path of the selected file, as well as its name. The user just needs to press the Return key and a new command will be added with no effort at all. FIGURE 37 depicts the described situation.
The same happens with all other commands that involve increasing Blaze’s vocabulary. For example, if the user wants to add a new website or search engine to Blaze, he just has to browse to the desired website and type on Blaze “Add New Website”. As the reader can see on FIGURE 38, Blaze automatically detects the website’s name and address. The very same thing happens if the user wants to add a new Quick Text. He just has to select text from anywhere and type on Blaze “Add New Quick Text”, and Blaze will automatically detect that text, as shown on FIGURE 39.

FIGURE 37 – ADDING A NEW COMMAND TO BLAZE

FIGURE 38 – ADDING A NEW WEBSITE TO BLAZE
3.1.3.2 Context-Sensitive Commands

Besides detecting things beforehand, Blaze also allows the users to reuse their own contextual information within other commands. Blaze features Context Commands, which consist of text tokens that represent a specific contextual data source and that can be embedded with other Blaze Commands. When the command in which a context command was embedded is executed, the latter will be replaced by the information it represents.

Blaze provides the following seven context commands:

- ![clipboard] represents the clipboard’s contents.
- ![desktop] represents the path to the desktop folder.
- ![here] represents the path to the folder currently being browsed by the user, on the top Windows Explorer window.
- ![selected] represents the selected files or folders on the folder currently being browsed by the user. These items are comma separated.
- ![this] represents the contents currently selected by the user in any application.
- ![topapp] represents the process name of the topmost application, running on the user’s computer.
- ![url] represents the address of the webpage currently being browsed by the user. Works for Internet Explorer, Mozilla Firefox, Opera and Google Chrome.

As an example of usage of these commands, imagine that you are browsing your favorite technological blog for some fresh news, and that you found an interesting headline. You would then like to share that article with your friend John. You could go to the start menu, open your default email client, type your friend’s email, and copy and paste both the headline and the webpage’s address. Or you could just open Blaze, and type "email john@provider.net "!this" "!url"", which would make the same effect and take less time to accomplish.

FIGURE 40 perfectly depicts this situation.
In the above example, it is easy to notice that, when the email command was executed, the `!this` command was replaced by the headline, which was selected by the user on Firefox, and the `!url` command was replaced by the webpage's address. The reuse of contextual information is, with no doubt, another interesting means of speeding up user activity.

Most of the user context information is mainly extracted by using specific Windows API calls or .NET Framework objects. These allow retrieving information regarding the clipboard, wireless networks, display settings, currently selected files on windows explorer, and so forth. However, there is a special case that is of major interest and needs to be mentioned. Capturing the currently selected contents on any application (the `!this` command) was the most challenging context command to implement, mainly because it must be application independent. As we saw in subsection 2.3.2.2, the only means of exchanging data that is really application independent is the clipboard.

Therefore, in order to capture user selected contents, Blaze performs a user context snapshot, whenever it is invoked by the user. The user context snapshot consists in using the clipboard to copy the selected contents on the top application, without losing any user data that might be copied to the clipboard meanwhile. Therefore, the user context snapshot is composed by the following sequence of actions:

1. Back up clipboard contents.
2. Send a Ctrl + Insert key press to the top window, in order to copy the selected contents by the user to the clipboard.
3. Save the clipboard contents as being the user selected ones.
4. Replace the current clipboard contents with the backed up ones, in order to prevent user data loss.

The user context snapshot is also used to perform a few verifications and validations, which optimize and facilitate other context retrieval functions.

By improving user context awareness, we meet requirement 3 (be aware of user context in order to speed up user activity).
3.1.4 System Extensibility

In order to extend its features, Blaze has plugin support. There are two types of supported plugins: the ones consisting of dynamic-link libraries, written in C#, and the ones consisting of a simple text scripts, written in Python. The former method offers more flexibility than the latter and has the advantage of allowing more than one command to be added per plugin. The user has a lot of freedom with the code and can take advantage of the powerful ContexLib library. However, developing and compiling dynamic-link libraries requires a lot of knowledge about programming languages.

That's why the second method of extending Blaze's functionality exists. Blaze integrates with IronPython\(^\text{39}\), which allow any user, with basic knowledge of Python and .NET, to build a plugin using only the notepad. As IronPython is an implementation of the Python programming language running under .NET, it allows the user to use the powerful objects and methods provided by that framework. Furthermore, Blaze implicitly injects a variable named ".user_context" in the execution scope of each IronPython script run by the user, which allow him to instantly access ContexLib without having to include any reference to it on the script.

As an example, take the "What is my IP" command, featured by Blaze. This command retrieves the user's external IP address, which is the IP address that was assigned to him by his ISP and not the one assigned to him by his router (in the case he is behind one). This command consists of an IronPython plugin for Blaze, which uses ContexLib to retrieve the external IP. The plugin also asks the users if they want the IP address to be copied to the clipboard. Yet again, this function is provided by ContexLib. The plugin's code follows bellow, in order to clarify how the ContexLib can be easily used inside an IronPython script:

```python
ip = _user_context.GetExternalIpAddress()
message = "Your current external IP address is: " + ip + "\n\n" + "Would you like to copy it to the clipboard?"
res = System.Windows.Forms.MessageBox.Show(message, "What is my IP?", 
    System.Windows.Forms.MessageBoxButtons.YesNo, 
    System.Windows.Forms.MessageBoxIcon.Information)

if res == System.Windows.Forms.DialogResult.Yes:
    _user_context.SetClipboardText(ip)
```

As the reader can see, the user only needs to reference the ".user_context" object in order to use all ContexLib's functionality. As users do not have to compile any code, they can edit scripts or build new ones with low effort, just by using the notepad and being a little proficient with Python and .NET. All the scripts are located on a specific folder and, each created script is instantly detect by Blaze and added as a new command, being thenceforth available to the user.

By offering these two mechanisms of extending Blaze's functionality, we meet requirement 4 (the system must be easily extensible).

3.2 BLAZE AS AN AUTOMATION AGENT

The underlying innovation behind Blaze resides in three main features:

1. The ability to monitor all user actions, operating system widely.

2. The ability to infer repetitive patterns in the user activity, thereby identifying recurrent tasks.
3. The ability to build an automation to complete a recurrent task and suggest accomplishing it in the user’s place.

These allow Blaze to detect and automate almost every repetitive task that may arise from everyday usage.

3.2.1 Overview

Just like APE, Blaze includes three agents: the Observer, the Apprentice and the Assistant. The Observer is responsible for monitoring user actions, at the operating system level. In other words, its job is to log every action the user performs in any possible application. To attain this, the Observer relies on several Windows API and .NET functions. As a matter of fact, the main core of user activity logging is composed by a file-system watcher and keyboard and mouse system-wide hooks. The file-system watcher is responsible for detecting when a user is performing an operation over a file or folder and which operation is that. The keyboard and mouse hooks are responsible for detecting what keys the user pressed and where have he performed mouse clicks, double-clicks, drags, and so on.

When the Observer detects that the user is performing an action, it creates a domain object called user action. User actions, in turn, can be combined amongst them to compose more elaborated user actions. For instance, if the user pressed the H, E, L, L and O keys, then these actions can be compressed in a single one representing "Type «hello»". All detected actions are stored in a list named user’s actions history.

The Apprentice’s duty is to examine the user’s actions history and detect repetitive patterns of performed actions. To attain this, a data mining algorithm is employed, which extracts sequences of repetitive actions, named repetitions, and stores them on a list named repetitions list.

The Assistant, in turn, is responsible for examining the repetitions list and build an automation suitable to complete the detected repetitive task. The automation is, therefore, suggested to the user, in a non intrusive way, by displaying a special icon in the task bar (FIGURE 41) and a suggestion button on Blaze’s main interface, if it happens to be invoked at that time (FIGURE 42). These do not interrupt the user in any way possible: there are no popup windows neither objects in motion on the screen.

The user can either click the suggestion button or double tap the CapsLock key, which will show up the Blaze Assistant window (FIGURE 43), or just ignore them and keep working. This window displays comprehensive narratives describing each suggested automation, and allows the user
to pick one to be executed. The Blaze Assistant also allows the user to parameterize the execution, by changing the number of iterations that will be performed and the execution speed. Furthermore, it allows the user to edit or export an automation or even create a new one.

FIGURE 43 – BLAZE ASSISTANT

Once a suggestion was accepted and the automation was executed, the user can reuse the performed automation by means of two commands, provided by Blaze: the continue and the redo commands. The former allows the user to continue the last automation for a specified amount of iterations. The latter, allows the user to reproduce the automation, from the beginning, for the specified amount of iterations. An example of this can be found below, in section 3.2.2.1.

By monitoring user activity and suggesting automations capable of completing recurrent tasks in the user’s place, Blaze meets requirements 5 and 6 (the system must detect repetitions ad offer to auto-complete the task for the user, without disturbing him).

A more detailed explanation of how these three agents cooperate and work together with Blaze will be provided in section 4.

3.2.2  A Few Practical Examples

There are a few noteworthy practical examples which demonstrate Blaze in action. These examples depict recurrent tasks that may emerge from everyday usage.

3.2.2.1 Automating Recurrent Tasks on Notepad

Imagine that you follow a TV series and that you use to jot some notes about every season. You would, therefore, like to prepare a document on notepad where you would have each season’s number and a space to write a review, such as follows:

```plaintext
### Season 1 ###
Review:

### Season 2 ###
Review:

### Season 3 ###
Review:

...

### Season 15 ###
Review:
```
Doing this by hand would take quite some time. With Blaze, you just have to write the first three cases and he will suggest an automation to complete the task for you, as depicted in FIGURE 44.

![Figure 44 - Blaze automating a recurrent task on Notepad](image)

Now imagine that you would like to add 10 more seasons to document, but you do not want to do it by hand. As you just added the previous 15 with Blaze, you just have to invoke it and type “continue 10”, in order for it to continue the last automation for 10 more iterations (from “Season 16” to “Season 25”).

### 3.2.2.2 Automating Recurrent Tasks on Excel

Now imagine that you have an Excel book with 5 sheets, where each one of them has a table containing the results of some medicine experiments, as shown in FIGURE 45. You would like, therefore, to highlight the lines beginning with “Medicine”, in order to draw the attention of your colleagues to those results.

To achieve that, you would like to apply bold to every line beginning with “Medicine”, on each one of the five sheets. Doing it by hand would take quite a long time. So, with Blaze, you just have to perform it 4 to 5 times, and it will do the rest, like depicted in FIGURE 46. The reader might notice that, for this example, the user have to perform more than 3 iterations. The reason for that is to increase the precision of the reproduced mouse clicks. The more mouse clicks are sampled to Blaze, the more precise the reproduced ones will be.
Note that the automation suggested by Blaze is for one sheet only. So, in order to reproduce it for other sheets, you must select each one of them, open Blaze and type “redo 12”, in order to redo the last performed automation for 12 iterations.

3.2.2.3 Automating Recurrent Tasks on the File-System

Imagine a similar example to the one provided in the Introduction of this document. Imagine that you took 50 photos with your digital camera on a summer party and that, when you downloaded them to your computer, you noticed that your camera assigned strange names to them, just like in FIGURE 47.
You would like to rename them with meaningful names, such as “Summer Vacations 01.jpg”, “Summer Vacations 02.jpg”, and so forth. Doing this by hand would take 10 minutes and learning how to use a dedicated tool would probably take even more. With Blaze, you have just to rename the first 3 files, respecting their sequential numeric order, and Blaze will figure out what to do next, as show in FIGURE 48.

Blaze introduces the *Adaptive Macro* concept. Adaptive Macros are macros that can adapt to new situations, as those in which the involved window’s position is not the same as when the macro was first recorded, which might induce reproduced mouse clicks miss their targets.

Blaze allows users to easily record a macro, by just starting and stopping a recording process. The recording process can be started by typing the `record macro` command and stopped by simply invoking Blaze. By doing so, a form will show up where the user may give a name to the newly created macro. Therefore, the user just has to type the macro’s name in order to execute it.

Every created macro is stored on an IronPython script, comprising all actions the user performed during the recording period. Each recorded user action includes contextual information concerning the window in which it was detected. For example, for actions regarding file-system operations, the path of the folder in which it occurred is stored, as well as the path of the affected folder items; for actions related to the keyboard or the mouse, such information can be the window’s title, process name, process id, position and dimension. This allows, for instance, a reproduced mouse clicks to recalculate their relative position, by using pure
arithmetic, and adapt to the window’s position, even if this position is different from when the action was first recorded.

As these actions can adapt to new situations, they are named *adaptive actions*. These actions are recorded to an IronPython script, in plain text, and can be edited with the notepad. The script also presents a variable that allows the user to parameterize the execution speed. By comprising adaptive actions and providing variables to parameterize their execution, these scripts are called *adaptive macros*.

The adaptive action and macro concepts led us to meet requirement 7 (Synthesized automations should be able to adapt to new situations).

### 3.3 BLAZE FROM AN ARCHITECTURAL POINT OF VIEW

Blaze’s architecture can be seen, from a layered point of view, as shown in Figure 49.

![Figure 49 - Blaze's Layer Architecture](image)

From the bottom to the top we have:

#### 3.3.1 System Abstraction

This layer wraps most of Windows API method calls used by Blaze and by its plugins. These methods are provided by several dynamic-link libraries, such as *user32.dll*, *kernel32.dll* or *shell32.dll*. Examples of methods wrapped by this layer are the ones responsible for retrieving the keyboard and mouse state, for manipulating windows, or related to keyboard and mouse system-wide hooks.

#### 3.3.2 User Context

This layer is composed by the ContextLib itself. This library is responsible for capturing and manipulating user context, as well as monitoring every user action and inferring automations to complete recurrent tasks.

ContexLib is mainly composed by three categories of objects and methods, as show in Figure 50:

**Standalone Retrieval** – Methods responsible for standalone context information retrieval, such as retrieving a wireless network's SSID or clipboard contents.
Pre and Post Focus – Methods that are executed once Blaze gets and loses focus. An example of such a method is the one that takes a snapshot to the user context, which is executed right before Blaze gets the user focus.

Automation – This category is composed by the Observer, the Apprentice and the Assistant, which are the agents responsible for monitoring user activity, detecting repetitive patterns of actions, and suggesting automations.

This layer directly uses the System Abstraction one. Both the Standalone Retrieval and the Pre and Post Focus methods use several .NET and Windows API calls to accomplish their job. The Automation agents, however, resort to file-system watcher objects and keyboard and mouse system-wide hooks to capture every user actions regarding file-system and computer peripherals manipulation.

3.3.3 Indexer

The Indexer layer is composed by a standalone executable, named BlazeIndexer. BlazeIndexer is responsible for indexing all files that might be relevant to the user, such as the ones contained in the Star Menu folder and the Recent Items folder.

BlazeIndexer builds an index containing every indexed file, which is subsequently used by Blaze. For each indexed file, BlazeIndexer stores in the index its path, icon and a set of associated semantic keywords. BlazeIndexer supports plugins, through dynamic-link libraries, which allow additional semantic keywords to be added for each indexed file.

3.3.4 Scripting Engine

For a given sequence of user actions, the Scripting Engine is responsible for building an IronPython script which will sequentially executed each one of those actions. The action object used in the scripts is exactly the same used internally by the automation agents, which allow them to adapt to new situations on both environments. Furthermore, each built script contains several comments, to make it easier to read and edit, and also a speed variable, which allow the user to control the reproduction speed.

This layer is directly used by the User Context, allowing macros and suggested automations to be exported as scripts. Furthermore, as the IronPython engine is embedded with the Scripting Engine, this layer is also responsible for reproducing scripts generated by Blaze.
3.3.5 Interpreter

The Interpreter is responsible for interpreting user commands. Every 15 minutes, the Interpreter asks BlazeIndexer to build a new index. Once the index is built, the Interpreter adds it to its own private index, which also contains the commands provided by other plugins. Hence, every time the user types a letter, the interpreter submits the user input and the index to the Textual Prediction Engine, which will weigh every item and return the 10 most approximated to the user request.

Interpreter’s functionality can be extended by dynamic-link libraries, which allow several commands to be added at once, or by IronPython scripts, which only add a command per script but are easier to create and edit.

3.3.6 Interface

Blaze Interface is composed by two elements:

**Main Interface** – Which is the one where the user types his commands. This interface element is responsible for transmitting the user input to the Interpreter.

**Blaze Assistant** – This interface element is responsible for displaying suggested automations to the user, inferred by the Assistant agent. There is a strong dependency between Blaze Assistant and the three automation agents, which is depicted in FIGURE 51. Each edge of the diagram can be read as “Notifies”.

---

**FIGURE 51 – COMMUNICATION PROCEDURE BETWEEN BLAZE ASSISTANT AND THE AUTOMATION AGENTS**

- **The Observer**
  - Captures low-level user actions.
  - Refines low-level actions into high-level actions.
  - Establishes generalizations between similar actions.

- **The Apprentice**
  - Identifies repetitive patterns of actions in the user’s actions history.

- **The Assistant**
  - Builds automations capable of completing repetitive tasks, based on repetitions identified by the Apprentice and generalizations built by the Observer.

- **Blaze Assistant (Interface)**
  - Provides the user with controls to parameterize their execution.
  - Displays comprehensive narratives describing the suggested automations.
4 AUTOMATING THE USER EXPERIENCE

In order to develop a system capable of automating recurrent tasks on a computer, it must have the ability to learn a new behavior from a sampled set of actions, which demonstrates a concrete example of a repetitive task. This is the basis for a typical Programming by Example system, which embraces the macro recorder metaphor, just like SMARTedit [15] or Creo [7]. However, these systems require the user to explicitly start and stop the recording process, which generates a sample of demonstrative actions. The mere fact that the user must be directly involved in the programming process makes the system directed to proficient users only, which, nowadays, are only a small part of all computer users. Also, having to start and stop the recording process may end up to be a nuisance, as users cannot commit any errors and must be aware of where the repetitive task starts and ends.

To overcome these issues, there is another technique, named Implicit Programming by Example, which avoids the macro recorder metaphor by detecting repetitive patterns of user actions without user intervention, and learning when to suggest an automation, capable of completing the repetitive task. Our approach is based on this technique, consisting of three automation agents with three distinct tasks: the Observer, which is responsible for capturing every action the user performs in the operating system, including in any application he might be using; the Apprentice, whose duty is to identify repetitive patterns in user’s actions history; and the Assistant, which is responsible for learning when to suggest an automation capable of completing a recurrent task. Our solution is architecturally similar to APE [28], however, instead of only monitoring a single application, Blaze monitors the whole operating system, thus covering all applications used by the user.

The next subsections will give a detailed description about how each one of the automation agents work.

4.1 THE OBSERVER

The Observer is responsible for monitoring user activity. More precisely, he is responsible for two tasks:

1. Capturing every action the user performs in the operating system and any application running under it;
2. Identifying relationships among captured actions and assign to them generalizations describing their similarity.

4.1.1 Monitoring User Activity

In order to develop a solution capable of carrying out these tasks, we had to face two major engineering challenges:

1. Covering something as wide as the whole operating system, including the broad range of applications that might be used under it, is quite a hard task. Mainly because not all applications are developed under the same rules and using the same standards. If the user is dragging the mouse inside an application, there is no way of knowing if he is selecting text, dragging a tab or drawing a line.
2. How to convert the sparse data provided from different sources into reliable information that can be, subsequently, used by the other two agents?
The first challenge emerges from the fact that we do not have at our disposal a database containing information about every domain object. APE and SMARTedit use domain information to identify which objects are being manipulated by the user and which actions can be performed over those objects. We, however, lack of that information as there are too many different applications for Microsoft Windows which are developed using distinct standards. It is not viable to build a domain database with hardcoded methods for every possible application as it would consume too much time and would be too hard to maintain.

Thereby, we opted for defining four main data sources to capture user's actions in most occasions. These sources are:

1. **Mouse system-wide hooks** – Capture every action the user performs with the mouse, such as mouse clicks, double clicks or wheel spins;
2. **Keyboard system-wide hooks** – Capture every key press the user performs with the keyboard;
3. **File-system watcher** – Monitors a specific directory and captures every action the user performs over a file or folder, such as creating, renaming, moving or deleting;
4. **User context** – Complements the other three sources with information regarding the user context, such as the directory that the user is currently browsing or the window name, process and size where a mouse click occurred.

The first three data sources are continuously monitoring the user and generate a domain object, named *User Action*, for each action the user performs. However, these sources provide user actions with different granularity levels, which bring up the second engineering challenge. For instance, if the user types "hello" in a text processor, the keyboard system wide hook would generate 5 *key press* user actions:

- Press "H"
- Press "E"
- Press "L"
- Press "L"
- Press "O"

However, if the user renamed a file, only one action would be generated:

 rename "C:\MSN" to "C:\WLM"

The main problem here is that the user typing "hello" should be represented by a single action and not by five. Being represented by as much as 5 would confer a greater importance to that action and make it hard to consider for a repetition detection algorithm. Therefore, in order to solve the granularity issue, we developed a *User Action Compression Algorithm* which is able to merge *low-level user actions*, of a certain type, into *high-level user actions*. This algorithm checks if a newly captured user action can be merged with any of the actions already contained in the *user's actions history* and, if it does, it merges them, thus generating a new high-level user action. For example, imagine that the user types "hello" in a text processor. The five generated *key press* actions are compressed into a single *type «hello»* user action, like depicted in Table 11. Furthermore, the algorithm also allows high-level user actions to be merged between them in order to generate more elaborated user actions.
Taking into account each one of the monitoring data sources, the Observer is able to detect the following actions:

**Mouse system wide hooks:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Low-Level User Actions</th>
<th>Description</th>
<th>High-Level User Actions</th>
<th>Results from</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mouse Down</td>
<td></td>
<td>The user presses a mouse button.</td>
<td>Mouse Click</td>
<td>Mouse Down + Mouse Up, in the same location.</td>
</tr>
<tr>
<td>Mouse Up</td>
<td></td>
<td>The user releases a mouse button.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mouse Wheel Spin</td>
<td></td>
<td>The user spins the mouse wheel. (can be merged with other actions of the same type)</td>
<td>Mouse Double Click</td>
<td>Mouse Click + Mouse Click within 500 milliseconds.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mouse Drag</td>
<td>Mouse Down + Mouse Up, with different locations.</td>
</tr>
</tbody>
</table>

**Keyboard system wide hooks:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Low-Level User Actions</th>
<th>Description</th>
<th>High-Level User Actions</th>
<th>Results from</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Down</td>
<td></td>
<td>The user presses a key.</td>
<td>Key Press</td>
<td>Key Down + Key Up</td>
</tr>
<tr>
<td>Key Up</td>
<td></td>
<td>The user releases a key.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type Text</td>
<td></td>
<td>The user types a sequence of characters.</td>
<td>Key Press + Key Press or Type Text + Key Press</td>
<td></td>
</tr>
</tbody>
</table>

**File-system watcher:**

<table>
<thead>
<tr>
<th>Name</th>
<th>User Actions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create File</td>
<td>The user creates a file or folder.</td>
<td></td>
</tr>
<tr>
<td>Delete File</td>
<td>The user deletes a file or folder.</td>
<td></td>
</tr>
<tr>
<td>Move File</td>
<td>The user moves a file or folder.</td>
<td></td>
</tr>
<tr>
<td>Rename File</td>
<td>The user renames a file or folder.</td>
<td></td>
</tr>
</tbody>
</table>

The user context also plays an important role in action monitoring. User context supplies valuable information regarding which path the user is currently browsing in the top Windows

<table>
<thead>
<tr>
<th>Capture User Action</th>
<th>Compression Operation</th>
<th>Compressed User Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Press &quot;H&quot;</td>
<td>Press &quot;H&quot;</td>
<td>Press &quot;H&quot;</td>
</tr>
<tr>
<td>Press &quot;E&quot;</td>
<td>Press &quot;H&quot; + Press &quot;E&quot;</td>
<td>Type &quot;he&quot;</td>
</tr>
<tr>
<td>Press &quot;L&quot;</td>
<td>Type &quot;he&quot; + Press &quot;L&quot;</td>
<td>Type &quot;hel&quot;</td>
</tr>
<tr>
<td>Press &quot;L&quot;</td>
<td>Type &quot;hel&quot; + Press &quot;L&quot;</td>
<td>Type &quot;hell&quot;</td>
</tr>
<tr>
<td>Press &quot;O&quot;</td>
<td>Type &quot;hell&quot; + Press &quot;O&quot;</td>
<td>Type &quot;hello&quot;</td>
</tr>
</tbody>
</table>
Explorer window. This path is used by the file-system watcher in order to monitor user activity. User context is also used to complement the data captured by keyboard and mouse hooks. To every single action captured by these sources is associated information regarding the application in which it occurred, such as the process name and id, and the window’s name, class, position and dimension.

Every user action is contained in the user's actions history. Each time an action is captured and processed by the Observer, it notifies the Apprentice, which, in turn, examines the user's actions history in order to identify repetitive patterns. The Observer maintains in memory the last 20 performed actions, discarding the older one whenever a new one is detected. The number 20 was chosen through trial-and-error, as less than 20 actions were too few and did not allow long recurrent tasks to be detected. More than 20 actions allow longer tasks to be detected but had a significant impact on computer's performance, which was not desirable. Along with the actions compression algorithm, 20 actions are enough for most recurrent tasks.

Also, every action has a life span of 40 seconds, counting from the moment in which it is created or merged. When this time is elapsed, the action is discarded. 40 seconds was chosen because a greater value, such as 1 minute or even 2 allowed old actions to be considered in recurrent tasks, which is not desirable in many cases. On the other hand, a value smaller than 40 seconds implies that Blaze will not detect recurrent tasks if the user performs them too slowly, as first actions would be discarded too soon.

4.1.2 Generalizing

Whenever a new action is created, it is assigned an id and two actions may have the same id if, and only if, they are similar. Two actions are said to be similar if, and only if, it is possible to create a generalization between them. A generalization is composed by an expression, which is a string representing the parameters for subsequent iterations of an action. A generalization is also composed by a set of functions, which describe a particular behavior. These functions are represented in the expression by a unique symbol, such as §1, §2, §3, and so forth. By replacing that symbol with the function’s result, passing as a parameter a future iteration number, we get the actual expression for that iteration. As expressions are used to parameterize actions, it is trivial to build future user actions using this mechanism. Also, it is easy to note that to each id is associated a set of generalizations.

4.1.2.1 Creating and Merging Generalizations

As a new action is added to the list, the Observer looks up for a similar older action and creates a generalization. If there are no previous generalizations bound to the older action’s id, then the new generalization is validated and the new action gets the elder’s id. Otherwise, the new generalizations and the ones already bound to that id have to be merged. If the merging between those generalizations is successful, the new action gets the id from the old one. If not, then a new id is assigned to the new action.

To clarify the generalization mechanics, follow the next example:

Observed Actions:

[ID: 13] Type "Hello 1"
[ID: 13] Type "Hello 2"
Generalizations:

[ID: 13] Hello \( \$1 \), \( \$1(n+1) = \$1(n) + 1, \$1(n) = 2 \);
[ID: 14] repeat;

In this case, the first and third actions have been generalized, as well as the second and the fourth ones. In the former case, the resulting generalization contains the expression "Example \( \$1 \)" and the function "\( \$1(n+1) = \$1(n) + 1, \$1(n) = 2 \)" where "\( \$n \)" is the current iteration number. This generalization expresses a sequential numeric relationship between the two actions, and its expression and function allow the Assistant to deduce the next iterations. The reader can easily notice, with this example, that generalizations are the key for inferring user’s future iterations. In the latter case, the resulting generalization only features the “repeat” expression, because “repeating” is the only relation that can be established between two key presses of the same key.

To simplify, only one generalization was shown per id, in the last example. However, in the real system, several generalizations may be created. Generalizations are created, orderly, following a predictive criterion. For example, while having two file renaming actions, a generalization that implies that there is a sequential numeric relationship between both old names and also a sequential numeric relationship between both new names, will have a higher probability of being right than one that implies that both old names just contain a common set of text tokens. This latter case would have a higher probability of being a misunderstanding. There are several kinds of generalizations, taking into account each type of actions to which they apply to, as show in Table 15.

<table>
<thead>
<tr>
<th>Generalization Name</th>
<th>Corresponding Action Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Generalization</td>
<td>Key Press</td>
</tr>
<tr>
<td>Mouse Generalization</td>
<td>Mouse Click, Mouse Double Click and Mouse Wheel Spin.</td>
</tr>
<tr>
<td>Mouse Drag Generalization</td>
<td>Mouse Drag</td>
</tr>
<tr>
<td>Text Generalization</td>
<td>Type Text</td>
</tr>
<tr>
<td>File Create Generalization</td>
<td>Create File</td>
</tr>
<tr>
<td>File Delete Generalization</td>
<td>Delete File</td>
</tr>
<tr>
<td>File Move Generalization</td>
<td>Move File</td>
</tr>
<tr>
<td>File Rename Generalization</td>
<td>Rename File</td>
</tr>
</tbody>
</table>

**TABLE 15 – LIST OF GENERALIZATIONS CREATED BY THE OBSERVER**

Each generalization is built by comparing data from both the new action and an older action, and checking which functions are most appropriate to describe the relationship between those actions. As depicted in Table 16, there are several types of functions. However, it is worthy to remark that a generalization may only be composed by functions of the same type. These functions are built based on the Difference algorithm\(^40\) \([23]\), regular expression validation and string and arithmetic operations.

For instance, the Constant File Name Function uses the Difference algorithm to detect which tokens of text are common among two File-system actions. Imagine that the user deletes the following files from a folder:

HW15 is awesome - \texttt{XPTO 2008-2009.txt}
player CK27 has cooking skillz - \texttt{xpto 2008-2009.txt}
xPto team arrangement \texttt{2008-2009.txt}

\(^{40}\) http://www.mathertel.de/Diff/ (Last visited in: August 30\(^{th}\), 2009)
By splitting the file names into tokens and identifying which are common to all deleted files, the composed generalization would be “delete $\$ (x).txt”, in which $\$ (x)$ is a Constant Text Function that describes “all files containing xpto, 2008 and 2009”.

The Sequential Int Function and Sequential Char Functions are quite similar and worthy of being mentioned. Imagine that the user typed on notepad:

Type “Hello 1”
Type “Hello 2”
Type “Hello 3”

The Observer uses the Difference algorithm to identify which portions of the typed strings changed and if they are integers, by using a regular expression. Therefore, to the integer value from the most recent generalization is subtracted the previous’ one, resulting in the increment that defines the sequence. Therefore, the generalization “type Hello $\$ (x)” is composed, in which $\$ (x)$ is a Sequential Int Function that describes the numeric sequence 4, 5, 6, 7, and so forth.

The Constant Text, Constant File Name and Constant File Extension functions are built by comparing text tokens between two actions and checking if they are or not the same.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Corresponding Generalization Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Text Function</td>
<td>Represents a piece of text which is repeated over and over again.</td>
<td>Text Generalization</td>
</tr>
<tr>
<td>Sequential Int Function</td>
<td>Represents a numerical sequence.</td>
<td>Text Generalization, Mouse Generalization, All File Generalizations</td>
</tr>
<tr>
<td>Sequential Char Function</td>
<td>Represents an alphabetic sequence.</td>
<td>Text Generalization, All File Generalizations</td>
</tr>
<tr>
<td>File Diff Function</td>
<td>Represents a set of common Diff matches between file names.</td>
<td>File Rename Generalization</td>
</tr>
<tr>
<td>Constant File Name</td>
<td>Represents a set of common text tokens between file names.</td>
<td>All File Generalizations</td>
</tr>
<tr>
<td>Constant File Extension</td>
<td>Represents a set of common extension between file names.</td>
<td>All File Generalizations</td>
</tr>
</tbody>
</table>

4.1.2.2 Generalization Storage and Sorting Order

All generalizations are stored by the Observer, in a hashtable, in which the key is an action id and the value is a list of generalizations. This hashtable is used by the Assistant in order to generate actions capable of completing the user’s recurrent task.

To each generalization is assigned a score which describes that generalization’s priority. This score is used to sort generalizations inside each list, present in the Observer’s hashtable. The ones with higher scores come first and the ones with lower scores come last.

As each generalization may only comprise functions of a single type, the score is equal to the priority level associated to that type. The priorities associated to each function type are depicted in Table 17, and were defined through trial-and-error. By performing a few informal tests with users, we found which functions where more prone to be misunderstandings and which were more commonly used.
Imagining that the user typed "Hello 1", "Hello 2" and "Hello 3", two generalizations would be associated to these actions: one treating the changing portion of the string as a numeric sequence, which is composed by Sequential Int Functions, and another one treating it an alphanumeric sequence, which is composed by Sequential Char Functions. Therefore, the generalization describing an alphanumeric sequence will have a score of 4 while the one describing a numeric sequence will have a score of 5. Therefore, the latter will appear first in the list, being the generalization with the higher probability of being what the user is looking for.

This ordering is crucial for the Assistant to build comprehensive narratives and automations capable of completing a repetitive task. This will be addressed in section 4.3.

### 4.2 THE APPRENTICE

Now that we already have an agent able to capture every user action, we need one to detect repetitive tasks. In typical Programming by Example approaches, a repetitive task is defined by a sequence of actions that are recorded by the user. However, this requires the users to be directly involved in the programming procedure. They must be aware of the macro concept and know precisely where an iteration of the repetitive task starts and ends. However, users are not always aware of that.

Our Implicit Programming by Example approach is able to identify recurrent tasks without user intervention. This job is done by the Apprentice, which is responsible for indentifying repetitive patterns of actions in the user's actions history, provided by the Observer.

#### 4.2.1 Learning What to Automate

As our system is not intrusive, it cannot interrupt the users’ work to ask their approval upon identifying positive or negative examples of iterations of the repetitive task. That being, it is impossible to simply label an example as positive or negative. All examples are possible positive examples. Therefore, we adopted a different approach to identify which examples are positive examples of a user task, worthy of being automated. As mentioned by Mahmud [20] in his report, a good way to compensate for lack of negative examples is to have repeated occurrences of the positive ones. This is, indeed, what we really need. Actions that the user repeats several times have an enormous potential to integrate a repetitive task, suitable to be automated. Hence, we need a fast way to detect repetitive sequences in the user’s actions history.

Assuming that all actions have an id and the user’s actions history have a finite size of 20 actions, we know that we will not have more than 20 different ids. Therefore, action ids can be seen as letters from a small alphabet, which allow us to treat this problem as a string search one.

The search problem described above can be seen as the Longest Repeated Substring (LRS) problem. We want to find the longest non overlapping substring of ids that occurs in the user’s

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Text Function</td>
<td>1</td>
</tr>
<tr>
<td>Constant File Extension</td>
<td>2</td>
</tr>
<tr>
<td>Constant File Name</td>
<td>3</td>
</tr>
<tr>
<td>Sequential Char Function</td>
<td>4</td>
</tr>
<tr>
<td>Sequential Int Function</td>
<td>5</td>
</tr>
<tr>
<td>File Diff Function</td>
<td>6</td>
</tr>
</tbody>
</table>

**TABLE 17 – FUNCTION PRIORITY LIST (6=HIGHEST PRIORITY)**
actions history. Thereby, the best solution to solve this problem is to build a suffix-tree\(^{41}\) and find all of its deepest internal nodes. The edge leading to one of these nodes represents a common prefix of suffixes of the input string. Therefore, we keep track of all common prefixes of suffixes, picking the longest as the *longest non overlapping repeated substring*. This approach allows us to detect more repetitions and offer a broader range of options to the user. Although the KMR algorithm would be suitable for the task, the suffix-tree approach is more efficient, as it solves the problem in linear time \(O(n)\), whereas KMR solves in \(O(n \log n)\)-time.

![Suffix Tree](http://www.allisons.org/ll/AlgDS/Tree/Suffix/)

**FIGURE 52 – THE SUFFIX-TREE REPRESENTING “ABCABD”**

Figure 52 depicts an example of a suffix-tree representing the string “abcabd”. Each edge is labeled with a prefix string and every path from the root node to a leaf corresponds to a possible suffix of that string. Note that squares represent leaf nodes, while circles represent internal nodes.

Now, imagine that a, b, c and d are distinct user action ids, assigned to the following actions:

- [id: a] Type "Hello 1"
- [id: b] Press Return
- [id: c] Press PrintScreen
- [id: a] Type "Hello 2"
- [id: b] Press Return
- [id: d] Press Alt+Tab

The user’s action history contains the sequence “abcabd”, as depicted in Figure 52. To find the longest repeated non overlapping substring, a depth-first search must be performed, going as deep as the deepest internal node in each path. The edge leading to one of these nodes represents the deepest common prefix of suffixes on a specific path, and each deepest prefix is suitable to be the longest repeated non overlapping substring. Ergo, it is trivial to find out that both “ab” and “b” are common prefixes of suffixes, as the former is depicted by the edge leading to node 7, which is the deepest internal one on his path, and the latter is described by the edge leading to internal node 8, which is also the deepest on his path. Accordingly, being both “ab” and “b” suitable, “ab” is chosen as the best choice, as it is the longest. This algorithm, thereafter, proves to be ideal to find the longest repeated non overlapping subsequence of user action ids. Furthermore, it allows Blaze to detect repetitions even if these do not occur consecutively in the actions’ history.

\(^{41}\) [http://www.allisons.org/ll/AlgDS/Tree/Suffix/](http://www.allisons.org/ll/AlgDS/Tree/Suffix/) (Last visited in: June 31\(^{th}\), 2009)
The Apprentice keeps a list of repetitions, sorted by descending order of length and date, which along with the generalizations provided by the Observer, allow the Assistant to compose consistent narratives, describing repetitive task automations.

4.2.2 Supported Repetition Types

The repetition detection algorithm is tightly dependent on the generalization system, as two actions only get the same id if they are similar. Currently, our repetition detection algorithm, along with the generalization system, is able to detect the following types of repetitions:

**Constant**: all iterations from one repetition are identical. For instance, repeatedly typing "test".

**Loop**: all iterations comprise the same actions but are repeated over a set of objects. For example, deleting all text files in a folder.

**Conditional loop**: iterations are identical but are repeated over a set of objects exhibiting a certain property. For instance, moving all files, containing in the name the token "mary", from one folder to another.

**Variable loop**: the iterations are repeated over a set of objects and depend on a property of the object. For example, a case where a numeric sequence is present, such as typing "test 1", "test 2", "test 3", ..., "test 40", or even renaming all files "File 01.txt", "File 02.txt", "File 03.txt", ..., to "Example 01.txt", "Example 02.txt", "Example 03.txt", ... and so forth.

Due to our generalization system, our approach is not able to detect nested loops yet. However, some improvements might be made in the future in order to get nested loops to be detected.

As the reader can notice, taking into account the repetition types described in section 2.3.1.4, our approach additionally detects conditional and variable loops, which was not achieved by any other Implicit Programming by Example approach.

4.3 THE ASSISTANT

Now that we already have an agent to monitor the user and another one that knows what to automate, there is the need for one that simply knows when to suggest. The suggestion must be non intrusive, as the users should not be unduly interrupted. Therefore, the Assistant is the one responsible for this task, using the generalizations built by the Observer and the recurrent sequences of actions identified by the Apprentice, to build comprehensive narratives and automations capable of completing recurrent tasks in the user’s place.

4.3.1 Learning When to Suggest

Whenever an action is detected by the Observer and the Apprentice detects a repetitive task, the Assistant is notified. Once this happens, it composes an automation, capable of completing the task, and suggests it to the user. In order to notify the user, whenever a repetitive task is detected, Blaze's icon in the system tray is lit up and an exclamation point shows in Blaze's main interface. The user can just double tap the CapsLock key or press the exclamation point, in order to check the suggestions, or simply ignore them if he does not want to be bothered. Hence, there is nothing getting in his way.
If the user accepts to check the suggestions, a new form shows up, named Blaze Assistant, which displays a narrative for the most probable automation. There may be one or several suggestions, which the user can cycle through using the appropriate controls. The user can also change the amount of iterations to be performed. Thereby, changing both the number of iterations and the currently selected automation, will cause the displayed narrative to be updated, in order to inform the user of the changes. Shortcut keys for these actions are provided, for a more efficient usage by expert users. Also, the most probable automation, the one that is first displayed to the user, is the longest and newest one. Automations are sorted descendingly by the amount of actions composing their repetitions (length) and, to play off a tie, the newest automation comes first.

### 4.3.1.1 Composing Narratives

The narrative engine was revised a few times, as it was validated with informal usability tests along with the users. In the first versions of Blaze Assistant, the produced narratives were in plaintext, which made it quite hard to read, as shown in Figure 53.

![FIGURE 53 – FIRST VERSION OF BLAZE ASSISTANT](image)

The users complained that it required too much effort to read and try to understand what was going to happen. Therefore, after two major revisions, we created a better generation engine, which produces more easy-to-read narratives. Current narratives expose text in a more structured way, displaying each action in a separated line and using RTF text formatting features to highlight important terms. This is depicted in Figure 54.

![FIGURE 54 – BLAZE ASSISTANT REVISED](image)

Each narrative is composed by the sub-narratives of the actions composing the automation. Each sub-narrative is composed taking into account the most probable generalization for the concerned action, according to the generalization ordering described in 4.1.2.2.
To each generalization type, function type and action type is associated a text token, which changes according to the parameterization inherent to the concerned object. Associated to these tokens are rtf tags, which allow Blaze Assistant to perform proper text formatting. Therefore, a sub-narrative consists of an aggregation of these tokens.

This system allows for sophisticated information visualization. Blaze Assistant is able to detect important visual elements and highlight them, in order to better inform the user. For example, when there are numerical sequences, the changing number is bolded (Figure 54); when there are common text tokens to several files, these are mentioned to the user and also bolded (Figure 57); and if there are portions of text that change, Blaze Assistant marks with red the missing text portions with green added text portions (Figure 55). This last marking system is very similar to well-known applications that use the Difference algorithm.

![Blaze Assistant showing color text formatting](image)

**FIGURE 55 – BLAZE ASSISTANT SHOWING COLOR TEXT FORMATTING**

### 4.3.1.2 Creating the Actions to Complete the Task

Imagine that the following repetition was detected by the Apprentice:

**Observed Actions:**

- [ID: 3] Type "Hello 1" on window "new 2 - Notepad++" (notepad++.exe)
- [ID: 17] Press Return key (Modifiers: None) on window "*new 2 - Notepad++" (notepad++.exe)
- [ID: 3] Type "Hello 2" on window "new 2 - Notepad++" (notepad++.exe)
- [ID: 17] Press Return key (Modifiers: None) on window "*new 2 - Notepad++" (notepad++.exe)
- [ID: 3] Type "Hello 3" on window "new 2 - Notepad++" (notepad++.exe)
- [ID: 17] Press Return key (Modifiers: None) on window "*new 2 - Notepad++" (notepad++.exe)

**Generalizations:**

- [ID: 3] Hello $1, S1(n+1) = S1(n) + 1, S1(n) = 3; Hello $1, S1(n+1) = S1(n) + 1, S1(n) = '3';
- [ID: 17] repeat;

**Repetitions:**

[3, 17]

Upon being detected, this repetition is **processed** by the Assistant. A repetition is said to be processed when the Assistant picks the most probable generalization for each action and predicts the next user iterations. As generalizations are already sorted accordingly to their priority, when bound to a specific id, the Assistant only needs to pick the first one from the
array. Therefore, in the above example, the Assistant picks the most probable generalization for action 3 and action 17. Despite of action 17 only having one generalization, action 3 has two distinct ones. The first one, the most probable, treats the changing portion of the string as an integer, while the second one treats it as a single character. Consequently, the Assistant uses each generalization’s expression and functions to generate the next values, therefore being able to build the narrative shown in Figure 56.

![Blaze Assistant screenshot](image)

**FIGURE 56 – NARRATIVE EXAMPLE #1**

However, there’s another big challenge that the Assistant needs to take care of: How many iterations should be performed, for a given repetition? As we cannot rely too much on domain knowledge, we do not have a trivial way to figure this out. For most common cases, like the one described in the example above, we let the user specify the number of iterations, through Blaze Assistant’s interface.

In more complex cases, like the ones describing operations over the file system, we can resort to user context information to indentify which folder items is he manipulating and how many more operations are needed to fulfill the repetitive task.

For example, imagine that you have a specific folder, let it be “C:\IM”, where your instant messaging client stores your received files. You noticed that many of them share the text token “mary” and it would be nice to organize that folder. Therefore, you would like to move all files containing “mary” in their names to a new folder. In order for the Assistant understand what you intend to do, you have to move a *generic* sample of, at least, 3 files. By generic sample we comprise a sample of files that only have in common a specific set of characteristics. In this case, the three files should only have in common the token “mary” in their names. Also, there should be, at least, two different file extensions.

The following actions describe the mentioned behavior, taking into account the best generation assigned to them by the Observer:

```
Observed Actions:
[ID: 15] File 'Evil Mary.jpg' moved from folder 'C:\IM' to folder 'C:\IM\Mary Jane'.
[ID: 15] File 'exam scan mary.tiff' moved from folder 'C:\IM' to folder 'C:\IM\Mary Jane'.
```
Best Generalization:

[ID: 15] $l_1, \ l_1(n) = C:\IM\*.(*)$ containing "mary".

This generalization denotes that all files in the folder "D:\IM", containing the token "mary", are suited to be moved to the folder "D:\IM\Mary Jane", regardless of their extension. Therefore, the Assistant only needs to query the user context object to gather all file names, contained in "D:\IM", and check which of them contain those tokens. The ones meeting this condition will be added to a list of suited folder items. With this list determined, there is no need to ask the user how many iterations should be performed. An iteration should be performed for each item in the suited folder items list.

Figure 57 depicts the narrative build by the Assistant for the last example. Note that the iterations counter is not shown to the user as the Assistant automatically detects the amount of iterations to perform.

![Figure 57 – NARRATIVE EXAMPLE #2](image)

### 4.3.2 Exploring Alternatives

It is also interesting to point out that the Assistant allows the users to modify each suggestion, in case that the displayed narrative did not really describe what they were intending to do. As every suggestion is displayed along with a "modify" button, when the user presses it, the Automation Editor form shows up, which allows the user to browse for alternative suggestions and modify them.

Figure 58 depicts Blaze assistant. It displays a column on the left, with all actions composing an iteration of the repetitive task. The user can use the right-top pane to browse for alternatives for the currently selected action. As the user cycles through the available alternatives, the right-bottom pane is updated, showing the impact that the alteration takes over the whole automation. Also, the user can edit an alternative if none of them are suitable for what he was aiming for.

These alternatives are generated based on all generalizations available for each action. These are sorted accordingly to the priority of the functions composing them, so that the first alternatives tend to be the best ones.
Also, the Automation Editor allows the user do remove or add new actions on-the-fly, assisted by an easy-to-use wizard.

![Automation Editor](image)

**FIGURE 58 – BLAZE’S AUTOMATION EDITOR**

## 4.4 MAJOR PROBLEMS

Despite all the research and testing, there are a few major problems that persist, which might be subject of future work.

One of them is related to the generalization engine. In order to avoid Blaze from over-generalizing and suggesting automations too soon, we tweaked the pattern search algorithm so that only tree edges with depth of 1 and further might be considered as repetitive prefixes of suffixes. This tweak improved the generalization for most cases but there are a few rare cases which are not properly detected. Improving the generalization engine might help to solve this problem.

Furthermore, there are several well-known recurrent subtasks that are performed by the user but which he does not want to be automated. Examples of these are when the users press alt-tab repeatedly, to cycle between windows, or where they press the backspace key on an instant messaging client. A possible solution for this problem might consist of allowing the user to add actions and applications to a blacklist, which would not be considered for recurrent task detection.

The other problem is related to Blaze Assistant’s interface. It should be easier to perform slight modifications to an automation without having to pass through the Automation Editor. The narrative itself should be, somehow, directly editable, in order to allow the user to instantly exclude a file or modify the search criterion. This problem is not easy to solve as we are stuck with the controls provided by the development environment. However, maybe using HTML and JavaScript to display the narratives, inside Blaze Assistant, would allow us to use more advanced formatting and editing.
5 EVALUATION AND RESULTS

In order to evaluate our approach, we performed both user and performance tests. User tests allowed us to identify performance gains by the users, while executing specific tasks. These gains are traduced in time savings. Also, these tests allowed us to evaluate the ease-of-use, error proneness and command recall.

Performance tests, in turn, allow us to prove that our approach, despite of using an advanced activity monitoring system, does not consume a significant amount of computational resources, working unobtrusively in the users’ computers.

5.1 USER TESTS

In order to evaluate the performance gains provided by Blaze and its ease-of-use, we carried out user tests.

5.1.1 Procedure

The tests were carried out by 20 test subjects. All of them filled a Characterization Survey before the actual tests and a Satisfaction Survey right after. The Characterization survey allowed us to know more about the test sample. From the 20 subjects, 12 were male and 17 had ages between 18 and 25. Every person used a computer daily and 18 of them used Microsoft Windows. Four also used Mac OS and two used Linux. Most of the people used the computer for listening to music, watching movies, viewing emails, browsing the Web and working with office tools. There are a few that also used the computer to play games, perform web design and program.

We verified that 50% of the test subjects were not familiarized with application launchers at all. The most well known application launchers, for the remaining 10 people, are LaunchBar, Launchy, Google Quick Search Box and QuickSilver. 5 people are currently using application launchers, being QuickSilver the most used. Also, 60% found applications launchers to be Useful in their daily life. 20% of them find it to be very useful and the other 20% to be somehow useful.

The tests were all performed on the same computer. The test computer was a laptop with a Core 2 Duo P8600 processor, operating at 2.4 GHz, with 3 GB of RAM memory. Users had to perform two sets of tests. The first one featured 8 tests, which were unique recurrent tasks that might arise from the everyday usage. The second batch of tests was composed by two tests covering some of Blaze’s advanced automation capabilities. Appendix A contains the test guide supplied to the users, which describes every test.

Each test took about 1 hour and 30 minutes to finish. In the first 30 minutes the system was presented to the user, as many of them did not even know what an application launcher was. In the next 30 minutes the user was free to play around with Blaze and was given a set of training examples. In the last 30 minutes he performed the tests described in Appendix A. While the user performed the tests, an Observer jotted down the time taken to perform each task, as well as the number of committed and recovered errors.

The number of committed errors also includes the number of recovered errors. As a recovered error we comprise and error which the user can work around without jeopardizing the test. Therefore, an unrecovered error consists of an error which prevents the user from completing the test. We consider as an error accepting a suggestion from Blaze which does not fully completes the task or discarding a suggestion that would complete the task.
It is important to remark that two times were recorded for each test: the time taken to perform the task using Blaze and the time taken to perform the task without Blaze. User had to perform the same task twice in order for us to have a comparison base. Users first performed the test using Blaze and then they performed it again using whatever tool they could find within the operating system.

Each performed test had a specific goal:

**Tests 1, 2 and 3** – Evaluate Blaze’s capability to automate recurrent tasks regarding the file-system. Each one of these tests covers a specific operation, such as moving, renaming and deleting files.

**Tests 4, 5 and 6** – Evaluate Blaze’s capability to automate recurrent tasks in other applications, such as Microsoft Excel and Word. Also, we want to determine whether it is difficult or not for the user to recall the commands *redo*, in test 4, and *continue*, in test 6. Although these commands might speed up the completion of the task, their usage is optional. Furthermore, test 4 is aimed to evaluate Blaze’s ability to automate recurrent tasks regarding the mouse peripheral.

**Tests 7 and 8** – Evaluate some typical application launcher features, presented by Blaze. However, in opposition to the first 7 tests, test 8 is not focused in performance gains. Instead, test 8 is focused in determining whether it is easy for the user to recall the contextual commands *this* and *url*.

**Tests 9** – Evaluate the ease of recording an adaptive macro and also if the recorded macro can adapt to the window’s position change or not.

**Test 10** – Evaluate how easy it is to the user to edit an automation.

5.1.2 Results

For all tests, there were a few numbers that were recorded:

- Time Spent – Time spent, in seconds, performing the task with Blaze.
- Time Spent w/o Blaze – Time spent, in seconds, performing the task without Blaze.
- Number of Errors – Number of errors performed during the test (including the recovered ones).
- Number of Recovered Errors – Number of errors which the user was able to recover from. A recovered error happens when the user is able to figure out by himself how to overcome the problem.
- Number of Assists – Number of times the user asked for help. This may be due to something he does not remember or know or even to recover from an error.
- Ease of Completion – How easy the user found the task to be accomplished using Blaze. This classification is done through a scale from one to five, in which one means that the task was very hard to accomplish with Blaze, and five means that the task was very easy to accomplish.

1.1.1.1 Overall Results

As we had 20 test subjects, we assumed the sample to have a normal distribution and, therefore, applied Student’s t-test to assess if the time spent to complete the tasks with and without Blaze was significantly different ($\mu_{\text{Blaze}} \neq \mu_{\text{other}}$, 95% CI). The null hypothesis was rejected for all tasks
and, indeed, the times spent to perform the tasks with Blaze are likely to be smaller than without Blaze. Dividing both averages, for a specific test, results in the speed multiplier that describes the speed gains.

Overall, as shown in Figure 59, Blaze allows the user to perform 2.98 times faster repetitive tasks, than with any other solution provided by the operating system. It is easy to note that there are some tasks that are more repetitive than others, like the ones depicted by test 2 and 6. Test 2 consisted on renaming 50 photos, according to a specific sequential pattern. As this task was, probably, the most repetitive of them all, Blaze offered a major speed boost, allowing the user to perform 6 times faster. Now imagine that, instead of 50 photos, there were 300... the speed boost would be incredibly higher. Moreover, the task would not be feasible without Blaze. Test 6 represented a task in which the user just used Blaze to perform 20 iterations of a repetitive sequential pattern, and uses the continue command to perform 20 additional iterations. For just 20 additional iterations, Blaze allows the user to perform 5.15 times faster. Now imagine if instead of 20, there were 500 additional more? The gain would be stupendously greater. Yet again, this task would be almost impossible without Blaze.

Except for test 4, all tests had a success rate of 100%. Test 4 presents a success rate of 75%, which means that there were 5 test subjects that could not finish the test.

In next sections we will fully examine each test and establish the speed gains and the error proneness.

### 5.1.2.1 File-System Repetitions

The following table presents the results for tests one, two and three. These tests represent recurrent tasks regarding the operating system. “Time w” represents the time spend with Blaze; “Time wo” represents the time spent without Blaze; “# Err” represents the number of errors; “# RecErr” represents the number of recovered errors; “# Assists” represents the number of assists; and “Ease” represents the ease-of-use grade assigned by the users.

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th></th>
<th>Test 2</th>
<th></th>
<th>Test 3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>σ</td>
<td>Max</td>
<td>Min</td>
<td>X</td>
<td>σ</td>
</tr>
<tr>
<td>Time w</td>
<td>28,60</td>
<td>7,13</td>
<td>40</td>
<td>10</td>
<td>39,20</td>
<td>12,62</td>
</tr>
<tr>
<td>Time wo</td>
<td>78,94</td>
<td>36,81</td>
<td>188</td>
<td>40</td>
<td>235,25</td>
<td>24,72</td>
</tr>
<tr>
<td>#Err</td>
<td>0,10</td>
<td>0,31</td>
<td>1</td>
<td>0</td>
<td>0,00</td>
<td>0,00</td>
</tr>
<tr>
<td>#RecErr</td>
<td>0,10</td>
<td>0,31</td>
<td>1</td>
<td>0</td>
<td>0,00</td>
<td>0,00</td>
</tr>
<tr>
<td>#Assists</td>
<td>0,00</td>
<td>0,00</td>
<td>0</td>
<td>0</td>
<td>0,00</td>
<td>0,00</td>
</tr>
<tr>
<td>Ease</td>
<td>4,75</td>
<td>0,44</td>
<td>5</td>
<td>4</td>
<td>5,00</td>
<td>0,00</td>
</tr>
</tbody>
</table>

**TABLE 18 – RESULTS FROM TESTS REGARDING FILE-SYSTEM REPETITIONS**

**Test 1** consisted of moving a set of files which had in common a specific text token. Results show that Blaze allowed users to perform this tasks 2.76 times faster. This test had a very low error rate.

**Test 2** consisted of renaming 50 photos according to a specific numeric pattern. This test presented the most significant gains, as it is more repetitive than the others. Blaze allowed users
to perform 6 times faster than any other tool provided by the operating system, which is a very significant boost. Also, if instead of 50 photos, there would be 300 to rename, this task would not be feasible without Blaze and the gains would be tremendously superior. It’s also interesting to point out that this test presented the lowest error rate. Users found it very easy to accomplish and committed no errors.

**Test 3** consisted of deleting a set of files which had in common a few text tokens and the extension. Results reveal that Blaze allowed users to perform this test 2.12 times faster, which is a good result. This test presented a low, still significant, error rate.

Results from **test 3** reveal performance gains of 112% for complex file deletion tasks. This test is the one, regarding the file-system, which is more error prone. The biggest problem with this test was the fact that Blaze Assistant does not allow the user to remove a generalized text token on-the-fly.

For example, in test 3, imagine that the user deletes three text files containing, in their names, the tokens “is”, “xpto”, “2008” and “2009”, like depicted in Figure 60. As the user did not want the “is” token to be considered, he had to close the Assistant, delete and extra file with no “is” in its name, and only then accept the suggestion. Although Blaze allows the user to correct an automation, by providing an addition positive example, Blaze Assistant’s interface should allow the user to instantly remove or add a token.

![FIGURE 60 – NARRATIVE CONTAINING AN EXTRA TOKEN.](image)

### 5.1.2.2 Repetitions on Other Applications

The following table presents the results for tests four, five and six. These tests represent recurrent tasks regarding other applications, such as Microsoft Word and Excel. The legend for this table is the same described in the previous subsection.

<table>
<thead>
<tr>
<th></th>
<th>Test 4</th>
<th></th>
<th>Test 5</th>
<th></th>
<th>Test 6</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\bar{X}$</td>
<td>$\sigma$</td>
<td>Max</td>
<td>Min</td>
<td>$\bar{X}$</td>
<td>$\sigma$</td>
</tr>
<tr>
<td>Time w</td>
<td>70,93</td>
<td>27,62</td>
<td>145</td>
<td>42</td>
<td>28,50</td>
<td>9,81</td>
</tr>
<tr>
<td>Time wo</td>
<td>103,08</td>
<td>24,36</td>
<td>160</td>
<td>71</td>
<td>60,00</td>
<td>25,71</td>
</tr>
<tr>
<td>#Err</td>
<td>0,75</td>
<td>1,37</td>
<td>5</td>
<td>0</td>
<td>0,15</td>
<td>0,49</td>
</tr>
<tr>
<td>#RecErr</td>
<td>0,05</td>
<td>0,22</td>
<td>1</td>
<td>0</td>
<td>0,15</td>
<td>0,49</td>
</tr>
<tr>
<td>#Assists</td>
<td>0,45</td>
<td>0,76</td>
<td>2</td>
<td>0</td>
<td>0,10</td>
<td>0,31</td>
</tr>
<tr>
<td>Ease</td>
<td>3,40</td>
<td>1,10</td>
<td>5</td>
<td>1</td>
<td>4,75</td>
<td>0,55</td>
</tr>
</tbody>
</table>

*TABLE 19 – RESULTS FROM TESTS REGARDING APPLICATION REPETITIONS*
**Test 4** consisted in applying bold to a sequence of lines in five Excel sheets. This was probably the most error prone test of them all. Although it allows the user to perform 1.45 times faster, it proved to be not so easy to accomplish. It had a success rate of 75%, which means that 5 people were not able to finish the test. This is due to the fact that Blaze requires the user to be precise when performing mouse clicks in order to precisely inferring further clicks. People with bad aim simply could not finish the test. Despite of requiring a lot of precision, users found Blaze to be good for automating recurrent tasks in spreadsheets. Also, few users had problems remembering the *redo* command. It is interesting to mention that a user which was not able to use Blaze’s suggestion to accomplish test 4 found a workaround, by using Blaze’s Adaptive Macros. These worked quite well. Also, the test was easier to be performed using the keyboard only, but one of the goals was to evaluate Blaze’s ability to automate recurrent tasks regarding the mouse.

**Test 6** consisted in performing 20 more iterations of the last test, continuing the numeric pattern’s sequence. All users resorted to the continue command, which was expected, which allowed the task to be performed 5.15 times faster with Blaze.

### 5.1.2.3 Typical Application Launcher Tasks and Context-Awareness

The following table presents the results for tests seven and eight. These tests represent tasks regarding typical application launcher features.

<table>
<thead>
<tr>
<th></th>
<th>Test 7</th>
<th></th>
<th>Test 8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>×</td>
<td>σ</td>
<td>Max</td>
</tr>
<tr>
<td>Time w</td>
<td>56.70</td>
<td>19.57</td>
<td>117</td>
</tr>
<tr>
<td>Time wo</td>
<td>84.80</td>
<td>28.10</td>
<td>159</td>
</tr>
<tr>
<td>#Err</td>
<td>0.15</td>
<td>0.37</td>
<td>1</td>
</tr>
<tr>
<td>#RecErr</td>
<td>0.15</td>
<td>0.37</td>
<td>1</td>
</tr>
<tr>
<td>#Assists</td>
<td>0.05</td>
<td>0.22</td>
<td>1</td>
</tr>
<tr>
<td>Ease</td>
<td>4.90</td>
<td>0.31</td>
<td>5</td>
</tr>
</tbody>
</table>

**Table 20 – Results from Tests Regarding Typical Application Launcher Features**

**Test 7** consisted in performing a couple of simple calculations. Blaze allowed user to perform it 1.5 times faster, with a very low error rate. Users found Blaze much easier to use than Window’s calculator. Also, 35% of the test subjects used the help tooltip in order to accomplish the task.

**Test 8** consisted of composing an e-mail with Blaze, using the contextual commands !this and !url. This test was not used to monitor any performance gains. Instead, we tried to indentify how hard it was for the user to remember the contextual commands. All users remembered the commands, which lead us to infer that users have no problems remembering them. Also 55% of the users resorted to the help tooltip in order to use the *email* command.

### 5.1.2.4 Advanced Automation Tasks

The following table presents the results for tests nine and ten. These tests represent tasks regarding some of Blaze’s advanced automation features.
Test 9 consisted of recording an adaptive macro with Blaze, moving the affected window to another position, and reproduce the macro, in order to check if the reproduction was indeed adaptive. Most of the users took less than one minute to record the macro and reproduce it. The resulting behavior was always the expected one.

Test 10 consisted of modifying the suggestion generated by Blaze, in test 4, to apply italic instead of bold to the selected lines. Results demonstrate that it is easy to modify an automation, which can be noticed by the low error rate.

5.1.3 User Satisfaction

After the tests were finished, each test subject had to fill a satisfaction survey. Users had to rate several aspects of the system with a grade from 1 to 5. Table 22 shows users’ classification regarding the utility of some of the most important Blaze’s features.

Table 23 depicts users’ classification regarding Blaze’s performance.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Average Utility</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automate File-System tasks</td>
<td>4.80</td>
<td>0.41</td>
</tr>
<tr>
<td>Automate other Applications’ tasks</td>
<td>4.50</td>
<td>0.61</td>
</tr>
<tr>
<td>Edit Automations</td>
<td>4.50</td>
<td>0.61</td>
</tr>
<tr>
<td>Perform Calculations</td>
<td>4.40</td>
<td>0.68</td>
</tr>
<tr>
<td>Browse for Alternative Suggestions</td>
<td>4.35</td>
<td>0.75</td>
</tr>
<tr>
<td>Record Macros</td>
<td>4.45</td>
<td>0.83</td>
</tr>
<tr>
<td>Use Contextual Commands</td>
<td>4.15</td>
<td>0.88</td>
</tr>
</tbody>
</table>

**TABLE 22 – USER RATING OF BLAZE’S FEATURES**

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Average Classification</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Interface</td>
<td>4.60</td>
<td>0.50</td>
</tr>
<tr>
<td>Assistant Interface</td>
<td>4.30</td>
<td>0.57</td>
</tr>
<tr>
<td>As an Application Launcher</td>
<td>4.85</td>
<td>0.37</td>
</tr>
<tr>
<td>As an Automation tool</td>
<td>4.55</td>
<td>0.60</td>
</tr>
<tr>
<td>Way of Suggesting</td>
<td>4.45</td>
<td>0.51</td>
</tr>
<tr>
<td>Narrative Explicitness</td>
<td>4.20</td>
<td>0.70</td>
</tr>
<tr>
<td>Automation Editor</td>
<td>4.40</td>
<td>0.60</td>
</tr>
<tr>
<td>Macro Recorder</td>
<td>4.55</td>
<td>0.76</td>
</tr>
<tr>
<td>Overall</td>
<td>4.65</td>
<td>0.49</td>
</tr>
</tbody>
</table>

**TABLE 23 – USER RATING OF BLAZE’S INTERFACE**

<table>
<thead>
<tr>
<th>Test 9</th>
<th>Test 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bar{X}$</td>
<td>$\bar{X}$</td>
</tr>
<tr>
<td>58.6</td>
<td>49.8</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>$\sigma$</td>
</tr>
<tr>
<td>15.99</td>
<td>21.15</td>
</tr>
<tr>
<td>Max</td>
<td>Max</td>
</tr>
<tr>
<td>105</td>
<td>117</td>
</tr>
<tr>
<td>Min</td>
<td>Min</td>
</tr>
<tr>
<td>31</td>
<td>28</td>
</tr>
</tbody>
</table>

**TABLE 21 – RESULTS FROM TESTS REGARDING ADVANCED AUTOMATION TASKS**
These tables show that test subjects really enjoyed Blaze as a tool for automating recurrent tasks, especially regarding the file-system. Blaze interface had a very good acceptance but users denoted that generated narratives should be a little clearer. Also, users enjoyed the non-intrusive way of suggesting, although some think that Blaze’s system tray notification may sometimes pass unnoticed.

Since Blaze is already known worldwide, users from everywhere have already reviewed Blaze and stated their satisfaction. Appendix B shows quotes from those reviews, denoting that users really enjoy using Blaze and that its unique automation features and typo tolerance mechanism really make Blaze superior to its counterparts.

5.2 PERFORMANCE TESTS

We carried out a few performance tests in order to demonstrate that Blaze’s ability to be aware of user’s context and monitor every single action he performs, does not consume too many computational resources, which would jeopardize the user experience. These tests were performed in two rounds: intensive use test and idle use test. Intensive use test covered a short period of time where the ten tests performed by users are also performed sequentially, which is prone to consume more computational resources than the idle use test. The idle use tests consisted on letting Blaze monitoring the user for an hour without invoking it.

The test machine was the same for the two rounds. The computer has average hardware, taking into account nowadays standards. The computer has an Intel quad core Q6600, operating at 3.2 GHz, and 4 GB of RAM memory. The operating system was the Microsoft Windows Vista SP1. Also, Blaze had an index with 1109 entries.

5.2.1 Intensive Use Test

This round is aimed to test Blaze’s performance under intensive use and demonstrate that it does not hog on too much computational resources.

5.2.1.1 Experimental Protocol

The test procedure consisted of executing the very same tests performed in user tests. Each one of the ten tests was performed, sequentially, with approximately 45 seconds of interval between them. The testing round lasted for 14 minutes and covered an intensive usage of Blaze, taking into account 3 tests regarding file-system automation, 3 tests regarding application automating, 2 tests for common application launcher features, and two tests covering advanced automation features.

The performance tests cover CPU usage and hit count, for each relevant method, and also memory usage by Blaze’s executable. We consider relevant methods to be those that are prone to consume a lot of CPU Time, due to a high hit count or to the inclusion of costly routines.

5.2.1.2 Results

TABLE 24 shows that the most computationally heavy methods are the ones regarding functions that are called by user demand, such as showing Blaze or predicting text, and the ones regarding activity monitoring, carried out by the Observer. For the first case, a CPU usage less than 0.6% is negligible, as those functions are only called when Blaze has the user focus. In the second case, a CPU usage slightly superior than 1% is insignificant and does not harm the user’s computer performance.
<table>
<thead>
<tr>
<th>Methods</th>
<th>Hit Count</th>
<th>CPU Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blaze - Calling and Dismissal</td>
<td>26</td>
<td>0,596</td>
</tr>
<tr>
<td>Blaze - Index Load &amp; Unload</td>
<td>28</td>
<td>0,409</td>
</tr>
<tr>
<td>Blaze - Predicting Text</td>
<td>127</td>
<td>0,751</td>
</tr>
<tr>
<td>Blaze Indexer - Build Index</td>
<td>2</td>
<td>0,572</td>
</tr>
<tr>
<td>Observer - Monitoring Activity</td>
<td>3729</td>
<td>1,377</td>
</tr>
<tr>
<td>Apprentice - Build Suffix-Tree</td>
<td>2633</td>
<td>0,342</td>
</tr>
<tr>
<td>Assistant - Generate Suggestions</td>
<td>2633</td>
<td>0,355</td>
</tr>
</tbody>
</table>

TABLE 24 – TEST RESULTS FROM INTENSIVE USE TEST

Also, results show that our pattern detection algorithm is efficient, as the Apprentice only used 0.34% of CPU time to detect repetitive tasks. Furthermore, The Assistant only used 0.36% of CPU to generate suggestions capable of automating recurrent tasks, which is a very positive and validating result.

The heaviest operations that run in the background are, without any doubt, the ones regarding indexation. The reader can notice that building the index got almost 0.6% of CPU usage for only two method calls. That is why indexing is not done continuously and we confined these methods to a standalone executable, Blaze Indexer, which is periodically called by Blaze’s main process. Furthermore, while a new index is being built, Blaze keeps a backup of the old one, making it always available to the user and not affecting Blaze’s performance.

Blaze also proved to perform well in terms of memory usage. Given the 1109 indexed items, Blaze had an average memory consumption of 37 megabytes, and a maximum of 45 Megabytes. Although this might seem a little too much, when compared to Launchy’s average of 6 Megabytes, we need to take into account that Blaze is continuously monitoring the user and that it takes his context into account all the time. When compared to other applications such as Mozilla Firefox, which consumes 80 Megabytes with only 5 tabs opened, just being idle, we consider Blaze to have a very reasonable memory consumption.

5.2.2 Idle Test

This round is aimed to test whether Blaze has or not a significant weight, in terms of computer resources, just for monitoring the user in the Background, without being invoked a single time.

5.2.2.1 Experimental Protocol

This test consisted of executing Blaze for almost 1 hour, more precisely 57 minutes, without invoking it. The user carried out a regular computer usage, including text processing in Microsoft Word and notepad, browsing the Web, watching videos on YouTube and chatting online with Windows Live Messenger.

5.2.2.2 Results

The reader can easily notice, taking into account TABLE 25, that the methods that most computational resources consumed were the ones regarding keyboard monitoring. This is due to the fact that the user used mostly the keyboard in his activity, mainly because of the text editing tasks. Although methods regarding activity monitoring were invoked 11154 times, they only consumed around 1.3% of CPU time, which can be considered negligible for a background monitoring process.
<table>
<thead>
<tr>
<th>Method</th>
<th>Hit Count</th>
<th>CPU Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer - Monitor Activity</td>
<td>11154</td>
<td>1,302</td>
</tr>
<tr>
<td>Apprentice - Build Suffix Tree</td>
<td>11148</td>
<td>0,406</td>
</tr>
<tr>
<td>Assistant - Generate Suggestions</td>
<td>11148</td>
<td>0,018</td>
</tr>
</tbody>
</table>

**TABLE 25 – TEST RESULTS FROM IDLE USE TEST**

The remaining methods present a CPU usage less than 0.5% which proves that they have no significant impact over the performance of the user’s computer.

5.3 DISCUSSION

Overall, test results were very positive. User tests showed that Blaze is an excellent tool for automating the unique recurrent tasks that arise from everyday usage. On average, Blaze allowed users to perform repetitive tasks 2.98 times faster, which is an excellent result, with a very low error rate and a success rate of 100% for all tests except one. Moreover, gains are much higher as the more repetitive a task is. For instance, a task regarding changing the name of 50 files may be performed 6 times faster or more using Blaze. If instead of 50 there were 300 files to rename, gains would be much higher. As a matter of fact, the task would not be feasible without Blaze. This is why users found Blaze to be particularly useful for automating tasks regarding the file-system and text processors. Also, tests denoted that users found the redo and continue automation commands easy to remember and use, as well as the !this and !url context commands. Furthermore, all users in our tests were novice users, with no previous knowledge of Blaze. With continued usage, they would undoubtedly be able to improve their performances significantly, after growing accustomed to the wide range of capabilities Blaze makes available.

Performance tests also demonstrate that Blaze does not consume significant computational resources and, therefore, does not jeopardize the user’s experience. Blaze’s CPU time consumption is very low for both idle and intensive use circumstances, which allow us to consider it negligible. Although presenting a significant memory consumption of 37 Megabytes, on average, we consider it to be a very reasonable amount, taking into account that Blaze is always monitoring user’s activity. Furthermore, 37 Megabytes of memory are nothing for nowadays’ computers and when compared to other applications, such as email clients or browsers, which consume over 80 Megabytes of memory just for being idle.

Blaze had a great acceptance by the public, which praised its ability to tolerate typos and automate recurrent tasks. This serves as an informal validation to our approaches. Moreover, currently there is not any other system capable of such deeds.
6 CONCLUSIONS AND FUTURE WORK

Nowadays computer users are constantly faced with recurrent tasks to which they feel the need for automation. Some tasks are well known and common to all users, like application launching, performing calculations, composing email, and so on. These are easily automated by applications launchers. However, there are also recurrent tasks that are unique, which emerge from everyday usage. There is not a tool with a graphical interface capable of trivially automating these tasks. Users often resort to scripting languages and macro editors to accomplish those tasks, but such measures require a lot of knowledge and have a steep learning curve.

There are a few approaches that adopt the Programming by Example technique, like SMARTedit and Creo, which are able to automate recurrent tasks in specific applications. SMARTedit is a text editor while Creo operates over Internet Explorer. Both embrace the macro recorder metaphor as they need the user to start and stop the recording process in order to capture a sample of actions which composes an iteration from the repetitive task. This is problematic as the user needs to be directly involved in the programming procedure. APE overcomes this issue by adopting the Implicit Programming by Example technique. This consists of capturing the sample of actions, which composes an iteration from the repetitive task, without requiring user intervention. However, APE also operates over a single application and its repetitive pattern detection algorithm is not able to detect a few kinds of repetitions.

In order to bridge these gaps, we also adopted the Implicit Programming by Example technique and developed a new system capable of monitoring user activity and inferring repetitive patterns, operating system widely. The user is not required direct intervention, as he does not have to start and stop any recording process. The system is composed by three automations agents: the Observer, the Apprentice and the Assistant.

The Observer is responsible for monitoring any user activity, regarding the file-system or the mouse and keyboard peripherals. This agent is also responsible for indentifying which user actions are similar and establish generalizations between them. The Apprentice, in turn, is responsible for indentifying repetitive patters of actions, using an algorithm, based on suffix-trees, to search for the longest non-overlapping subsequence of actions. This algorithm is more adequate than the KMR one, used by APE, as it solves the problem in linear time and is able to detect variable and conditional loops, in actions regarding the file-system. The Assistant uses the repetitive tasks detect by the Apprentice and the generalizations built by the Observer to generate comprehensive narratives and automations capable of automating recurrent tasks. The Blaze Assistant interface uses rtf tags in order to provide rich text formatting, which focuses the user in the changes that are going to be performed by Blaze.

Blaze also embraces the application launcher metaphor. It allows the user to type in commands to launch applications, perform calculations, web search or even compose an email. Blaze outrivals its counterpart application launchers by featuring a typo tolerance mechanism, which allow the user to mistype commands, and by taking the user context into account.

Our tests showed that users could perform repetitive tasks 3 times faster, while using Blaze. However, the more repetitive the task is, the higher is the gain. For instance, in the test concerning changing files’ names, the user was able to perform 6 times faster. If there were 250 photos to rename instead of just 50, the gains would be much superior. Furthermore, our tests also proved that users did not have problems remembering the automation commands, such as continue or redo, and the contextual commands, such as !this and !url.
Our performance tests proved that, besides being continuously monitoring user activity, Blaze uses insignificant amounts of CPU time and acceptable amounts of RAM memory, thus not jeopardizing user experience. Blaze consumes an average of 37MB which is a very small amount when compared to other applications like Mozilla Firefox, which consume over 80 Megabytes just for being idle, with 5 tabs opened.

It is also important to denote that Blaze offers a lot of room for extensibility, due to its support for both dynamic link library plugins and IronPython plugins. The first way offers more flexibility, by allowing several commands to be added at once, dynamically. Also, it allows new indexation methods to be added, which may offer new ways of collecting meta-data from files. The IronPython scripts, in turn, are easier to do and require no code compilation. The user just needs some knowledge about Python and .NET and the notepad. Furthermore, IronPython scripts can still use the powerful ContextLib, which provides means to access the user’s contextual information.

Also, user reviews throughout the globe were very positive, praising Blaze’s innovative features. The mere fact that users commended Blaze’s ability to automate recurrent tasks and to tolerate typos is a significant validation to our approach.

In terms of future work, there is some room for improvement in the pattern detection algorithm, more precisely for the generalization system. A few enhancements could be done in order to allow Blaze to detect nested loops and to be sensitive to the few rare cases that are not correctly detected. Furthermore, the Assistant could also comprise a few heuristics to overcome overgeneralization, which would allow well known cases not to be considered as recurrent tasks. Also, a blacklist of actions and applications could also be implemented in order to allow the user to parameterize special cases that should not be considered as repetitions.

There is also some room for improvement in Blaze Assistant’s interface, as users stated that the narratives could be clearer. Also, it should also allow the user to perform quick modifications, such as adding or removing a generalized token and adding or removing a file. In order to overcome this issue, maybe a dynamic interface could be generated, resorting to html and Javascript or maybe to Windows Presentation Foundation.

In terms of additional functionality, we intend to implement a cronjob system, which will allow users to schedule commands to be performed at certain times or even periodically. These commands may also resort to contextual information. For instance, it will allow users to schedule Blaze to always turn the speakers’ volume down when their computer is connected to their workplace’s wireless network and turn the volume up when connected to their home’s wireless network.


First of all, thank you so very much for cooperating with me. Your participation is very valuable because you are helping me testing and improving this innovative application. With your help, Blaze will get even better and will be able to make life easier for thousands of users worldwide.

You are going to perform simple tests, which won’t take too much of your time. There will be two test rounds, the first one with 8 tests and the second one with 2 tests. Each test is described by a brief scenario and followed by a set of actions with you should follow in order to accomplish the test.

If you have any doubt, regarding some technical detail, don’t hesitate asking me. Take your time and don’t rush.

You may turn the page.
FIRST ROUND

The first wave consists of 8 tests, which describe unique recurrent tasks that might arise from the everyday usage. You should perform each test twice: Once using Blaze and once without it, resorting only to any tool provided by the operating system.

Relax and take your time.

TEST 1

SCENARIO

Your favorite instant messaging client, like Windows Live Messenger, stores your received files on a specific folder. You have received many files from Mary Jane, which is very narcissistic friend of yours. She uses to write her name on every file she sends, such as "Report Mary 01.doc". As you have received a lot of files from her, you would like to move all those files to a subfolder, in order to clean up the mess.

ACTIONS

1. Go to folder “Test 1”.
2. Create a subfolder with a name of your choice.
3. Move all files with the token “mary” in their names to the newly created folder.

TEST 2

SCENARIO

You have just downloaded, from your digital camera, 50 photos taken on a summer party. These photos have meaningless names for a human being, since cameras usually assign strange names to them, somehow related to the date. You would like to change the names of the photos to meaningful ones, such as "Summer Party 01", “Summer Party 02”, “Summer Party 03”, and so on.

ACTIONS

1. Return to the root directory and go to folder "Test 2".
2. Change the name of all photos, such as “03082008(001).jpg” to “Summer Party 01.jpg”, “03082008(002).jpg” to “Summer Party 02.jpg”, “03082008(003).jpg” to “Summer Party 03.jpg”, and so on.
TEST 3

SCENARIO

Imagine that you use to take notes about the performance of each player from the XPTO football team. This is your favorite football team and you don’t miss a single game. Hence, you have a folder on your computer where you store the progress of each player during the championships. You initially jot it down on a text file (.txt) and, latter, write it up to a Word file (.doc). Once the 2008-2009 championship is over and the 2009-2010 has already begun, you feel the need to organize the folder, as there are too many drafts from both championships making a mess. Therefore, you would like to delete all drafts from the previous championship, without harming the other important text files present in the same folder.

ACTIONS

1. Return to the root directory and go to folder “Test 3”.
2. Delete all text files (.txt) containing the tokens “xpto”, “2008” and “2009” in their names.

TEST 4

SCENARIO

Imagine that you and your crew carried out some laboratory experiments and that the results were noted down on five Microsoft Excel spreadsheets. As you are sharing the results with your colleagues, you want to highlight the lines mentioning whether the test subjects have taken or not their medicine. Therefore, you would like to apply bold to each line beginning with “Medicine”, on each one of the five sheets.

ACTIONS

1. Return to the root directory and go to folder “Test 4”.
2. Open the file ExperimentBook.xlsx.
3. Apply bold to each line beginning with “Medicine 1”, “Medicine 2”, “Medicine 3” and so on, for all five sheets. Use the Ctrl+B hotkey to apply bold.

TEST 5

SCENARIO

You need to discuss with your colleagues twenty cases of study related to the previous experiment. You would, therefore, like to created a document on Microsoft Word and write “Case 1 –”, “Case 2 –”, “Case 3 –”, and so on, separated by a new line, in order to latter write notes for each case.
ACTIONS

1. Open Microsoft Word.

TEST 6

SCENARIO

After you have completed the previous task, you remembered that instead of twenty cases it were forty. Therefore, in the same document, opened in the previous test, write the remaining cases separated by a new line.

ACTIONS

2. Close the document without saving any changes.

TEST 7

SCENARIO

Your nephew has some questions regarding math and is asking you for help with a few exercises. As you are a little busy in the moment, you would like to use a tool that is able to quickly solve math problems for you. We wouldn’t like to keep the boy waiting, right?

ACTIONS

1. Solve the following problems:
   a. \((8^4 \times -1 + 8^5)/16 - 1024)/12 - 22\)
   b. Remainder of dividing 400 by 35.

TEST 8

SCENARIO

You just woke up and turned you PC on to browse for some fresh news. You just got to your favorite news website and found the top headline to be very interesting. Therefore, you would like to share the article with your friend Antony, by email, with the headline in the subject and the page address in the body.
ACTIONS

1. Browse to the webpage http://www.neowin.net/.
2. Browse to the first headline on the page.
3. Create an email with the following data:
   a. Recipient: antony@provider.com
   b. Subject: article headline.
   c. Body: webpage’s address.
4. Discard the created email.
SECOND ROUND

This second round is composed by 2 tests which should be executed with Blaze only.

Don't worry, it’s almost over!

TEST 9:

SCENARIO

You share your computer with a roommate and, whence, you wouldn’t like him to peek on your private data, such as the web pages you browsed or your banking accounts. You would like to record a macro with Blaze, which would open your web browser, clean the private data and then close the application.

ACTIONS

1. Record a macro with Blaze comprising the following actions:
   a. Open Mozilla Firefox.
   b. Clear the private data.
   c. Close Mozilla Firefox.
2. Open Mozilla Firefox again.
3. Move Mozilla Firefox’s window to other position.
5. Invoke the recorded macro and observe its execution.

TEST 10:

SCENARIO

Perform the same exercise described in Test 4. However, modify the suggestion so that it will apply italic to each line instead of bold.

ACTIONS

1. Go to folder “Test 4”.
2. Open the file ExperimentBook.xlsx.
3. Apply **bold** to the first lines beginning with ”Medicine”, using the **Ctrl+B** hotkey.
4. Modify Blaze’s suggestion so that *italic* will be applied instead of bold.
5. Execute the modified automation and observe the result.

Thank you so much for your participation!
Currently there are many reviews of Blaze spread throughout the Web. Four major reviews can be found at Lifehacker\(^42\), makeuseof.com\(^43\), Instant Fundas\(^44\) and Download Squad\(^45\), which are renowned technological websites, with thousands of daily readers. These reviews are highlighting Blaze’s success as an automating mechanism and as an innovative application launcher.

**LIFEHACKER**

Lifehacker review\(^46\) praised Blaze’s ability to automate recurrent tasks, which is, with no doubt, its most innovative feature:

> “Not only does Blaze act as a Launchy-style application launcher, but it also monitors your work and leaps into action when it appears that it could help you by automating the task.”

Readers from that website posted lots of comments, also praising Blaze’s innovation. Many comments denoted Blaze’s potential superiority over its counterparts:

> “I’m very tempted to leave executor now for this, but executor is customized to my liking so much that it’s gonna be hard, but the spell checker and automation in this program look phenomenal.”

**MAKEUSEOF.COM**

The makeuseof.com review\(^47\) was quite complete and covered most of Blaze’s features. It denoted Blaze’s superiority when compared to other applications launchers:

> “Application launchers generally give way when you accidentally type in a character that is not a part of the application name, Blaze handles this scenario better than others. Let’s face it we all make typos and having a program smart enough to still guess the correct application really helps.”

It also praised Blaze’s automation capabilities, which, currently, are unique:

> “Blaze can monitor and automate anything from entering text to mouse clicks to renaming files to creating folders. In short, if your previous few actions have something in common, Blaze will most likely come up with a suggestion to do it for you.”

The makeuseof.com’s review also gone a step further and also covered some advanced features, regarding Blaze’s Automation Editor. It was pretty impressive how users explored this further, as these features were not even documented yet:

“Blaze also lets you create new automations from scratch or modify the suggested automation to your liking. You can do so by clicking on ‘Create a new Automation’ or ‘Modify this Automation’ buttons. You will be shown the Automation editor, in here you can modify the automations, add, subtract and modify actions to your liking.”

INSTANT FUNDAS

The Instant Fundas’s review was similar to the ones described above, highlighting Blazes capabilities as an application launcher, capable of bearing typos, and as an automation agent:

“Blaze has the ability to bear typos and, therefore, you don’t have to be so careful while typing a command. You can also type program names in shortcut, and it almost always recognizes what you are trying to access.”

“It runs in the background and whenever it detects a repetitive action, it offers to complete the task for you. You don’t have to teach him a thing. Blaze learns by itself and offers suggestions on his own. You just have to choose whether to accept its help or not.”

This review also addressed Blaze as a macro recording, citing Blaze’s ability to record macros that can adapt to new situations.

DOWNLOAD SQUAD

The Download Squad’s review was also very interesting and enthusiastic, giving a great focus on typo tolerance:

“For starters, it’s smart enough to find what you’re looking for even if you should happen to stumble over your keys a bit (e.g. the butchered Photoshop entry above). Lurking behind the scenes is a powerful automation handling system and plugin support (skin support is coming soon).”

“I can see myself enjoying Blaze on my netbook even without all the advanced functions. Since my typing isn’t quite as accurate on the scaled-down keys, it’s nice to know that Blaze will still be able to launch apps even if I hit a few wrong keys here or there.”

However, Blaze was also praised as an automation mechanism:

“The automator watches silently in the background, waiting to observe you performing repetitive tasks. When it spots something, Blaze will alert you and offer to create an action to handle the task next time. On the project page, the author demonstrates this functionality by renaming a directory full of photos. Blaze can also record macros -- so you can just press record, start doing what you need to do, and presto! You’ve got a new action.”

This review even questions if could Blaze be the best open source application launcher for windows, as shows the headline:

“Could open source Blaze be the best Windows application launcher ever?”

---

At Blaze’s website on SourceForge, users from everywhere are starting to post their own reviews. Users are finding Blaze to be superior to other application launchers in many aspects, especially regarding the ones in which we were supposed to innovate: recurrent task automation and typo tolerance.

Examples of users’ reviews are:

“most excellent alternative to launchy and find&run robot. great project!”

“It helped me a lot with the Assistant and the Launch box.”

“bests Launchy and other windows launchers in speed and responsiveness. Now, if only I can find a way to update twitter through it, it’ll be awesome.”

“By far the best of the bunch when you consider the automation tools you provide. You need to multithread whatever is happening when you close the automation window though...it’s somewhat slow in the UI. Otherwise great, just uninstalled executor!”

“I’ve used Launchy for a while now and this is right up there with must have apps. the suggestions and command features are especially good”

50 http://blaze-wins.sourceforge.net/ (Last visited in: September 14th, 2009)